Chapter 1

Imperative programming

Abstract Imperative programming is both and overarching term for a collection of programming paradigms and the name for specific paradigm. Here we will discuss the specific paradigm, and in **??–??** we will discuss another imperative paradigm called the object-oriented paradigm. When programming using the imperative paradigm, the programmer specifies a number of steps which to maninipulate the state of the computer for a desired result. Key features of this paradigm are *mutable values* also know as *variables* and `for`- and `while`-loops. In this chapter, you will learn

- How to define and use variables
- How to make loops using the `for`- and `while`-keywords as an alternative to recursive functions
- How to arrays as alternative to lists
- How to trace-by-hand imperative programs

1.1 Variables

A state is a value that may change over time. E.g., a traffic light as shown in Figure 1.1, consists of 3 colored lamps: red (top), yellow (middle), and green (bottom), and typically cycles through the cyclic sequence red, red+yellow, green, yellow. A simple

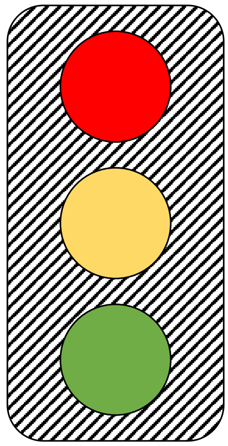

Fig. 1.1 For a traffic light, the different state of the coloured lamps can be modelled as different states of the ligth. Top lamp is red, middle is yellow, and bottom is green.

model of this could be to represent the state of each lamp as being on or off. This can be done with a mutable boolean value. To create a mutable value, we initally declare and identifyer using the `mutable` keyword with the following syntax:

Listing 1.1: Syntax for defining mutable values with an initial value.

```
let mutable <ident> = <expr>
```

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

Listing 1.2: Value reassignment for mutable variables.

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

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 1.3 for an example. Here, an area in memory was denoted `x`, initially assigned the integer 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 1.4 is not an assignment but a comparison which is evaluated to be false.

Listing 1.3 mutableAssignReassingShort.fsx:
A variable is defined and later reassigned a new value.

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

```
$ dotnet fsi mutableAssignReassingShort.fsx
5
-3
```

Listing 1.4: 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 1.5. I.e., once the type of an identifier has been declared or inferred, it cannot be changed.

Listing 1.5 mutableAssignReassingTypeError.fsx:
Assignment type mismatching causes a compile-time error.

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

```
$ dotnet fsi mutableAssignReassingTypeError.fsx
mutableAssignReassingTypeError.fsx(3,6): error FS0001: This
   expression was expected to have type
    'int'
but here has type
    'float'
```

A typical variable is a counter of type integer, and a typical use of counters is to increment them, see Listing 1.6 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 1.7. In the example, we see that the type is a value, and not mutable.

Listing 1.6 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
```

```
$ dotnet fsi mutableAssignIncrement.fsx
5
6
```

Listing 1.7: Returning a mutable variable returns its value.

```
> let g () =
  let mutable y = 0
  y
let y' = g();;
val g: unit -> int
val y': int = 0
```

In F# it is possible to encapsulate a mutable value. For example, consider a counter function `inc: unit->int`, which increments a counting state and returns its present value, every time we call it, i.e., calling `inc ()` the first time should return the value 1, the second time the value 2, and so on. A first attempt could be as shown in Listing 1.8. Even though `inc` has an internal state, the identifier “`s`” is reset to

Listing 1.8 inc.fsx:
A failed version of the a counter function `inc`.

```
let inc () =
  let mutable s = 0
  s <- s + 1
  s

printfn "%A" (inc ())
printfn "%A" (inc ())
printfn "%A" (inc ())
```

```
$ dotnet fsi inc.fsx
1
1
1
```

0 every time the function is called. However, in F# it is possible to solve this problem by using a side-effect as shown in Listing 1.9. The reason this works is that `makeCounter` returns a closure that includes the environment of the anonymous function at the point where it is defined. Hence, this closure includes a mutable value but does not reset it, every time it is called.

Listing 1.9 makeCounter.fsx:
Returning a counter function with a side-effect. Compare with Listing 1.8.

```
let makeCounter () =
  let mutable s = 0
  fun () ->
    s <- s + 1
    s

let inc = makeCounter ()
printfn "%A" (inc ())
printfn "%A" (inc ())
printfn "%A" (inc ())
```

```
$ dotnet fsi makeCounter.fsx
1
2
3
```

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 **??** which defines a function using lexical scope and returns the number `6.0`, however, if `a` is made `mutable`, then the behavior is different, as shown in Listing 1.10. Here, the

Listing 1.10 dynamicScopeNFunction.fsx:
Mutual variables implement dynamic scope rules. Compare with ??.

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

```
$ dotnet fsi dynamicScopeNFunction.fsx
8.0
```

response is `8.0`, since the value of `a` changed before the function `f` was called.

1.2 While and For Loops

This book has previously emphasized recursion as a structure to encapsulate and repeat code. In the imperative paradigm, often *`for`*- and *`while`*-loops are preferred. A *`while`*-loop has the following syntax:

Listing 1.11: While loop.

```
while <condition> do
  <expr>
```

The *condition* <condition> is an expression that evaluates to true or false. A while-loop repeats the <expr> expression as long as the condition is true. The *do* keyword is mandatory and the body of the loop is indicated by indentation as usual. The return value of the while expression is "()".

The program in Listing 1.14 is an example of a while-loop which counts from 1 to 10. Since the variable i is used for counting, it is often called the counter variable.

Listing 1.12 countWhile.fsx:
Count to 10 with a counter variable.

```
1 let mutable i = 1
2 while i <= 10 do
3   printf "%d " i
4   i <- i + 1
5 printf "\n"
```

```
1 $ dotnet fsi countWhile.fsx
2 1 2 3 4 5 6 7 8 9 10
```

The counting is done by performing the following computation: In line 1, the counter variable is first given an initial value of 1. Then in line 2, the head of the while-loop and examines the condition. Since 1 <= 10, the condition is true, and execution enters the body of the loop starting in line 3. The body prints the value of the counter to the screen and increases the counter by 1. Then execution returns to the head of the while-loop and reexamines the condition. Now the condition is 2 <= 10, which is also true, and so execution enters the body and so on until the counter has reached the value 11, in which case the condition 11 <= 10 is false, and execution continues in line 5.

Counters are so common that a special syntax has been reserved for loops using counters. These are called *for*-*to*-loops. For-loops come in several variants, and here we will focus on the one using an explicit counter. Its syntax is:

Listing 1.13: For loop.

```
for <ident> = <firstExpr> to <lastExpr> do
  <expr>
```

A for-loop initially binds the counter identifier <ident> to be the value <firstExpr>. Then execution enters the body, and <bodyExpr> is evaluated. Once done, the counter is increased, and execution evaluates <bodyExpr> once again. This is repeated as long as the counter is not greater than <lastExpr>. Againg, the *do* keyword is mandatory and the body of the loop is indicated by indentation as usual. The return value of the `for` expression is "`()`".

The counting example from Listing 1.12 using a `for`-loop is shown in Listing 1.14 As this interactive script demonstrates, the identifier `i` takes all the values between

Listing 1.14 count.fsx:
Counting from 1 to 10 using a `for`-loop.

```
for i = 1 to 10 do printf "%d " i done
printfn ""
```

```
$ dotnet fsi count.fsx
1 2 3 4 5 6 7 8 9 10
```

1 and 10, but in spite of its changing state, it is not mutable.

Counting backwards is sufficiently common that F# has a *`for`-`downto`* structure, which works exactly like a `for`-`to`-loop except that the counter is decreased by 1 in each iteration. An example of this is shown in Listing 1.15.

Listing 1.15 countBackwards.fsx:
Counting from 10 to 1 using a `for`-`downto`-loop.

```
for i = 10 downto 1 do
  printf "%d " i
printfn ""
```

```
$ dotnet fsi countBackwards.fsx
10 9 8 7 6 5 4 3 2 1
```

There is also a customized syntax for indexing lists as shown in Listing 1.16 This

Listing 1.16 listFor.fsx:
Iterating over elements of a list with the `for`-`in`-loop.

```
for elm in [3..2..9] do
  printf "%A " elm
printfn ""
```

```
$ dotnet fsi listFor.fsx
3 5 7 9
```

particular syntax for sequentially indexing into lists using a for loop is to be prefered, since it completely avoids index-out-of-bound errors.

1.3 Programming Intermezzo: Imperative Fibonacci numbers

To further compare for- and while-loops, consider the following problem.

Problem 1.1
Write a program that calculates the n'th Fibonacci number.

Fibonacci numbers is a sequence of numbers starting with 1, 1, and where the next number is calculated as the sum of the previous two. Hence the first ten numbers are: $1, 1, 2, 3, 5, 8, 13, 21, 34, 55$. Fibonacci numbers are related to Golden spirals shown in Figure 1.2. Often the sequence is extended with a preceding number 0, to

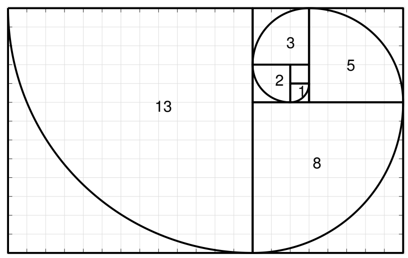

Fig. 1.2 The Fibonacci spiral is an approximation of the golden spiral. Each square has side lengths of successive Fibonacci numbers, and the curve in each square is the circular arc with a radius of the square it is drawn in.

be $0, 1, 1, 2, 3, \dots$, which we will do here as well.

In **??** we gave a solution using recursion. Here we will first use a `for`-loop, as shown in Listing 1.17. The basic idea of the solution is that if we are given the $(n-1)$'th and $(n-2)$'th numbers, the n'th number is trivial to compute. And assuming that fib(1) and fib(2) are given, then it is trivial to calculate fib(3). For fib(4), we only need fib(3) and fib(2), hence we may disregard fib(1). Thus, we realize that we can cyclicly update the previous, current, and next values by shifting values until we have reached the desired fib(n). This is implement in Listing 1.17 as the function `fib`, which takes an integer `n` as argument and returns the n'th Fibonacci number. The function does this iteratively using a `for`-loop, where `i` is the counter value, and `pair` is the pair of the $i-1$'th and i'th Fibonacci numbers. In the body of the loop, the i'th and $i+1$'th numbers are assigned to `pair`. The `for`-loop automatically updates `i` for next iteration. When $n < 2$ the body of the for-loop is not evaluated, and 1 is returned. This is of course wrong for $n < 1$, but we will ignore this for now.

Listing 1.18 shows a program similar to Listing 1.17 using a while-loop instead of for-loop. The programs are almost identical. In this case, the `for`-loop is to be preferred, since more lines of code typically mean more chances of making a mistake. However, while-loops are somewhat easier to argue correctness about.

Listing 1.17 fibFor.fsx:
The n'th Fibonacci number calculated using a for-loop.

```
let fib n =
  let mutable pair = (0, 1)
  for i = 2 to n do
    pair <- (snd pair, (fst pair) + (snd pair))
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ dotnet fsi fibFor.fsx
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

Listing 1.18 fibWhile.fsx:
The n'th Fibonacci number calculated using a while-loop.

```
let fib (n : int) : int =
  let mutable pair = (0, 1)
  let mutable i = 1
  while i < n do
    pair <- (snd pair, fst pair + snd pair)
    i <- i + 1
  snd pair

printfn "fib(1) = %d" (fib 1)
printfn "fib(2) = %d" (fib 2)
printfn "fib(3) = %d" (fib 3)
printfn "fib(10) = %d" (fib 10)
```

```
$ dotnet fsi fibWhile.fsx
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(10) = 55
```

While-loops also allow for logical structures other than for-loops, such as the case when the number of iteration cannot easily be decided when entering the loop. As an example, consider a slight variation of the above problem, where we wish to find the largest Fibonacci number less or equal some number. A solution to this problem is shown in Listing 1.19. The strategy here is to iteratively calculate Fibonacci numbers until we've found one larger than the argument `n`, and then return the previous. This could not be calculated with a for-loop.

Listing 1.19 fibWhileLargest.fsx:
Search for the largest Fibonacci number less than a specified number.

```
let largestFibLeq n =
  let mutable pair = (0, 1)
  while snd pair <= n do
    pair <- (snd pair, fst pair + snd pair)
  fst pair

for i = 1 to 10 do
  printfn "largestFibLeq(%d) = %d" i (largestFibLeq i)
```

```
$ dotnet fsi fibWhileLargest.fsx
largestFibLeq(1) = 1
largestFibLeq(2) = 2
largestFibLeq(3) = 3
largestFibLeq(4) = 3
largestFibLeq(5) = 5
largestFibLeq(6) = 5
largestFibLeq(7) = 5
largestFibLeq(8) = 8
largestFibLeq(9) = 8
largestFibLeq(10) = 8
```

1.4 Arrays

One dimensional *arrays*, or just arrays for short, are mutable lists of the same type and follow a similar syntax as lists. Arrays can be stated as a *sequence expression*,

Listing 1.20: The syntax for an array using the sequence expression.

```
[|[<expr>{; <expr>}]|]
```

E.g., `[|1; 2; 3|]` is an array of integers, `[|"This"; "is"; "an"; "array"|]` is an array of strings, `[|(fun x -> x); (fun x -> x*x)|]` is an array of functions, `[||]` is the empty array. Arrays may also be given as ranges,

Listing 1.21: The syntax for an array using the range expression.

```
[|<expr> .. <expr> [.. <expr>]|]
```

but arrays of *range expressions* must be of `<expr>` integers, floats, or characters. Examples are `[|1 .. 5|]`, `[|-3.0 .. 2.0|]`, and `[|'a' .. 'z'|]`. Range expressions may include a step size, thus, `[|1 .. 2 .. 10|]` evaluates to `[|1; 3; 5; 7; 9|]`.

The array type is defined using the `array` keyword or alternatively the "`[]`" lexeme. Like strings and lists, arrays may be indexed using the "*`[]`*" notation. Arrays cannot be resized, but are mutable, as shown in Listing 1.22. Notice that in spite of

Listing 1.22 arrayReassign.fsx:
Arrays are mutable in spite of the missing `mutable` keyword.

```
let square (a : int array) =
  for i = 0 to a.Length - 1 do
    a[i] <- a[i] * a[i]

let A = [| 1; 2; 3; 4; 5 |]
printfn "%A" A
square A
printfn "%A" A
```

```
$ dotnet fsi arrayReassign.fsx
[|1; 2; 3; 4; 5|]
[|1; 4; 9; 16; 25|]
```

the missing `mutable` keyword, the function `square` still has the *side-effect* of squaring all entries in `A`. F# implements arrays as chunks of memory and indexes arrays via address arithmetic. I.e., element i in an array, whose first element is in memory address α and whose elements fill β addresses each, is found at address $\alpha + i\beta$. Hence, indexing has computational complexity of $O(1)$, but appending and prepending values to arrays and array concatenation requires copying the new and existing values to a fresh area in memory and thus has computational complexity $O(n)$, where n is the total number of elements. Thus, **indexing arrays is fast, but cons and concatenation is slow and should be avoided.** ⋆

Arrays support *slicing*, that is, indexing an array with a range result in a copy of the array with values corresponding to the range. This is demonstrated in Listing 1.23. As illustrated, the missing start or end index imply from the first or to the last element, respectively.

Arrays do not have explicit operator support for appending and concatenation, instead the `Array` namespace includes an `Array.append` function, as shown in Listing 1.24.

Arrays are *reference types*, meaning that identifiers are references and thus suffer from aliasing, as illustrated in Listing 1.25.

Listing 1.23: Examples of array slicing. Compare with ?? and ??.

```
> let arr = [|'a' .. 'g'|];;
val arr: char[] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]

> arr[0];;
val it: char = 'a'

> arr[3];;
val it: char = 'd'

> arr[3..];;
val it: char[] = [|'d'; 'e'; 'f'; 'g'|]

> arr[..3];;
val it: char[] = [|'a'; 'b'; 'c'; 'd'|]

> arr[1..3];;
val it: char[] = [|'b'; 'c'; 'd'|]

> arr[*];;
val it: char[] = [|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

Listing 1.24 arrayAppend.fsx:
Two arrays are appended with `Array.append`.

```
let a = [|1; 2;|]
let b = [|3; 4; 5|]
let c = Array.append a b
printfn "%A, %A, %A" a b c
```

```
$ dotnet fsi arrayAppend.fsx
[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
```

Listing 1.25 arrayAliasing.fsx:
Arrays are reference types and suffer from aliasing.

```
let a = [|1; 2; 3|];
let b = a
a[0] <- 0
printfn "a = %A, b = %A" a b;;
```

```
$ dotnet fsi arrayAliasing.fsx
a = [|0; 2; 3|], b = [|0; 2; 3|]
```

1.4.1 Array Properties and Methods

Some important properties and methods for arrays are:

`Clone()`: `'T []`.
: Returns a copy of the array.

Listing 1.26: `Clone`

```
> let a = [|1; 2; 3|];
let b = a.Clone()
a[0] <- 0
printfn "a = %A, b = %A" a b;;
a = [|0; 2; 3|], b = [|1; 2; 3|]
val a: int[] = [|0; 2; 3|]
val b: obj = [|1; 2; 3|]
val it: unit = ()
```

`Length`: `int`.
: Returns the number of elements in the array.

Listing 1.27: `Length`

```
> [|1; 2; 3|].Length;;
val it: int = 3
```

1.4.2 The Array Module

There are quite a number of built-in procedures for arrays in the `Array` module, and many of them are almost identical to those in the `List` module, discussed in **??**. However, a few additional functions are noted below:

`Array.append`: `arr1:'T [] -> arr2:'T [] -> 'T []`.
: Creates an new array whose elements are a concatenated copy of `arr1` and `arr2`.

Listing 1.28: `Array.append`

```
> Array.append [|1; 2;|] [|3; 4; 5|];;
val it: int[] = [|1; 2; 3; 4; 5|]
```

`Array.copy`: `'T [] -> 'T []`.
: Creates an array that contains the elements of the supplied array.

Listing 1.29: Array.copy

```
> let a = [|1; 2; 3|]
let b = Array.copy a;;
val a: int[] = [|1; 2; 3|]
val b: int[] = [|1; 2; 3|]
```

`Array.ofList`: `lst:'T list -> 'T []`.
Creates an array whose elements are copied from `lst`.

Listing 1.30: Array.ofList

```
> Array.ofList [1; 2; 3];;
val it: int[] = [|1; 2; 3|]
```

`Array.toList`: `arr:'T [] -> 'T list`.
Creates a new list whose elements are copied from `arr`.

Listing 1.31: Array.toList

```
> Array.toList [|1; 2; 3|];;
val it: int list = [1; 2; 3]
```

1.4.3 Multidimensional Arrays

Multidimensional arrays can be created as arrays of arrays (of arrays ...). These are known as *jagged arrays*, since there is no inherent guarantee that all sub-arrays are of the same size. The example in Listing 1.32 is a jagged array of increasing width. Indexing arrays of arrays is done sequentially, in the sense that in the above example,

Listing 1.32 arrayJagged.fsx:
An array of arrays of non-equal lenghts is a jagged array.

```
let arr = [|[|1|]; [|1; 2|]; [|1; 2; 3|]|]

for row in arr do
  for elm in row do
    printf "%A " elm
  printf "\n"
```

```
$ dotnet fsi arrayJagged.fsx
1
1 2
1 2 3
```

the number of outer arrays is `a.Length`, `a[i]` is the i'th array, the length of the i'th array is `a[i].Length`, and the j'th element of the i'th array is thus `a[i][j]`. Often 2-dimensional rectangular arrays are used, which can be implemented as a jagged array, as shown in Listing 1.33. Note that, the `arr[i][j]` argument in line 4

Listing 1.33 arrayJaggedSquare.fsx:
A rectangular array.

```
1  let pownArray  (arr : int array array) p =
2    for i = 1 to arr.Length - 1 do
3      for j = 1 to arr[i].Length - 1 do
4        arr[i][j] <- pown (arr[i][j]) p
5
6  let printArrayOfArrays (arr : int array array) =
7    for row in arr do
8      for elm in row do
9        printf "%3d " elm
10     printf "\n"
11
12 let A = [|[|1 .. 4|]; [|1 .. 2 .. 7|]; [|1 .. 3 .. 10|]|]
13 pownArray A 2
14 printArrayOfArrays A
```

```
1  $ dotnet fsi arrayJaggedSquare.fsx
2    1   2   3   4
3    1   9  25  49
4    1  16  49 100
```

must be parenthesized to avoid ambiguity. Further, the `for-in` cannot be used in `pownArray`, e.g.,

```
for row in arr do
  for elm in row do
    elm <- pown elm p
```

since the iterator value `elm` is not mutable, even though `arr` is an array.

Square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. Here, we will describe *`Array2D`*. The workings of *`Array3D`* and *`Array4D`* are very similar. A generic `Array2D` has type `'T [,]`, and it is indexed also using the `[,]` notation. The `Array2D.length1` and `Array2D.length2` functions are supplied by the `Array2D` module for obtaining the size of an array along the first and second dimension. Rewriting the with jagged array example in Listing 1.33 to use `Array2D` gives a slightly simpler program, which is shown in Listing 1.34. Note that the `printf` supports direct printing of the 2-dimensional array. `Array2D` arrays support slicing. The "*" lexeme is particularly useful to obtain all values along a dimension. This is demonstrated in Listing 1.35. Note that in almost all cases, slicing produces a sub-rectangular 2 dimensional array, except for `arr[1,*]`, which is an array, as can be seen by the single "`[`". In contrast, `A[1..1,*]` is an

Listing 1.34 array2D.fsx:
Creating a 3 by 4 rectangular array of integers.

```
let arr = Array2D.create 3 4 0
for i = 0 to (Array2D.length1 arr) - 1 do
  for j = 0 to (Array2D.length2 arr) - 1 do
    arr[i,j] <- j * Array2D.length1 arr + i
printfn "%A" arr
```

```
$ dotnet fsi array2D.fsx
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Listing 1.35: Examples of Array2D slicing. Compare with Listing 1.34.

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j);;
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]

> arr[2,3];;
val it: int = 32

> arr[1..,3..];;
val it: int[,] = [[31]
                  [32]]

> arr[..1,*];;
val it: int[,] = [[0; 10; 20; 30]
                  [1; 11; 21; 31]]

> arr[1,*];;
val it: int[] = [|1; 11; 21; 31|]

> arr[1..1,*];;
val it: int[,] = [[1; 11; 21; 31]]
```

Array2D. Note also that `printfn` typesets 2 dimensional arrays as `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion with lists of lists.

Multidimensional arrays have the same properties and methods as arrays, see Section 1.4.1.

1.4.4 The Array2D Module

The `Array2D` module is somewhat limited. In particular neither `fold` nor `foldback` functions exists. Most of the module's functions are mentioned below:

`copy`: `arr:'T [,] -> 'T [,]`.
Creates a new array whose elements are copied from `arr`.

Listing 1.36: `Array2D.copy`

```
> let a = Array2D.init 3 4 (fun i j -> i + 10 * j)
let b = Array2D.copy a;;
val a: int[,] = [[0; 10; 20; 30]
                 [1; 11; 21; 31]
                 [2; 12; 22; 32]]
val b: int[,] = [[0; 10; 20; 30]
                 [1; 11; 21; 31]
                 [2; 12; 22; 32]]
```

`create`: `m:int -> n:int -> v:'T -> 'T [,]`.
Creates an `m` by `n` array whose elements are set to `v`.

Listing 1.37: `Array2D.create`

```
> Array2D.create 2 3 3.14;;
val it: float[,] = [[3.14; 3.14; 3.14]
                    [3.14; 3.14; 3.14]]
```

`init`: `m:int -> n:int -> f:(int -> int -> 'T) -> 'T [,]`.
Creates an `m` by `n` array whose elements are the result of applying `f` to the index of an element.

Listing 1.38: `Array2D.init`

```
> Array2D.init 3 4 (fun i j -> i + 10 * j);;
val it: int[,] = [[0; 10; 20; 30]
                  [1; 11; 21; 31]
                  [2; 12; 22; 32]]
```

`iter`: `f:('T -> unit) -> arr:'T [,] -> unit`.
Applies `f` to each element of `arr`.

Listing 1.39: Array2D.iter

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
Array2D.iter (fun elm -> printf "%A " elm) arr
printfn "";;
0 10 20 30 1 11 21 31 2 12 22 32
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
val it: unit = ()
```

`length1`: `arr:'T [,] -> int`.
Returns the length the first dimension of `arr`.

Listing 1.40: Array2D.length1

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length1
   arr;;
val it: int = 2
```

`length2`: `arr:'T [,] -> int`.
Returns the length of the second dimension of `arr`.

Listing 1.41: Array2D.forall length2

```
> let arr = Array2D.create 2 3 0.0 in Array2D.length2
   arr;;
val it: int = 3
```

`map`: `f:('T -> 'U) -> arr:'T [,] -> 'U [,]`.
Creates a new array whose elements are the results of applying `f` to each of the elements of `arr`.

Listing 1.42: Array2D.map

```
> let arr = Array2D.init 3 4 (fun i j -> i + 10 * j)
Array2D.map (fun x -> x * x) arr;;
val arr: int[,] = [[0; 10; 20; 30]
                   [1; 11; 21; 31]
                   [2; 12; 22; 32]]
val it: int[,] = [[0; 100; 400; 900]
                  [1; 121; 441; 961]
                  [4; 144; 484; 1024]]
```

1.5 Tracing Imperative Programs

Debugging imperative programs is more complicated than declarative programs. In particular, the notion of states require us to keep track of the dynamic scope of values.

In the following, we will discuss trace-by-hand of `for`-loops followed by programs which involves states.

1.5.1 Tracing Loops

Consider the program in Listing 1.43. The program includes a function for printing

Listing 1.43 printSquares.fsx:
Print the squares of a sequence of positive integers.

```
1 let N = 3
2 let printSquares n =
3   for i = 1 to n do
4     let p = i * i
5     printfn "%d: %d" i p
6
7 printSquares N
```

```
$ dotnet fsi printSquares.fsx
1: 1
2: 4
3: 9
```

the sequence of the first N squares of integers. It uses a `for`-loop with a counting value. F# creates a new environment each time the loop body is executed. Thus, to trace this program, we mentally *unfold* the loop as shown in Listing 1.44. The unfolding contains 3 new scopes lines 3–7, lines 8–12, and lines 13–17 corresponding to the 3 times, the loop is repeated, and each scope starts by binding the counting value to the name `i`.

In the rest of this section, we will refer to the code in Listing 1.43. The first rows in our tracing-table looks as follows:

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	$N = 3$
2	2	E_0	printSquares = $\big((n), \text{printSquares-body}, (N = 3)\big)$
3	7	E_0	printSquares N = ?

Note that due to the lexical scope rule, the closure of `printSquares` includes `N` in its environment element. Calling `printSquares N` causes the creation of a new environment,

Listing 1.44 printSquaresUnfold.fsx:
An unfolded version of Listing 1.43.

```
let N = 3
let printSquaresUnfold n =
  (
    let i = 1
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 2
    let p = i * i
    printfn "%d: %d" i p
  )
  (
    let i = 3
    let p = i * i
    printfn "%d: %d" i p
  )

printSquaresUnfold N
```

```
$ dotnet fsi printSquaresUnfold.fsx
1: 1
2: 4
3: 9
```

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$\big((n = 3), \text{printSquares-body}, (N = 3)\big)$

The first statement of printSquares-body is the `for`-loop. As our unfolding in Listing 1.44 demonstrated, each time the loop-body is executed, a new scope is created with a new binding to `i`. Reusing the notation from closures, we write

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$\text{for} \cdots = ?$

and create a new environment as if it had been a function,

Step	Line	Env.	Bindings and evaluations
6	3	E_2	$\big((i = 1), \text{for-body}, (n = 3, N = 3)\big)$

As for functions, this denotes the bindings available at beginning of the execution of the `for`-body. The first line in the `for`-body is the binding of the value of an expression to `p`. The expression is `i*i`, and to calculate its value, we look in the `for`-loop's pseudo-closure where we find the $i = 1$ binding. Hence,

Step	Line	Env.	Bindings and evaluations
7	4	E_2	$i * i = 1$
8	4	E_2	$p = 1$

The final step in the for-body is the `printfn`-statement. Its arguments we get from the updated, active environment E_2 and write,

Step	Line	Env.	Bindings and evaluations
9	5	E_2	output = "1 : 1\n"

At this point, the `for`-loop has reached its last line, E_2 is deleted, we create a new environment with the counter variable increased by 1, and repeat. Hence,

Step	Line	Env.	Bindings and evaluations
10	3	E_3	$\big((i = 2), \text{for-body}, (n = 3, N = 3)\big)$
11	4	E_3	$i * i = 4$
12	4	E_3	$p = 4$
13	5	E_3	output = "2 : 4\n"

Again, we delete E_3, create E_4 where i is incremented, and repeat,

Step	Line	Env.	Bindings and evaluations
14	3	E_4	$\big((i = 3), \text{for-body}, (n = 3, N = 3)\big)$
15	4	E_4	$i * i = 9$
16	4	E_4	$p = 9$
17	5	E_4	output = "3 : 9\n"

Finally, incrementing i would mean that $i > n$, hence the `for`-loop ends and as all statements returns `()`

Step	Line	Env.	Bindings and evaluations
18	3	E_4	return = ()

At this point, the environment E_4 is deleted, and we return to the enclosing environment E_1 and the statement or expression following Step 5. Since the `for`-loop is the last expression in the `printSquares` function, its return value is that of the `for`-loop,

Step	Line	Env.	Bindings and evaluations
19	3	E_1	return = ()

Returning to Step 3 and environment E_0, we have now calculated the return-value of `printSquares N` to be `()`, and since this line is the last of our program, we return `()` and end the program:

Step	Line	Env.	Bindings and evaluations
20	3	E_0	return = ()

1.5.2 Tracing Mutable Values

For mutable bindings, the scope is dynamic. For this, we need the concept of storage. Consider the program in Listing 1.45. To trace the dynamic behavior of this program,

Listing 1.45 dynamicScopeTracing.fsx:
Example of lexical scope and closure environment.

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

```
1 $ dotnet fsi dynamicScopeTracing.fsx
2 8.0
```

we add a second table to our hand tracing, which is initially empty and has the columns Step and Value to hold the Step number when the value was updated and the value stored. For Listing 1.45, the firsts 4 steps thus look like,

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$

Step	Value
0	-

The mutable binding in line 2 creates an internal name and a dynamic storage location. The name `a` will be bound to a reference value, which we call α_1, and which is a unique name shared between the two tables:

Step	Line	Env.	Bindings and evaluations
4	2	E_1	$a = \alpha_1$

Step	Value
4	$\alpha_1 = 3.0$

The following closure of `f` uses the reference-name instead of its value,

Step	Line	Env.	Bindings and evaluations
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$

Step	Value
4	$\alpha_1 = 3.0$

In line 4, the value in the storage is updated by the assignment operator, which we denote as,

Step	Line	Env.	Bindings and evaluations
6	4	E_1	a <- 4.0

Step	Value
6	$\alpha_1 = 4.0$

Hence, when we evaluate the function `f`, its closure looks up the value of `a` by following the reference and finding the new value:

Step	Line	Env.	Bindings and evaluations
7	5	E_1	f x = ?
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = "8.0\n"
12	6	E_0	return = ()

Step	Value
6	$\alpha_1 = 4.0$

For reference, the complete pair of tables is shown in Table 1.1. By comparing this

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	testScope = $\big((x), \text{body}, ()\big)$
2	6	E_0	testScope 2.0 = ?
3	1	E_1	$\big((x = 2.0), \text{body}, ()\big)$
4	2	E_1	$a = \alpha_1$
5	3	E_1	$f = \big((z), a * z, (x = 2.0, a = \alpha_1)\big)$
6	4	E_1	a <- 4.0
7	5	E_1	f x = ?
8	5	E_2	$\big((z = 2.0), a * z, (x = 2.0, a = \alpha_1)\big)$
9	5	E_2	$a * z = 8.0$
10	5	E_2	return = 8.0
10	5	E_1	return = 8.0
11	6	E_0	output = "8.0\n"
12	6	E_0	return = ()

Step	Value
0	-
4	$\alpha_1 = 3.0$
6	$\alpha_1 = 4.0$

Table 1.1 The complete table produced while tracing the program in Listing 1.45 by hand.

to the value-bindings in **??**, we see that the mutable values give rise to a different result due to the difference between lexical and dynamic scope.

1.6 Key Concepts and Terms in This Chapter

In this chapter, we have looked at programming with states using the imperative programming paradigm. You have seen how to:

- define **mutable variables** and make loops with **while**- and **for**- loops
- work with **arrays**, **jagged arrays**, and **2-dimensional arrays**.
- **trace-by-hand** programs involving mutable values and for-loops.