Chapter 1

Values, Functions, and Statements

Abstract In the previous chapter, you got an introduction to the fundamental concepts of booleans, numbers, characters, and strings, how to perform calculations with them, and some of the limitations they have on the computer. This allows us to perform simple calculations as if the computer was an advanced pocket calculator. In this chapter, we will look at how we can:

- Make assign values to names to make programs that are easier to read and write.
- Organise lines of code in functions, to make the same line reusable, and to make programs shorter, and easier to understand.
- Use operators as functions.
- Conditionally execute code.
- How we can simulate the computer's execution of code by tracing-by-hand, to improve our understanding of, how it interprets the code, we write, and to find errors.

Examples of problems, you can solve, after having read this chapter are:

- Write a function that given the parameters a and b in $f(x) = ax + b$, find the value of x when $f(x) = 0$.
-

In this chapter, we will see how we can bind expressions to identifiers either as new constants, functions, or operators, how this saves time when building large programs, and how this makes programs easier to read and debug. As an example, consider the following problem,

Problem 1.1

or given set constants a, b, and c, solve for x in

$$ax^2 + bx + c = 0 \tag{1.1}$$

To solve for x we use the quadratic formula from elementary algebra,

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}, \tag{1.2}$$

which gives the general solution for any values of the coefficients. Here, we will assume a positive discriminant, $b^2 - 4ac > 0$. In order to write a program where the code may be reused later, we define a function

```
discriminant : float -> float -> float -> float
```

that is, a function that takes 3 arguments, `a`, `b`, and `c`, and calculates the discriminant. Likewise, we will define

```
positiveSolution : float -> float -> float -> float
```

and

```
negativeSolution : float -> float -> float -> float
```

that also take the polynomial's coefficients as arguments and calculate the solution corresponding to choosing the positive and negative sign for ± in the equation. Details on function definition is given in Section 1.2. Our solution thus looks like Listing 1.1. Here, we have further defined names of values `a`, `b`, and `c` which are used as inputs to our functions, and the results of function application are bound to the names `d`, `xn`, and `xp`. The names of functions and values given here are examples of identifiers, and with these, we may reuse the quadratic formulas and calculated values later, while avoiding possible typing mistakes and reducing the amount of code that needs to be debugged.

The use of identifiers is central in programming. For F#, not to be confused with built-in functionality, identifiers must follow a specific set of rules:

Identifier

Listing 1.1 identifiersExample.fsx:
Finding roots for quadratic equations using function name binding.

```
let discriminant a b c = b ** 2.0 - 4.0 * a * c
let positiveSolution a b c = (-b + sqrt (discriminant a b c))
   / (2.0 * a)
let negativeSolution a b c = (-b - sqrt (discriminant a b c))
   / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let d = discriminant a b c
let xp = positiveSolution a b c
let xn = negativeSolution a b c
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has discriminant %A and solutions %A and %A" d
   xn xp
```

```
$ dotnet fsi identifiersExample.fsx
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

- Identifiers are used as names for values, functions, types etc.
- They must start with a Unicode letter or underscore '_', but can be followed by zero or more letters, digits, and a range of special characters except for SP, LF, and CR (space, line feed, and carriage return). See **??** for more on codepoints that represent letters.
- They can also be a sequence of identifiers separated by a period.
- They cannot be keywords, see Table 1.1.

Examples of identifiers are: `a`, `theCharacter9`, `Next_Word`, `_tok`, and `f.sharp.rocks`. Since programmers often work in a multilingual environment dominated by the English language it is advisable to **restrict identifiers to use letters from the English alphabet, numbers, periods, and '_'.** However, the number of possible identifiers is enormous. The full definition refers to the Unicode general categories described in Appendix **??**, and there are currently 19.345 possible Unicode code points in the latter category and 2.245 possible Unicode code points in the special character category. ★

Identifiers may be used to carry information about their intended content and use, and a careful selection of identifiers can aid programmers to communicate thoughts about the code. Thus, identifiers are often a word or several concatenated words conveying some relevant meaning. For example, in the function definition `let discriminant a b c = b ** 2.0 - 4.0 * a * c`, the function identifier has

Type	Keyword
Regular	abstract, and, as, assert, base, begin, class, default, delegate, do, done, downcast, downto, elif, else, end, exception, extern, false, finally, for, fun, function, global, if, in, inherit, inline, interface, internal, lazy, let, match, member, module, mutable, namespace, new, null, of, open, or, override, private, public, rec, return, sig, static, struct, then, to, true, try, type, upcast, use, val, void, when, while, with, and yield.
Reserved	atomic, break, checked, component, const, constraint, constructor, continue, eager, fixed, fori, functor, include, measure, method, mixin, object, parallel, params, process, protected, pure, recursive, sealed, tailcall, trait, virtual, and volatile.
Symbolic	let!, use!, do!, yield!, return!, |, ->, <-, ., :, (,), [,], [<, >], [|, |], {, }, ', #, :?>, :?, :>, .., ::, :=, ;;, ;, =, _, ?, ??, (*), <@, @>, <@@, and @@>.
Reserved symbolic	~ and `

Table 1.1 Table of (possibly future) *keywords* and symbolic keywords in F#.

been chosen to be `discriminant`. F# places no special significance on the word 'discriminant', and the program would work exactly the same had the function been called `let f a b c = b ** 2.0 - 4.0 * a * c`. However, to programmers, the word 'discriminant' informs us of the intended role of the function and thus
★ is much preferred. This is a general principle: **identifier names should be chosen to reflect their semantic value.**. The arguments `a`, `b`, and `c` are short, but adheres to a textbook tradition of elementary algebra. Again, we might as well have used, `let discriminant c a b = a ** 2.0 - 4.0 * c * b`, which is semantically identical to the original expression, but due to tradition, this would con-
★ fuse most readers of the code. Thus, **identifier names should be chosen consistently with the readers' traditions.** Finally, identifiers are often concatenations of words, as `positiveSolution` in Listing 1.1. Concatenations can be difficult to read. Without the capitalization of the second word, we would have had `positivesolution`. This is readable at most times but takes longer time to understand in general. Typical solutions are to use a separator, such as `positive_solution`, *lower camel case* also known as *mixed case* as in the example `positiveSolution`, and *upper camel case* also known as *pascal case* as `PositiveSolution`. In this book, we use the lower camel case except where F# requires a capital first letter. Again, the choice does not influence what a program does, only how readable it is to a fellow programmer.
★ The important part is that **identifier names consisting of concatenated words are often preferred over names with few characters, and concatenation should be emphasized, e.g., by camel casing.** Choosing the length of identifier names is a balancing act, since when working with large programs, very long identifier names can be tiresome to write, and a common practice is that the length of identifier names is proportional to the complexity of the program. I.e., complex programs use long names, simple programs use short names. What is complex and what is simple is naturally in the eye of the beholder, but when you program, remember that a future reader of the program most likely has not had time to work with the problem as long

as the programmer, thus **choose identifier names as if you were to explain the meaning of a program to a knowledgeable outsider.** ⋆

Another key concept in F# is expressions. An expression can be a mathematical expression, such as $3 * 5$, a function application, such as $f\,3$, and many other things. Central in this chapter is the binding of values and functions to identifiers, which is done with the keyword `let`, e.g., `let a = 1.0`.

Expressions are the main workhorse of F# and have an enormous variety in how they may be written. We will in this book gradually work through some of the more important facets.

Expressions

- An Expression is a computation such as `3 * 5`.
- They can be value bindings between identifiers and expressions that evaluate to a value or a function, see Sections 1.1 and 1.2.
- They can be `do`-bindings that produce side-effects and whose results are ignored, see Section 1.2
- They can be assignments to variables, see Section 1.1.
- They can be a sequence of expressions separated with the "`;`" lexeme.
- They can be annotated with a type by using the "`:`" lexeme.

Before we begin a deeper discussion on bindings, note that F# adheres to two different syntaxes: *verbose* and *lightweight*. In the verbose syntax, newlines and whitespaces are generally ignored, while in lightweight syntax, certain keywords and lexemes may be replaced by newlines and whitespaces. The lightweight syntax is the most common and will be used in this book.

1.1 Value Bindings

Binding identifiers to literals, or expressions that are evaluated to be values, is called *value-binding*, and examples are `let a = 3.0` and `let b = cos 0.9`. Value bindings have the following syntax:

Listing 1.2: Value binding expression.

```
let <valueIdent> = <bodyExpr>
```

The *let* keyword binds a value-identifier with an expression. The above notation means that <valueIdent> is to be replaced with a name and <bodyExpr> with an expression that evaluates to a value. The binding *is* available in later lines until the end of the scope it is defined in.

The value identifier is annotated with a type by using the “:” lexeme followed by the name of a type, e.g., int. The “_” lexeme may be used as a value-identifier. This lexeme is called the *wildcard pattern*, and for value-bindings it means that <bodyExpr> is evaluated, but the result is discarded.

For example, letting the identifier p be bound to the value 2.0 and using it in an expression is done as shown in Listing 1.3. Note that the expression 3.0 ** p must

Listing 1.3 letValueLightWeight.fsx:
Lightweight syntax does not require the in keyword, but the expression must be aligned with the let keyword.

```
let p = 2.0
do printfn "%A" (3.0 ** p)
```

```
$ dotnet fsi letValueLightWeight.fsx
9.0
```

be put in parentheses here, to force F# to *first* calculate the result of the expression and *then* give it to printfn to be printed on the screen. The same expression in interactive mode will also show with the inferred types, as shown in Listing 1.4. By

Listing 1.4: Interactive mode also outputs inferred types.

```
> let p = 2.0
do printfn "%A" (3.0 ** p);;
9.0
val p: float = 2.0
val it: unit = ()
```

the val keyword in the line val p : float = 2.0, we see that p is inferred to be of type float and bound to the value 2.0. The inference is based on the type of the right-hand-side which is float. Identifiers may be defined to have a type using the “:” lexeme, but the types on the left-hand-side and right-hand-side of the “=” lexeme must be identical. Mixing types gives an error, as shown in Listing 1.5. Here, the left-hand side is defined to be an identifier of type float, while the right-hand side is a literal of type integer.

A key concept of programming is *scope*. When F# seeks the value bound to a name, it looks left and upward in the program text for its let-binding in the present or

Listing 1.5 letValueTypeError.fsx:
Binding error due to type mismatch.

```
let p = 2.0
do printfn "%A" (3 ** p)
```

```
$ dotnet fsi letValueTypeError.fsx


letValueTypeError.fsx(2,18): error FS0001: The type 'int'
   does not support the operator 'Pow'
```

higher scopes. This is called *lexical scope*. Some special bindings are mutable, that is, they can change over time in which case F# uses the *dynamic scope*. This will be discussed in **??**.

Scopes are given levels, and scopes may be nested, where the nested scope has a level one lower than its parent. F# distinguishes between the top and lower levels, and at the top level in the lightweight syntax, redefining values is not allowed, as shown in Listing 1.6. However, using parentheses, we create a *code block*, i.e., a

Listing 1.6 letValueScopeLowerError.fsx:
Redefining identifiers is not allowed in lightweight syntax at top level.

```
let p = 3
let p = 4
do printfn "%A" p;
```

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

letValueScopeLowerError.fsx(2,5): error FS0037: Duplicate
   definition of value 'p'
```

nested scope, as demonstrated in Listing 1.7. In lower scope-levels, redefining is

Listing 1.7 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

```
let p = 3
(
  let p = 4
  do printfn "%A" p
)
```

```
$ dotnet fsi letValueScopeBlockAlternative3.fsx
4
```

★ allowed but **avoid reusing names unless it's in a deeper scope.** Inside the block in Listing 1.7 we used indentation, which is good practice, but not required here.

Defining blocks is used for controlling the extent of a lexical scope of bindings. For example, bindings inside a nested scope are not available outside, as shown in Listing 1.8. Nesting is a natural part of structuring code, e.g., through function

Listing 1.8 letValueScopeNestedScope.fsx:
Bindings inside a scope are not available outside.

```fsharp
let p = 3
(
  let q = 4
  do printfn "%A" q
)
do printfn "%A %A" p q
```

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

letValueScopeNestedScope.fsx(6,22): error FS0039: The value
   or constructor 'q' is not defined.
```

definitions to be discussed in Section 1.2 and flow control structures to be discussed in **??**. Blocking code by nesting is a key concept for making robust code that is easy to use by others, without the user necessarily needing to know the details of the inner workings of a block of code.

1.2 Function Bindings

A function is a mapping between an input and output domain, as illustrated in Figure Figure 1.1. A key advantage of using functions when programming is that

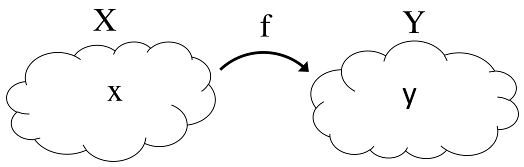

Fig. 1.1 A function $f : X \rightarrow Y$ is a mapping between two domains, such that $f(x) = y$, $x \in X$, $y \in Y$.

they encapsulate code into smaller units, that may be reused and are easier to find errors in. F# is a functional-first programming language and offers a number of alternative methods for specifying parameters, which will be discussed in this section. Binding identifiers to functions follows a syntax similar to value-binding,

Listing 1.9: Function binding expression

```
let <ident> (<arg> {<arg>}) | () =
  <expr>
```

Here `<ident>` is an identifier and is the name of the function, `<arg>` is zero or more identifiers, that bind to the value used when calling the function, and which is to be used in the *body* of the function, the expression `<expr>`. The `|` notation denotes a choice, i.e., either that on the left-hand side or that on the right-hand side. Thus `let f x = x * x` and `let f () = 3` are valid function bindings, but `let f = 3` would be a value binding, not a function binding. The arguments and the function may be annotated with a type, in which case for arguments we write

Listing 1.10: Function binding expression

```
let <ident> ((<arg> : <type>) {(<arg> : <type>)} : <type>) | (
    () : <type>) =
  <expr>
```

where `<type>` is a name of an existing type. The argument types are given in parentheses, and the return type is given last.

Functions are a key concept in F#, and in this chapter, we will discuss the very basics. Recursive functions will be discussed in **??** and higher-order functions in **??**.

The function's body can be placed either on the same line as the `let`-keyword or on the following lines using indentation. An example of defining a function and using it in interactive mode is shown in Listing 1.11. Here we see that the function is

Listing 1.11: An example of a binding of an identifier and a function.

```

> let sum (x : float) (y : float) : float = x + y
let c = sum 357.6 863.4
do printfn "%A" c;;
1221.0
val sum: x: float -> y: float -> float
val c: float = 1221.0
val it: unit = ()
```

interpreted to have the type `val sum : x:float -> y:float -> float`. The "->" lexeme means a mapping between sets, in this case, floats. The function is also a higher-order function, to be discussed in detail below, and here it suffices to think of `sum` as a function that takes 2 floats as arguments and returns a float.

Not all types need to be declared, just a sufficient number for F# to be able to infer the types for the full statement. For the example, one is sufficient, and we could just have declared the type of the result, as in Listing 1.12. Or even just one of the arguments, as in Listing 1.13. In both cases, since the + *operator* is only defined for

**Listing 1.12 letFunctionAlterantive.fsx:
Not every type needs to be declared.**

```
let sum x y : float = x + y
```

**Listing 1.13 letFunctionAlterantive2.fsx:
Just one type is often enough for F# to infer the rest.**

```
let sum (x : float) y = x + y
```

operands of the same type, declaring the type of either arguments or result implies the type of the remainder.

Arguments need not always be inferred to types, but may be of the generic type when *type safety* is ensured, as shown in Listing 1.14. Here, the function `second`

Listing 1.14: Type safety implies that a function will work for any type.

```
> let second x y = y
let a = second 3 5
do printfn "%A" a
let b = second "horse" 5.0
do printfn "%A" b;;
5
5.0
val second: x: 'a -> y: 'b -> 'b
val a: int = 5
val b: float = 5.0
val it: unit = ()
```

does not use the first argument `x`, which therefore can be of any type, and which F#, therefore, calls `'a`. The type of the second element, `y`, can also be of any type and not necessarily the same as `x`, so it is called `'b`. Finally, the result is the same type as `y`, whatever it is. This is an example of a *generic function*, since it will work on any type.

A function may contain a sequence of expressions but must return a value. E.g., the quadratic formula may be written as shown in Listing 1.15. Here, we use a set of nested scopes. In the scope of the function `solution` there are further two functions defined, each with their own scope, which F# identifies by the level of indentation. The amount of space used for indentation does not matter, but all lines in the same scope must use the same amount. The scope ends before the first line with the previous indentation or none. Notice how the last expression is not bound to an identifier but is the result of the function, i.e., in contrast to many other languages, F# does not have an explicit keyword for returning values but requires a final expression, which will be returned to the caller of the function. Note also that since the function `discriminant` is defined in the nested scope of `solution`, and because the scope ends before `let a = 1.0`, the function `discriminant` cannot be called outside `solution`.

**Listing 1.15 identifiersExampleAdvance.fsx:
A function may contain sequences of expressions.**

```
let solution a b c sgn =
  let discriminant a b c =
    b ** 2.0 - 4.0 * a * c
  let d = discriminant a b c
  (-b + sgn * sqrt d) / (2.0 * a)

let a = 1.0
let b = 0.0
let c = -1.0
let xp = solution a b c +1.0
let xn = solution a b c -1.0
do printfn "0 = %A * x ** 2.0 + %A * x + %A" a b c
do printfn "  has solutions %A and %A" xn xp
```

```
$ dotnet fsi identifiersExampleAdvance.fsx
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -1.0 and 1.0
```

Lexical scope and function definitions can be a cause of confusion, as the following example in Listing 1.16 shows. Here, the value-binding for a is redefined after it

**Listing 1.16 lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.**

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
do printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeNFunction.fsx
6.0
```

has been used to define a helper function f. So which value of a is used when we later apply f to an argument? To resolve the confusion, remember that value-binding is lexically defined, i.e., the binding `let f z = a * z` uses the value of a as it is defined by the ordering of the lines in the script, not dynamically by when f was called. Hence, **think of lexical scope as substitution of an identifier with its value or function immediately at the place of definition.** Since a and 3.0 are synonymous in the first lines of the program, the function f is really defined as `let f z = 3.0 * z`. ★

Functions do not need a name, but may be declared as an *anonymous function* using the *fun* keyword and the "->" lexeme, as shown in Listing 1.17. Here, a name is bound to an anonymous function which returns the first of two arguments. The

Listing 1.17 functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

```fsharp
let first = fun x y -> x
do printfn "%d" (first 5 3)
```

```
$ dotnet fsi functionDeclarationAnonymous.fsx
5
```

difference to `let first x y = x` is that anonymous functions may be treated as values, meaning that they may be used as arguments to other functions and the new values may be reassigned to their identifiers when mutable, as will be discussed in **??**. A common use of anonymous functions is as arguments to other functions, as demonstrated in Listing 1.18. Note that here `apply` is given 3 arguments: the

Listing 1.18 functionDeclarationAnonymousAdvanced.fsx:
Anonymous functions are often used as arguments for other functions.

```fsharp
let apply f x y  = f x y
let mul = fun a b -> a * b
do printfn "%d" (apply mul 3 6)
```

```
$ dotnet fsi functionDeclarationAnonymousAdvanced.fsx
18
```

function `mul` and 2 integers. It is not given the result of `mul 3 6`, since that would not match the definition of `apply`. **Anonymous functions and functions as arguments are powerful concepts but tend to make programs harder to read, and their use should be limited.** ★

The result of one function is often used as an argument of another. This is function composition, and an example is shown in Listing 1.19. In the example we combine

Listing 1.19 functionComposition.fsx:
Composing functions using intermediate bindings.

```fsharp
let f x = x + 1
let g x = x * x

let a = f 2
let b = g a
let c = g (f 2)
do printfn "a = %A, b = %A, c = %A" a b c
```

```
$ dotnet fsi functionComposition.fsx
a = 3, b = 9, c = 9
```

two functions `f` and `g` by storing the result of `f 2` in `a` and using that as argument of g. This is the same as `g (f 2)`, and in the later case, the compile creates a temporary value for `f 2`. Such compositions are so common in F# that a special set of operators has been invented, called the *piping* operators: "|>" and "<|". They are used as demonstrated in Listing 1.20. The example shows regular composition,

Listing 1.20 functionPiping.fsx:
Composing functions by piping.

```
1 let f x = x + 1
2 let g x = x * x
3
4 let a = g (f 2)
5 let b = 2 |> f |> g
6 let c = g <| (f <| 2)
7 do printfn "a = %A, b = %A, c = %A" a b c
```

```
1 $ dotnet fsi functionPiping.fsx
2 a = 9, b = 9, c = 9
```

left-to-right, and right-to-left piping. The word piping is a pictorial description of data as if it were flowing through pipes, where functions are connection points of pipes distributing data in a network. The three expressions in Listing 1.20 perform the same calculation. The left-to-right piping in line 5 corresponds to the left-to-right reading direction, i.e., the value `2` is used as argument to `f`, and the result is used as argument to `g`. In contrast, right-to-left piping in line 6 has the order of arithmetic composition as line 4. Unfortunately, since the piping operators are left-associative, without the parenthesis in line 6 `g <| f <| 2`, F# would read the expression as `(g <| f) <| 2`. That would have been an error since `g` takes an integer as an argument, not a function. F# can also define composition on a function level. Further discussion on this is deferred to **??**.

A *procedure* is a generalization of the concept of functions, and in contrast to functions, procedures need not return values. This is demonstrated in Listing 1.21. In F#, this is automatically given the unit type as the return value. Procedural

Listing 1.21 procedure.fsx:
A procedure is a function that has no return value, and in F# returns "()".

```
let printIt a = printfn "This is '%A'" a
do printIt 3
do printIt 3.0
```

```
$ dotnet fsi procedure.fsx
This is '3'
This is '3.0'
```

thinking is useful for *encapsulation* of scripts, but is prone to *side-effects*. For this

★ reason, it is advised to **prefer functions over procedures.** More on side-effects in **??**.

In F#, functions (and procedures) are *first-class citizens*, which means that functions are values: They may be passed as arguments, returned from a function, and bound to a name. For first-class citizens, the name it is bound to does not carry significance to the language, as, e.g., illustrated with the use of anonymous functions. Technically, a function is stored as a *closure*. A closure is a description of the function, its arguments, its expression, and the environment at the time it was created, i.e., the triple $(args, exp, env)$. Consider the listing in Listing 1.22. It defines two functions

Listing 1.22 functionFirstClass.fsx:
The function `ApplyFactor` has a non-trivial closure.

```
1 let mul x y = x * y
2 let factor = 2.0
3 let applyFactor fct x =
4   let a = fct factor x
5   string a
6
7 do printfn "%g" (mul 5.0 3.0)
8 do printfn "%s" (applyFactor mul 3.0)
```

```
1 $ dotnet fsi functionFirstClass.fsx
2 15
3 6
```

`mul` and `applyFactor`, where the latter is a higher-order function taking another function as an argument and using part of the environment to produce its result. The two closures are:

$$\texttt{mul} : (\text{args}, \text{exp}, \text{env}) = \left((\texttt{x}, \texttt{y}), (\texttt{x * y}), ()\right) \tag{1.3}$$

$$\texttt{applyFactor} : (\text{args}, \text{exp}, \text{env}) = ((\texttt{x}, \texttt{fct}), (\texttt{body}), (\texttt{factor} \rightarrow 2.0)) \tag{1.4}$$

where lazily write `body` instead of the whole function's body. The function `mul` does not use its environment, and everything needed to evaluate its expression is values for its arguments. The function `applyFactor` also takes two arguments, a function and a value. It uses `factor` from the environment, thus this is stored in its closure. When `mul` is given as an argument in Listing 1.22 line 8, then it is its closure which is given to `applyFactor`, and the closure contains everything that `applyFactor` requires to use `mul`. Likewise, if `applyFactor` is given as the argument to yet another function, then its closure includes the relevant part of its environment at the time of definition, `factor`, such that when `applyFactor` is applied to two arguments, then its closure contains everything needed to evaluate its expression.

1.3 Do-Bindings

Aside from let-bindings that bind names with values or functions, sometimes we just need to execute code. This is called a *do*-binding or, alternatively, a *statement*. The syntax is as follows:

Listing 1.23: Syntax for do-bindings.

```
[do ]<expr>
```

The expression <expr> must return unit. The keyword do is optional in most cases, but using it emphasizes that the expression is not a function that returns a useful value. Procedures are examples of such expressions, and a very useful family of procedures is the printf family described below. In the remainder of this book, we will refrain from using the do keyword.

1.4 Conditional Expressions

Programs often contain code that should only be executed under certain conditions. The match – with expressions allows us to write such expressions, and its syntax is as follows:

Listing 1.24: Syntax for match-expressions.

```
match <inputExpr> with
 [| ]<pat> [when <guardExpr>] -> <caseExpr>
  | <pat> [when <guardExpr>] -> <caseExpr>
  | <pat> [when <guardExpr>] -> <caseExpr>
 ...
```

where <inputExpr> is the *input pattern* to find matches of, <pat> is a pattern to match with, <guardExpr> is an optional guard expression, and <caseExpr> is the resulting expression. Each set starting with <pat> is called a case. The indentation must be equal to or higher than the indentation of match. All cases must return a value of the same type, and F# reports an error when the complete domain of the input pattern is not covered by cases in match-expressions.

For example, a calendar program may need to highlight weekends, and thus highlighting code should be called, if and only if the day of the week is either Saturday og Sunday, as shown in the following. Remember that match – with is the last expression in the function whatToDoToday, hence the resulting string of match – with is also the resulting string of the function. F# examines each case from top to bottom, and as soon as a matching case is found, then the code following the

Listing 1.25 weekdayPatternSimple.fsx:
Using `match` – `with` to print discriminated unions.

```
let whatToDoToday (d : int) : string =
  match d with
    0 -> "go to work"
    | 1 -> "go to work"
    | 2  -> "go to work"
    | 3 -> "go to work"
    | 4 -> "go to work"
    | 5 -> "take time off"
    | 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternSimple.fsx
Day 3 of the week you should "go to work"
```

"->"-symbol is executed. This code is called the case's *branch*. The "_" pattern is a *wildcard* and matches everything.

In the above code, each day has its own return string associated with it, which is great, if we want to return different messages for different days. However, in this code, we only return two different strings, and if, e.g., choose the one for weekdays, then we need to make 5 identical changes to the code. This is wasteful and risks introducing new errors. So instead, we may compact this code by concatenating the "`|`"'s as shown in Listing 1.26. It still seems unnecessarily clumsy to have to

Listing 1.26 weekdayPatternSum.fsx:
Concatenating cases. Compare with Listing 1.25.

```
let whatToDoToday (d : int) : string =
  match d with
    0 | 1 | 2  | 3 | 4 -> "go to work"
    | 5 | 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternSum.fsx
Day 3 of the week you should "go to work"
```

mention the number of each weekday explicitly and to avoid this, we may use *guards* as illustrated in Listing 1.27. In the guard expression, we match `d` with the name

Listing 1.27 weekdayPatternGuard.fsx:
Using guards. Compare with Listing 1.26.

```
let whatToDoToday (d : int) : string =
  match d with
    n when 0 <= n && n < 5 -> "go to work"
    | n when 5 <= n && n <= 6  -> "take time off"
    | _ -> "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternGuard.fsx
Day 3 of the week you should "go to work"
```

`n` as long as the condition is fulfilled, i.e., as long $0 \leq n < 5$ or as it is written in F#, `0 <= n && n < 5`. The logical and-operator ("&&") is needed in F#, since both `0 <= n` and `n < 5` needs to be true, and F# can only evaluate them individually.

Note that `match` – `with` expressions will give a warning if there are no cases for the full domain of what is being matched. I.e., if we don't end with the wildcard pattern when matching for integers, we would be missing cases for all other patterns

Listing 1.28 weekdayPatternError.fsx:
Using `match` – `with` to print discriminated unions.

```
let whatToDoToday (d : int) : string =
  match d with
    n when 0 <= n && n < 5 -> "go to work"
    | n when 5 <= n && n <= 6  -> "take time off"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayPatternError.fsx


weekdayPatternError.fsx(2,9): warning FS0025: Incomplete
   pattern matches on this expression.

Day 3 of the week you should "go to work"
```

An alternative to `match` – `with` is the `if` – `then` expressions.

Listing 1.29: Conditional expressions.

```
if <cond> then <expr> {elif <cond> then <expr>} [else <expr>]
```

The condition `<con>` is an expression resulting in a Boolean value, and there can be zero or more `elif` conditions, as indicated by `{}`. Each expression `<expr>` is called a *branch*, and all branches must have the same type, such that regardless of which branch is chosen, the type of the result of the conditional expression is the same. Then the expression of the first if-branch, whose condition is true, is evaluated. If all conditions are false then the `else`-branch is evaluated. If no `else` expression is present, then "`()`" will be returned. See Listing 1.30 for a simple example. The branches are often several lines of code, in which case it is more

Listing 1.30 weekdayIfThenElse.fsx:
Conditional computation with `if` – `then`. Compare with Listing 1.27.

```
let whatToDoToday (d : int) : string =
    if 0 <= d && d < 5 then "go to work"
    elif 5 <= d && d <= 6  then "take time off"
    else "unknown day of the week"

let dayOfWeek = 3
let task = whatToDoToday dayOfWeek
printfn "Day %A of the week you should %A" dayOfWeek task
```

```
$ dotnet fsi weekdayIfThenElse.fsx
Day 3 of the week you should "go to work"
```

useful to write them indented, below the keywords, which may also make the code easier to read. The identical `whatToDoToday` function with indented branches is shown in Listing 1.31. In contrast to `match` – `with`, the `if` – `then` expression is

Listing 1.31 weekdayIfThenElseIndentation.fsx:
Conditional computation with `if` – `then`. Compare with Listing 1.27.

```
let whatToDoToday (d : int) : string =
    if 0 <= d && d < 5 then
        "go to work"
    elif 5 <= d && d <= 6  then
        "take time off"
    else
        "unknown day of the week"
```

not as thorough in investigating whether cases of the domain in question have been covered. For example, in both

```
let a = if true then 3
```

and

```
let a = if true then 3 elif false then 4
```

the value of a is uniquely determined, but F# finds them to be erroneous since F# looks for the `else` to ensure all cases have been covered, and that the branches have the identical type. Hence,

```
let a = if true then 3 else 4
```

is the only valid expression of the 3. In practice, F# assumes that the omitted branch returns "`()`", and thus it is fine to say `let a = if true then ()` and `if true then printfn "hej"`. Nevertheless, it is good practice in F# to always include an `else` branch.

1.5 Tracing code by hand

The concept of Tracing by hand will be developed throughout this book. Here we will concentrate on the basics, and as we introduce more complicated programming structures, we will develop the Tracing by hand accordingly. Tracing may seem tedious in the beginning but in conjunction with strategically placed debugging `printfn` statements, it is a very valuable tool for debugging.

Consider the program in Listing 1.32. The program calls `testScope 2.0`, and by

Listing 1.32 lexicalScopeTracing.fsx:
Example of lexical scope and closure environment.

```
let testScope x =
  let a = 3.0
  let f z = a * z
  let a = 4.0
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeTracing.fsx
6.0
```

running the program, we see that the return-value is `6.0` and not `8.0`, as we had expected. Hence, we will use tracing to understand the result.

Tracing a program by hand means that we simulate its execution and, as part of that, keep track of the bindings, assignments closures, scopes, and input and output of the program. To do this, we need to consider the concept of *environments*.

Environments describe bindings available to the program at the present scope and at a particular time and place in the code. There is always an outer environment, called E_0, and each time we call a function or create a scope, we create a new environment. Only one environment can be active at a time, and it is updated as we simulate the execution of code with new bindings and temporary evaluations of expressions. Once a scope is closed, then its environment is deleted and a return value is transported to its enclosing environment. In tracing, we note return values explicitly. Likewise, output from, e.g., `printfn` is reported with a special notation.

To trace code, we make a table with 4 columns: Step, Line, Environment, and Bindings and evaluations. The Step column enumerates the steps performed. The Line column contains the program-line treated in a step *where* the present environment is updated. The Environment contains the name of the present environment, and Bindings . . . shows *what* in the environment is updated.

The code in Listing 1.32 contains a function definition and a call, hence, the first lines of our table look like this,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x)$, testScope-body, $()\big)$
2	6	E_0	testScope 2.0 = ?

The elements of the table are to be understood as follows. Step 0 initializes the outer environment. In order for us to remember that the environment is empty, we write the unit value "`()`". Reading the code from top to bottom, the first nonempty and non-comment line we meet is line 1, hence, in Step 1, we update the environment with the binding of a function to the name `testScope`. Since functions are values in F#, we note their bindings by their closures: a list of argument names, the function-body, and the values lexically available at the place of binding. See Section 1.2 for more information on closures. Following the function-binding, the `printfn` statement is called in line 6 to print the result `testScope 2.0`. However, before we can produce any output, we must first evaluate `testScope 2.0`. Since we do not yet know what this function evaluates to, in Step 2 we simply write the call with a question mark. The call causes the creation of a new environment, and we continue our table as follows,

Step	Line	Env.	Bindings and evaluations
3	1	E_1	$\big((x = 2.0)$, testScope-body, $()\big)$

This means that we are going to execute the code in testScope-body. The function was called with 2.0 as an argument, causing $x = 2.0$. Hence, the only binding available at the start of this environment is to the name `x`. In the testScope-body, we make 3 further bindings and a function call. First to `a`, then to `f`, then to another `a`, which will overshadow the previous binding, and finally we call `f`. Thus, our table is updated as follows,

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = 3.0$
5	3	E_1	$\mathrm{f} = \big((z), \mathrm{a} * \mathrm{z}, (a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	$\mathrm{f}\, x = ?$

Note that by lexical scope, the closure of `f` includes everything above its binding in E_1, and therefore we add $a = 3.0$ and $x = 2.0$ to the environment element in its closure. This has consequences for the following call to `f` in line 5, which creates a new environment based on `f`'s closure and the value of its arguments. The value of `x` in Step 7 is found by looking in the previous steps for the last binding to the name `x` in E_1, which occurs in Step 3. Note that the binding to a name `x` in Step 5 is an internal binding in the closure of `f` and is irrelevant here. Hence, we continue the table as,

Step	Line	Env.	Bindings and evaluations
8	3	E_2	$\big((z = 2.0), \mathrm{a} * \mathrm{z}, (a = 3.0, x = 2.0)\big)$

Executing the body of `f`, we initially have 3 bindings available: `z = 2.0`, `a = 3.0`, and `x = 2.0`. Thus, to evaluate the expression `a * z`, we use these bindings and write,

Step	Line	Env.	Bindings and evaluations
9	3	E_2	$a * z = 6.0$
10	3	E_2	return = 6.0

The 'return'-word is used to remind us that this is the value to replace the question mark within Step 7. Here we will make a mental note and not physically replace the question mark with the calculated value. If you are ever in doubt which call is connected with which return value, seek upwards in the table from the return statement for the first question mark. Now we delete E_2 and return to the enclosing environment, E_1. Here the function call was the last expression, hence the return-value from `testScope` will be equal to the return-value from `f`, and we write,

Step	Line	Env.	Bindings and evaluations
11	3	E_1	return = 6.0

Similarly, we delete E_1 and return to the question mark in Step 2, which is replaced by the value 6.0. We can now finish the `printfn` statement and produce the output,

Step	Line	Env.	Bindings and evaluations
12	6	E_0	output = "6.0\n"

The return-value of a `printfn` statement is `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
13	6	E_0	return = ()

The full table is shown for completeness in Table 1.2. Hence, we conclude that the

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x)$, testScope-body, $()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0)$, testScope-body, $()\big)$
4	2	E_1	$a = 3.0$
5	3	E_1	f = $\big((z)$, a * z, $(a = 3.0, x = 2.0)\big)$
6	4	E_1	$a = 4.0$
7	5	E_1	f x = ?
8	3	E_2	$\big((z = 2.0)$, a * z, $(a = 3.0, x = 2.0)\big)$
9	3	E_2	$a * z = 6.0$
10	3	E_2	return = 6.0
11	3	E_1	return = 6.0
12	6	E_0	output = “6.0\n”
13	6	E_0	return = ()

Table 1.2 The complete table produced while tracing the program in Listing 1.32 by hand.

program outputs the value `6.0`, since the function `f` uses the first binding of `a = 3.0`, and this is because the binding of `f` to the expression `a * z` creates a closure with a lexical scope. Thus, in spite that there is an overshadowing value of `a`, when `f` is called, this binding is ignored in the body of `f`. To correct this, we update the code as shown in Listing 1.33.

Listing 1.33 lexicalScopeTracingCorrected.fsx:
Tracing the code in Listing 1.32 by hand produced the table in Table 1.2, and to get the desired output, we correct the code as shown here.

```
let testScope x =
  let a = 4.0
  let f z = a * z
  f x
printfn "%A" (testScope 2.0)
```

```
$ dotnet fsi lexicalScopeTracingCorrected.fsx
8.0
```

1.6 Key Concepts and Terms in This Chapter

In this chapter, we have taken the first look at organizing code. Key concepts have been:

- **Binding** values and functions using the **let** statements.
- Function **closures**, which are the values bound in **let** statements of functions.
- Certain names are forbidden in particular **keywords**, which are reserved for other uses.
- In contrast to **let** expressions, **do** statements do something, such as `printfn` prints to the screen. They return the "`()`" value.
- Conditional expressions using **match–with** and some **patterns** and **guards**. Alternatively, **if–then** expressions can be used to a similar effect.
- **Trace by hand** as a method for simulating execution and in particular keeping track of the **environments** defined by the code structure.