1 Values and Functions

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

For 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 $\pm$ 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

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
```

```
$ fsharpc --nologo identifiersExample.fsx && mono
  identifiersExample.exe
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has discriminant 4.0 and solutions -1.0 and 1.0
```

avoiding possible typing mistakes and reducing the amount of code which 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

- 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 of 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 represents 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`.

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#.

Since programmers often work in multilingual environment dominated by the English language it is advicable to **restrict identifiers to use letters from the English alphabet, numbers, period, 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 letter 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 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 been chosen to be `discriminant`. F# places no special significance to 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 confuse 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 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 character, 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 $f3$, 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 result 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, but the syntaxes may be mixed, and we will highlight the options, when relevant.

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.

```
1 let <valueIdent> = <bodyExpr> [in <expr>]
```

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 following square bracket notation `[]` means that the enclosed is optional, and F# is able to identify whether or not the optional part is used as signified by the optional presence of the *in* keyword. If the `in` keyword is used, then the value-identifier is a local definition in the `<expr>` expression, and it is not available in later lines. For lightweight syntax, the `in` keyword is replaced with a newline, and the binding *is* available in later lines until the end of the scope it is defined in.

The value identifier 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. See **??** for more details on patterns.

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. F# will ignore most newlines between lexemes, i.e., the above is equivalent to writing as shown in Listing 1.4. F# also allows for an alternative notation called *lightweight syntax*, where e.g., the `in` keyword is replaced with a newline, and the expression starts on the next line at the same column as `let` starts in, i.e., the above is equivalent to Listing 1.5. The same expression in interactive mode will also show with the inferred types, as shown in Listing 1.6. By 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

Listing 1.3 letValue.fsx:
The identifier p is used in the expression following the in keyword.

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

```
$ fsharpc --nologo letValue.fsx && mono letValue.exe
9.0
```

Listing 1.4 letValueLF.fsx:
Newlines after in make the program easier to read.

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

```
$ fsharpc --nologo letValueLF.fsx && mono letValueLF.exe
9.0
```

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 1.7. 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.

An expression can be a sequence of expressions separated by the lexeme ";", see Listing 1.8. The lightweight syntax automatically inserts the ";" lexeme at newlines, hence using the lightweight syntax, the above is the same as shown in Listing 1.9.

A key concept of programming is *scope*. When F# seeks the value bound to a name,

Listing 1.5 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)
```

```
$ fsharpc --nologo letValueLightWeight.fsx
$ mono letValueLightWeight.exe
9.0
```

Listing 1.6: 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 = ()
```

Listing 1.7 letValueTypeError.fsx: Binding error due to type mismatch.

```
let p : float = 3
do printfn "%A" (3.0 ** p)
```

```
$ fsharpc --nologo letValueTypeError.fsx && mono
  letValueTypeError.exe

letValueTypeError.fsx(1,17): error FS0001: This expression
  was expected to have type
    'float'
but here has type
    'int'
```

it looks left and upward in the program text for its `let`-binding in the present or higher scopes, see Listing 1.10 for an example. This is called *lexical scope*. Some special bindings are mutable, in which case F# uses the *dynamic scope*, that is, the value of a binding is defined by when it is used. This will be discussed in Section 1.7.

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 1.11. However, using parentheses, we create a *code block*, i.e., a *nested scope*, and then redefining is allowed, as demonstrated in Listing 1.12. Nevertheless, **avoid reusing names unless it's in a deeper scope.** ⋆

Inside the block in Listing 1.12 we used indentation, which is good practice, but not required here.

Bindings inside a nested scope are not available outside, as shown in Listing 1.13. Nesting is a natural part of structuring code, e.g., through function 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,

Listing 1.8 letValueSequence.fsx:
A value-binding for a sequence of expressions.

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

```
$ fsharpc --nologo letValueSequence.fsx && mono
  letValueSequence.exe
2.0
9.0
```

Listing 1.9 letValueSequenceLightWeight.fsx:
A value-binding for a sequence using lightweight syntax.

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

```
$ fsharpc --nologo letValueSequenceLightWeight.fsx
$ mono letValueSequenceLightWeight.exe
2.0
9.0
```

without the user necessarily needing to know the details of the inner workings of a block of code.

Defining blocks is used for controlling the extent of a lexical scope of bindings. For example, adding a second `printfn` statement, as in Listing 1.14, will print the value 4, last bound to the identifier `p`, since F# interprets the above as `let p = 3 in let p = 4 in (printfn "%A" p; printfn "%A" p)`. Had we intended to print the two different values of `p`, then we should have created a block as in Listing 1.15.

Listing 1.10 letValueScopeLower.fsx:
Redefining identifiers is allowed in lower scopes.

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

```
$ fsharpc --nologo letValueScopeLower.fsx && mono
   letValueScopeLower.exe
 4
```

Listing 1.11 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'
```

Listing 1.12 letValueScopeBlockAlternative3.fsx:
A block may be created using parentheses.

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

```
$ fsharpc --nologo letValueScopeBlockAlternative3.fsx
$ mono letValueScopeBlockAlternative3.exe
4
```

Listing 1.13 letValueScopeNestedScope.fsx: Bindings inside a scope are not available outside.

```
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. Maybe you want
   one of the following:
    p
```

Listing 1.14 letValueScopeBlockProblem.fsx: Overshadowing hides the first binding.

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

```
$ fsharpc --nologo letValueScopeBlockProblem.fsx
$ mono letValueScopeBlockProblem.exe
4
4
```

Listing 1.15 letValueScopeBlock.fsx: Blocks allow for the return to the previous scope.

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

```
$ fsharpc --nologo letValueScopeBlock.fsx && mono
   letValueScopeBlock.exe
4
3
```

1.2 Function Bindings

A function is a mapping between an input and output domain. A key advantage of using functions when programming is that they encapsulate code into smaller units, that are easier to debug and may be reused. 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 1.16: Function binding expression

```
let <funcIdent> <arg> {<arg>} | () = <bodyExpr> [in <expr>]
```

Here `<funcIdent>` 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 `<bodyExpr>`. 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 1.17: Function binding expression

```
let <funcIdent> (<arg> : <type>) {(<arg> : <type>)} : <type>
  | () : <type> = <bodyExpr> [in <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 **??**.

An example of defining a function and using it in interactive mode is shown in Listing 1.18. Here we see that the function is 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 argument 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

Listing 1.18: An example of a binding of an identifier and a function.

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

just have declare the type of the result, as in Listing 1.19. Or even just one of the

Listing 1.19 letFunctionAlterantive.fsx: Not every type needs to be declared.

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

arguments, as in Listing 1.20. In both cases, since the `+` *operator* is only defined for

Listing 1.20 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. As for values, lightweight syntax automatically inserts the keyword `in` and the lexeme "`;`", as shown in Listing 1.21.

Arguments need not always be inferred to types, but may be of the generic type when *type safety* is ensured, as shown in Listing 1.22. Here, the function `second` 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 1.23. Here, we used the lightweight syntax, where the "`=`" identifies the start of a nested scope, and F# identifies the scope by indentation. The amount of space used for indentation does not matter, but all lines in the same scope must use the same amount. The

Listing 1.21 letFunctionLightWeight.fsx: Lightweight syntax for function definitions.

```
let sum x y : float = x + y
let c = sum 357.6 863.4
do printfn "%A" c
```

```
$ fsharpc --nologo letFunctionLightWeight.fsx
$ mono letFunctionLightWeight.exe
1221.0
```

Listing 1.22: 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 = ()
```

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 becausethe scope ends before `let a = 1.0`, `discriminant` cannot be called outside `solution`.

Lexical scope and function definitions can be a cause of confusion, as the following example in Listing 1.24 shows. Here, the value-binding for `a` is redefined after it 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` ⋆

Listing 1.23 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
```

```
$ fsharpc --nologo identifiersExampleAdvance.fsx
$ mono identifiersExampleAdvance.exe
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -1.0 and 1.0
```

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 1.25. Here, a name is bound to an anonymous function which returns the first of two arguments. The 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 Section 1.7. A common use of anonymous functions is as arguments to other functions, as demonstrated in Listing 1.26. Note that here `apply` is given 3 arguments: the 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 1.27. In the example we combine 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

Listing 1.24 lexicalScopeNFunction.fsx:
Lexical scope means that $f(z) = 3x$ and not $4x$ at the time of calling.

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

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

Listing 1.25 functionDeclarationAnonymous.fsx:
Anonymous functions are functions as values.

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

```
1  $ fsharpc --nologo functionDeclarationAnonymous.fsx
2  $ mono functionDeclarationAnonymous.exe
3  5
```

of operators has been invented, called the *piping* operators: "`|>`" and "`<|`". They are used as demonstrated in Listing 1.28. The example shows regular composition, left-to-right, and right-to-left piping. The word piping is a picturial 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 1.28 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-associatitve, 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 argument, not a function. F# can also define composition on a function level. Further discussion on this is deferred to **??**. The piping operator comes in four variants: "`||>`", "`<||`", "`|||>`", and "`<|||`". These allow for piping between pairs and triples to functions of 2 and 3 arguments, see Listing 1.29 for an example. The example demonstrates right-to-left piping, left-to-right works analogously.

Listing 1.26 functionDeclarationAnonymousAdvanced.fsx: Anonymous functions are often used as arguments for other functions.

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

```
$ fsharpc --nologo functionDeclarationAnonymousAdvanced.fsx
$ mono functionDeclarationAnonymousAdvanced.exe
18
```

Listing 1.27 functionComposition.fsx: Composing functions using intermediate bindings.

```
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
```

```
$ fsharpc --nologo functionComposition.fsx
$ mono functionComposition.exe
a = 3, b = 9, c = 9
```

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 1.30. In F#, this is automatically given the unit type as the return value. Procedural thinking is useful for *encapsulation* of scripts, but is prone to *side-effects*. For this reason, it is ⋆ adviced to **prefer functions over procedures.** More on side-effects in Section 1.7.

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 1.31. It defines two functions `mul` and `applyFactor`, where the latter is a higher-order function taking another function as an argument and uses part of the environment to produce its result. The

Listing 1.28 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 $ fsharpc --nologo functionPiping.fsx && mono
    functionPiping.exe
2 a = 9, b = 9, c = 9
```

two closures are:

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

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

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 are 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 1.31 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 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.

Listing 1.29 functionTuplePiping.fsx:
Tuples can be piped to functions of more than one argument.

```
let f x = printfn "%A" x
let g x y = printfn "%A %A" x y
let h x y z = printfn "%A %A %A" x y z

1 |> f
(1, 2) ||> g
(1, 2, 3) |||> h
```

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

Listing 1.30 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
```

```
$ fsharpc --nologo procedure.fsx && mono procedure.exe
This is '3'
This is '3.0'
```

Listing 1.31 functionFirstClass.fsx:
The function `ApplyFactor` has a non-trivial closure.

```
let mul x y = x * y
let factor = 2.0
let applyFactor fct x =
  let a = fct factor x
  string a

do printfn "%g" (mul 5.0 3.0)
do printfn "%s" (applyFactor mul 3.0)
```

```
$ fsharpc --nologo functionFirstClass.fsx && mono
  functionFirstClass.exe
15
6
```

1.3 Operators

Operators are functions, and in F#, the infix multiplication operator `*` is equivalent to the function `(*)`, as shown in Listing 1.32. All operators have this option, and you

Listing 1.32 addOperatorNFunction.fsx: Operators have function equivalents.

```
let a = 3.0
let b = 4.0
let c = a * b
let d = (*) a b
do printfn "%A plus %A is %A and %A" a b c d
```

```
$ fsharpc --nologo addOperatorNFunction.fsx
$ mono addOperatorNFunction.exe
3.0 plus 4.0 is 7.0 and 7.0
```

may redefine them and define your own operators, but in F# names of user-defined operators are limited:

- A *unary operator* name can be: "`+`", "`-`", "`+.`", "`-.`", "`%`", "`&`", "`&&`", "`~~`", "`~~~`", "`~~~~`", ..., apostropheOp. Here apostropheOp is an operator name starting with "`!`" and followed by one or more of either "`!`", "`%`", "`&`", "`*`", "`+`", "`-`", "`.`", "`/`", "`<`", "`=`", "`>`", "`@`", "`^`", "`|`", "`~`", but apostropheOp cannot be "`!=`".

- An *binary operator* name can be: "`+`", "`-`", "`+.`", "`-.`", "`%`", "`&`", "`&&`", "`:=`", "`::`", "`$`", "`?`", dotOp. Here dotOp is an operator name starting with "`.`" and followed by "`+`", "`-`", "`+.`", "`-.`", "`%`", "`&`", "`&&`", "`-`", "`+`", "`||`", "`<`", "`>`", "`=`", "`|`", "`&`", "`^`", "`*`", "`/`", "`%`", "`!=`". Only "`?`" and "`?<-`" may start with "`?`".

The precedence rules and associativity of user-defined operators follow the rules for which they share prefixes with built-in rules, see **??**. For example, `.*`, `+++`, and `<+` are valid operator names for infix operators, they have precedence as ordered, and their associativities are all left. Using `~` as the first character in the definition of an operator makes the operator unary and will not be part of the name. Examples of definitions and use of operators are, Operators beginning with `*` must use a space in its definition. For example, without a space `( *` would be confused with the beginning of a comment `(*`, see **??** for more on comments in the code.

Beware, redefining existing operators lexically redefines all future uses of the operators
⋆ for all types, hence **it is not a good idea to redefine operators, but better to define new ones.** In **??** we will discuss how to define type-specific operators,

Listing 1.33 operatorDefinitions.fsx:
Operators may be (re)defined by their function equivalent.

```
let (.*) x y = x * y + 1
printfn "%A" (3 .* 4)
let (+++) x y = x * y + y
printfn "%A" (3 +++ 4)
let (<+) x y = x < y + 2.0
printfn "%A" (3.0 <+ 4.0)
let (~+.) x = x+1
printfn "%A" (+.1)
```

```
$ fsharpc --nologo operatorDefinitions.fsx
$ mono operatorDefinitions.exe
13
16
true
2
```

including prefix operators.

1.4 Do-Bindings

Aside from `let`-bindings that binds 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 1.34: 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 are the `printf` family described below. In the remainder of this book, we will refrain from using the `do` keyword.

1.5 The Printf Function

A common way to output information to the console is to use one of the family of *printf* commands. These functions are special, since they take a variable number of arguments, and the number is decided by the first argument - the format string. The syntax for the `printf` commands are as follows:

Listing 1.35: `printf` statement.

```
printf <format-string> {<ident>}
```

The `formatString` is a string (simple or verbatim) with placeholders. The function `printf` prints `formatString` to the console, where all `placeholder` have been replaced by the values of the corresponding arguments formatted as specified. For example, in `printfn "1 2 %d" 3`, the `formatString` is `"1 2 %d"` and the placeholder is `%d`. When executed, `printf` will replace `%d` with the following argument, `3`, and print the result to the console: `1 2 3`. There are specifiers for all the basic types, and more, as elaborated in Table 1.2.

Specifier	Type	Comment
`%b`	`bool`	formatted as “true” or “false”
`%s`	`string`	
`%c`	`char`	
`%d`, `%i`	basic integer	
`%u`	basic unsigned integers	
`%x`	basic integer	formatted as unsigned hexadecimal with lower case letters
`%X`	basic integer	formatted as unsigned hexadecimal with upper case letters
`%o`	basic integer	formatted as unsigned octal integer
`%f`, `%F`,	basic floats	formatted on decimal form
`%e`, `%E`,	basic floats	formatted on scientific form. Lower case uses "e" while upper case uses "E” in the formatting.
`%g`, `%G`,	basic floats	formatted on the shortest of the corresponding decimal or scientific form.
`%A`, `%O`	any value	formatted as a basic type or using the object’s `ToString` method

Table 1.2: Some printf placeholder strings.

The placeholder can be parametrized, e.g., the placeholder string `%8s` will print a right-aligned string which that is eight characters wide and padded with spaces, as needed. For floating point numbers, `%8f` will print a number that is exactly seven digits and a decimal point, making eight characters in total. Zeros are added after the decimal point, as needed. Alternatively, we may specify the number of digits after the decimal point, such that `%8.1f` will print a floating point number, aligned to the right, with one digit after the decimal point padded with spaces, as needed. The default is for the value to be right justified in the field, but left justification can be specified by the `-` character. For number types, you can specify their format by `"0"` for padding the number with zeros to the left when right justifying the number; `"+"` to explicitly show a plus sign for positive numbers; `SP` to enforce a space, where there otherwise would be a plus sign for positive numbers. The placeholder parameter may also be given as an argument to `printf` which case the placeholder should use the `*` character instead of an integer.

Examples of placeholder parametrization are shown in Listing 1.36. Not all combinations of flags and identifier types are supported, e.g., strings cannot have the number of integers after the decimal point specified. The placeholder types `"%A"`, `"%a"`, and `"%t"` are special for F#, examples of their use are shown in Listing 1.37. The `%A` is special in that all built-in types, including tuples, lists, and arrays to be discussed in **??**, can be printed using this formatting string, but notice that the formatting performed includes the named literal string. The two formatting strings `%t` and `%a` are options for user-customizing the formatting and will not be discussed further.

Beware, `formatString` is not a `string` but a `Printf.TextWriterFormat`, so to predefine a `formatString` as, e.g., `let str = "hello %s" in printf str "world"`, will be a type error.

The family of `printf` is shown in Table 1.3. The function `fprintf` prints to a stream, e.g., `stderr` and `stdout`, of type `System.IO.TextWriter`. For the moment it is sufficient to think of both `stderr` and `stdout` to be the console. Streams will be discussed in further detail in **??**. The function `failwithf` is used with exceptions, see **??** for more details. The function has a number of possible return value types, and for testing, the *ignore* function ignores it all, e.g., `ignore (failwithf "%d failed apples" 3)`.

1.6 Reading from the Console

The `printf` and `printfn` functions allow us to write text on the screen. A program often needs to ask a user to input data, e.g., by typing text on a keyboard. Text typed on the keyboard is accessible through the `stdin` stream, and F# provides

Function	Example	Description
`printf`	`printf "%d apples" 3`	Prints to the console, i.e., `stdout`
`printfn`		As `printf` and adds a newline.
`fprintf`	`fprintf stream "%d apples" 3`	Prints to a stream, e.g., `stderr` and `stdout`, which would be the same as `printf` and `eprintf`.
`fprintfn`		As `fprintf` but with added newline.
`eprintf`	`eprintf "%d apples" 3`	Prints to `stderr`
`eprintfn`		As `eprintf` but with added newline.
`sprintf`	`sprintf "%d apples" 3`	Return printed string
`failwithf`	`failwithf "%d failed apples" 3`	Prints to a string and used for raising an exception.

Table 1.3: The family of printf functions.

several library functions for capturing text typed on the keyboard. In the following section, we will briefly discuss the `System.Console.ReadLine` function. For more details and other methods of input see **????**.

The function `System.Console.ReadLine` takes a unit value as an argument and returns the string the user typed. The program will not advance until the user presses newline. An example of a program that multiplies two floating point numbers supplied by a user is given in Listing 1.38, and an example dialogue is shown in Listing 1.39. Note that the string is immediately cast to floats such that we can multiply the input using the float multiplication operator. This also implies that if the user inputs a non-number, then `mono` will halt with an error message.

Listing 1.36 printfExample.fsx:
Examples of printf and some of its formatting options.

```
let pi = 3.1415192
let hello = "hello"
printf "An integer: %d\n" (int pi)
printf "A float %f on decimal form and on %e scientific
  form\n" pi pi
printf "A char '%c' and a string \"%s\"\n" hello.[0] hello
printf "Float using width 8 and 1 number after the
  decimal:\n"
printf "  \"%8.1f\" \"%8.1f\"\n" pi -pi
printf "  \"%08.1f\" \"%08.1f\"\n" pi -pi
printf "  \"% 8.1f\" \"% 8.1f\"\n" pi -pi
printf "  \"%-8.1f\" \"%-8.1f\"\n" pi -pi
printf "  \"%+8.1f\" \"%+8.1f\"\n" pi -pi
printf "  \"%8s\"\n\"%-8s\"\n" "hello" "hello"
```

```
$ fsharpc --nologo printfExample.fsx && mono
  printfExample.exe
An integer: 3
A float 3.141519 on decimal form and on 3.141519e+000
  scientific form
A char 'h' and a string "hello"
Float using width 8 and 1 number after the decimal:
  "     3.1" "    -3.1"
  "000003.1" "-00003.1"
  "     3.1" "    -3.1"
  "3.1     " "-3.1    "
  "    +3.1" "    -3.1"
  "   hello"
"hello   "
```

Listing 1.37 printfExampleAdvance.fsx: Custom format functions may be used to specialise output.

```
let noArgument writer = printf "I will not print anything"
let customFormatter writer arg = printf "Custom formatter
  got: \"%A\"" arg
printf "Print examples: %A, %A, %A\n" 3.0m 3uy "a string"
printf "Print function with no arguments: %t\n" noArgument
printf "Print function with 1 argument: %a\n"
  customFormatter 3.0
```

```
$ fsharpc --nologo printfExampleAdvance.fsx
$ mono printfExampleAdvance.exe
Print examples: 3.0M, 3uy, "a string"
Print function with no arguments: I will not print anything
Print function with 1 argument: Custom formatter got: "3.0"
```

Listing 1.38 userDialoguePrintf.fsx: Interacting with a user using `ReadLine`.

```
printfn "To perform the multiplication of a and b"
printf "Enter a: "
let a = float (System.Console.ReadLine ())
printf "Enter b: "
let b = float (System.Console.ReadLine ())
printfn "a * b = %A" (a * b)
```

Listing 1.39: An example dialogue of running Listing 1.38. The user typed "3.5" and "7.4".

```
$ fsharpc --nologo userDialoguePrintf.fsx && mono
  userDialoguePrintf.exe
To perform the multiplication of a and b
Enter a: 3.5
Enter b: 7.4
a * b = 25.9
```

1.7 Variables

Identifiers may be mutable, which means that the it may be rebound to a new value. Mutable identifiers are specified using the *mutable* keyword with the following syntax:

Listing 1.40: Syntax for defining mutable values with an initial value.

```
let mutable <ident> = <expr> [in <expr>]
```

Changing the value of an identifier is called *assignment* and is done using the "<-" lexeme. Assignments have the following syntax:

Listing 1.41: Value reassignment for mutable variables.

```
<ident> <- <ident>
```

Mutable values is synonymous with the term *variable*. A variable is an area in the computer's working memory associated with an identifier and a type, and this area may be read from and written to during program execution, see Listing 1.42 for an example. Here, an area in memory was denoted `x`, initially assigned the integer

Listing 1.42 mutableAssignReassingShort.fsx: A variable is defined and later reassigned a new value.

```
let mutable x = 5
printfn "%d" x
x <- -3
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignReassingShort.fsx
$ mono mutableAssignReassingShort.exe
5
-3
```

value 5, hence the type was inferred to be `int`. Later, this value of `x` was replaced with another integer using the "<-" lexeme. The "<-" lexeme is used to distinguish the assignment from the comparison operator. For example, the statement `a = 3` in Listing 1.43 is not an assignment but a comparison which is evaluated to be false. However, it is important to note that when the variable is initially defined, then the "=" operator must be used, while later reassignments must use the "<-" expression.

Listing 1.43: It is a common error to mistake "=" and "<-" lexemes for mutable variables.

```
> let mutable a = 0
- a = 3;;
val mutable a : int = 0
val it : bool = false
```

Assignment type mismatches will result in an error, as demonstrated in Listing 1.44. I.e., once the type of an identifier has been declared or inferred, it cannot be changed.

Listing 1.44 mutableAssignReassingTypeError.fsx: Assignment type mismatching causes a compile-time error.

```
let mutable x = 5
printfn "%d" x
x <- -3.0
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignReassingTypeError.fsx

mutableAssignReassingTypeError.fsx(3,6): error FS0001:
   This expression was expected to have type
     'int'
but here has type
     'float'
$ mono mutableAssignReassingTypeError.exe
Cannot open assembly 'mutableAssignReassingTypeError.exe':
   No such file or directory.
```

A typical variable is a counter of type integer, and a typical use of counters is to increment them, see Listing 1.45 for an example. Using variables in expressions, as opposed to the left-hand-side of an assignment operation, reads the value of the variable. Thus, when using a variable as the return value of a function, then the value is copied from the local scope of the function to the scope from which it is called. This is demonstrated in Listing 1.46. In the example we see that the type is a value, and not mutable.

Variables implement dynamic scope, that is, the value of an identifier depends on *when* it is used. This is in contrast to lexical scope, where the value of an identifier depends on *where* it is defined. As an example, consider the script in Listing 1.24 which defines a function using lexical scope and returns the number `6.0`, however, if

Listing 1.45 mutableAssignIncrement.fsx: Variable increment is a common use of variables.

```
let mutable x = 5 // Declare a variable x and assign the
   value 5 to it
printfn "%d" x
x <- x + 1 // Increment the value of x
printfn "%d" x
```

```
$ fsharpc --nologo mutableAssignIncrement.fsx
$ mono mutableAssignIncrement.exe
5
6
```

Listing 1.46: Returning a mutable variable returns its value.

```
> let g () =
-   let mutable y = 0
-   y
- printfn "%d" (g ());;
0
val g : unit -> int
val it : unit = ()
```

`a` is made `mutable`, then the behavior is different, as shown in Listing 1.47. Here, the response is `8.0`, since the value of `a` changed before the function `f` was called.

1.8 Reference Cells

F# has a variation of mutable variables called *reference cells*. Reference cells have the built-in function ***`ref`*** and the operators "`!`" and "*`:=`*", where `ref` creates a reference variable, and the "`!`" and the "`:=`" operators respectively reads and writes its value. An example of using reference cells is given in Listing 1.48. Reference cells are different from mutable variables, since their content is allocated on *The Heap*. The Heap is a global data storage that is not destroyed when a function returns, which is in contrast to the *call stack*, also known as *The Stack*. The Stack maintains all the local data for a specific instance of a function call, see **??** for more details. As a consequence, when a reference cell is returned from a function, then it is the reference to the location on The Heap, which is returned as a value. Since this points outside the local data area of the function, this location is still valid after the function

Listing 1.47 dynamicScopeNFunction.fsx:
Mutual variables implement dynamic scope rules. Compare with Listing 1.24.

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

```
$ fsharpc --nologo dynamicScopeNFunction.fsx
$ mono dynamicScopeNFunction.exe
8.0
```

Listing 1.48 refCell.fsx:
Reference cells are variants of mutable variables.

```
let x = ref 0
printfn "%d" !x
x := !x + 1
printfn "%d" !x
```

```
$ fsharpc --nologo refCell.fsx && mono refCell.exe
0
1
```

returns, and the variable stored there is accessible to the caller. This is illustrated in Figure 1.1

Reference cells may cause *side-effects*, where variable changes are performed across independent scopes. Some side-effects are useful, e.g., the `printf` family changes the content of the screen, and the screen is outside the scope of the caller. Another example of a useful side-effect is a counter shown in Listing 1.49. Here `incr` is an anonymous function with an internal state `counter`. At first glance, it may be surprising that `incr ()` does not return the value `1` at every call. The reason is that the value of the `incr` is the closure of the anonymous function `fun () -> counter := ...`, which is

$$\texttt{incr} : (\text{args}, \text{exp}, \text{env}) = \Big((), \big(\texttt{counter := !counter + 1}, \texttt{!counter}\big), (\texttt{counter} \to \texttt{ref 0})\Big). \tag{1.5}$$

Thus, `counter` is only initiated once at the initial binding, while every call of `incr ()`

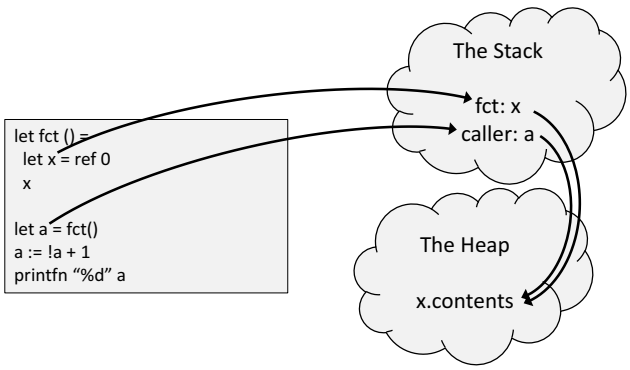

Figure 1.1: A reference cell is a pointer to The Heap, and the content is not destroyed when its reference falls out of scope.

updates its value on The Heap. Such a programming structure is called *encapsulation*, since the `counter` state has been encapsulated in the anonymous function, and the only way to access it is by calling the same anonymous function. In general, it is advisable to **use encapsulation to hide implementation details irrelevant to the user of the code.** ⋆

The `incr` example in Listing 1.49 is an example of a useful side-effect. An example to be avoided is shown in Listing 1.50. In the example, the function `updateFactor` changes a variable in the scope of the function `multiplyWithFactor`. The code style is prone to errors, since the computations are not local at the place of writing, i.e., in `multiplyWithFactor`, and if `updateFactor` were defined in a library, then the source code may not be available. Better style of programming is shown in Listing 1.51. Here, there can be no doubt in `multiplyWithFactor` that the value of `a` is changing. Side-effects do have their use, but should, in general, be avoided at almost all costs, and it is advised to **minimize the use of side effects**. ⋆

Reference cells give rise to an effect called *aliasing*, where two or more identifiers refer to the same data, as illustrated in Listing 1.52. Here, `a` is defined as a reference cell, and by defining `b` to be equal to `a`, we have created an alias. This can be very confusing since as the example shows, changing the value of `b` causes `a` to change as well. Aliasing is a variant of side-effects, and **aliasing should be avoided at all costs**. ⋆

Since F# version 4.0, the compiler has automatically converted mutable variables to reference cells, where needed. E.g., Listing 1.49 can be rewritten using a mutable variable, as shown in Listing 1.53. Reference cells are preferred over mutable variables for encapsulation, in order to avoid confusion.

Listing 1.49 refEncapsulation.fsx:
An increment function with a local state using a reference cell.

```
let incr =
  let counter = ref 0
  fun () ->
    counter := !counter + 1
    !counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

```
$ fsharpc --nologo refEncapsulation.fsx && mono
  refEncapsulation.exe
1
2
3
```

Listing 1.50 refSideEffect.fsx:
Intertwining independent scopes is typically a bad idea.

```
let updateFactor factor =
  factor := 2

let multiplyWithFactor x =
  let a = ref 1
  updateFactor a
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
$ fsharpc --nologo refSideEffect.fsx && mono
  refSideEffect.exe
6
```

Listing 1.51 refWithoutSideEffect.fsx:
A solution similar to Listing 1.50 without side-effects.

```
let updateFactor () =
  2

let multiplyWithFactor x =
  let a = ref 1
  a := updateFactor ()
  !a * x

printfn "%d" (multiplyWithFactor 3)
```

```
$ fsharpc --nologo refWithoutSideEffect.fsx
$ mono refWithoutSideEffect.exe
6
```

Listing 1.52 refCellAliasing.fsx:
Aliasing can cause surprising results and should be avoided.

```
let a = ref 1
let b = a
printfn "%d, %d" !a !b
b := 2
printfn "%d, %d" !a !b
```

```
$ fsharpc --nologo refCellAliasing.fsx && mono
  refCellAliasing.exe
1, 1
2, 2
```

Listing 1.53 mutableEncapsulation.fsx:
Local mutable content can be indirectly accessed outside its scope.

```
let incr =
  let mutable counter = 0
  fun () ->
    counter <- counter + 1
    counter
printfn "%d" (incr ())
printfn "%d" (incr ())
printfn "%d" (incr ())
```

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

1.9 Tuples

Tuples are a direct extension of constants. They are immutable and have neither concatenations nor indexing operations. Tuples are unions of immutable types and have the following syntax:

Listing 1.54: Tuples are list of expressions separated by commas.

```
<expr>{, <expr>}
```

Tuples are identified by the "," lexeme and often enclosed in parentheses, but that is not required. An example is a triple, also known as a 3-tuple, `(2,true,"hello")`. In interactive mode, the type of tuples is demonstrated in Listing 1.55. The values

Listing 1.55: Tuple types are products of sets.

```
> let tp = (2, true, "hello")
- printfn "%A" tp;;
(2, true, "hello")
val tp : int * bool * string = (2, true, "hello")
val it : unit = ()
```

`2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. From the response of F#, we see that the tuple is inferred to have the type `int * bool * string`. The "`*`" denotes the Cartesian product between sets. Tuples can be products of any types and follow the lexical scope rules like value and function bindings. Notice also that a tuple may be printed as a single entity by the `%A` placeholder. In the example we bound `tp` to the tuple. The opposite is also possible, as demonstrated in Listing 1.56. In this example, a function is defined that takes 1 argument, a 3-tuple. If we wanted a function with 3 arguments, then the function binding should have been `let deconstructNPrint a b c = ...`. The value binding `let (a, b, c) = tp`, binds a tuple with 3 named members to a value, thus deconstructing it in terms of its members. This is called pattern matching and will be discussed in further details in **??**. Since we used the `\%A` placeholder in the `printfn` function, the function can be called with 3-tuples of different types. F# informs us that the tuple type is variable by writing `'a * 'b * 'c`. The "`'`" notation means that the type can be decided at run-time, see **??** for more on variable types.

Pairs or 2-tuples are so common that F# includes two built-in functions, ***fst*** and ***snd***, to extract the first and second element of a pair. This is demonstrated in Listing 1.57.

Listing 1.56: Definition of a tuple.

```
> let deconstructNPrint tp =
-   let (a, b, c) = tp
-   printfn "tp = (%A, %A, %A)" a b c
-
- deconstructNPrint (2, true, "hello")
- deconstructNPrint (3.14, "Pi", 'p');;
tp = (2, true, "hello")
tp = (3.14, "Pi", 'p')
val deconstructNPrint : 'a * 'b * 'c -> unit
val it : unit = ()
```

Listing 1.57 pair.fsx:
Deconstruction of pairs with the built-in functions `fst` and `snd`.

```
let pair = ("first", "second")
printfn "fst(pair) = %s, snd(pair) = %s" (fst pair) (snd
  pair)
```

```
$ fsharpc --nologo pair.fsx && mono pair.exe
fst(pair) = first, snd(pair) = second
```

Tuples of equal lengths can be compared, and the comparison is defined similarly to string comparison. Tuples of equal length are compared element by element. E.g., `(1,2) = (1,3)` is false, while `(1,2) = (1,2)` is true. The "<>" operator is the boolean negation of the "=" operator. For the "<" , "<=", ">", and ">=" operators, the strings are ordered lexicographically, such that `('a', 'b', 'c') < ('a', 'b', 's') && ('a', 'b', 's') < ('c', 'o', 's')` is true, that is, the "<" operator on two tuples is true if and only if the left operand should come before the right when sorting alphabetically. See Listing 1.58 for an example. The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `a1` and `b1` are different, then the result of `(a1, a2) < (b1, b2)` is equal to the result of `a1 < b1`. If `a1` and `b1` are equal, then we move on to the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable. This is demonstrated in Listing 1.59. However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it, as shown in Listing 1.60. Mutable tuples are value types, meaning that binding to new names makes copies, not aliases, as demonstrated in Listing 1.61. The use of tuples shortens code and highlights

Listing 1.58 tupleCompare.fsx:
Tuples comparison is similar to string comparison.

```
let lessThan (a, b, c) (d, e, f) =
  if a <> d then a < d
  elif b <> e then b < d
  elif c <> f then c < f
  else false

let printTest x y =
  printfn "%A < %A is %b" x y (lessThan x y)

let a = ('a', 'b', 'c');
let b = ('d', 'e', 'f');
let c = ('a', 'b', 'b');
let d = ('a', 'b', 'd');
printTest a b
printTest a c
printTest a d
```

```
$ fsharpc --nologo tupleCompare.fsx && mono
   tupleCompare.exe
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

semantic content at a higher level, e.g., instead of focusing on the elements, tuples focus on their union. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, where an unprepared reader of the code may not easily understand the computation nor appreciate its elegance without an accompanying explanation. Hence, **always keep an eye out for compact and concise ways to write code, but never at the expense of readability.** ⋆

Listing 1.59 tupleOfMutables.fsx:
A mutable changes value, but the tuple defined by it does not refer to the new value.

```
let mutable a = 1
let mutable b = 2
let c = (a, b)
printfn "%A, %A, %A" a b c
a <- 3
printfn "%A, %A, %A" a b c
```

```
$ fsharpc --nologo tupleOfMutables.fsx && mono
  tupleOfMutables.exe
1, 2, (1, 2)
3, 2, (1, 2)
```

Listing 1.60 mutableTuple.fsx:
A mutable tuple can be assigned a new value.

```
let mutable pair = 1,2
printfn "%A" pair
pair <- (3,4)
printfn "%A" pair
```

```
$ fsharpc --nologo mutableTuple.fsx && mono
  mutableTuple.exe
(1, 2)
(3, 4)
```

Listing 1.61 mutableTupleValue.fsx:
A mutable tuple is a value type.

```
let mutable pair = 1,2
let mutable aCopy = pair
pair <- (3,4)
printfn "%A %A" pair aCopy
```

```
$ fsharpc --nologo mutableTupleValue.fsx && mono
  mutableTupleValue.exe
(3, 4) (1, 2)
```