Learning to program with F#

Jon Sporring

September 23, 2016

Chapter 9

Ordered series of data

F# is tuned to work with ordered series, and there are several built-in lists with various properties making them useful for different tasks. E.g.,

Listing 9.1, tuplesQuadraticEq.fsx:
Using tuples to gather values.

```
let solution a b c =
  let d = b ** 2.0 - 2.0 * a * c
  if d < 0.0 then
    (nan, nan)
  else
    let xp = (-b + sqrt d) / (2.0 * a)
    let xn = (-b - sqrt d) / (2.0 * a)
    (xp,xn)

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

```
0 = 1.0 * x ** 2.0 + 0.0 * x + -1.0
  has solutions -0.7071067812 and 0.7071067812
```

F# has four built-in list types: strings, tuples, lists, arrays, and sequences. Strings were discussed in Chapter 5, sequences will be discussed in Chapter 15. Here we will concentrate on tuples, lists, and arrays, and following this (simplified) syntax:

Listing 9.2:

```
expr = ...
  | exprTuple (*tuple*)
  | "[" (exprSeq | rangeExpr) "]" (*list*)
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)

exprTuple = expr | expr "," exprTuple;
exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
```

Tuples are a direct extension of constants. They are immutable and do not have concatenations nor indexing operations. This is in contrast to lists. Lists are also immutable, but have a simple syntax for concatenation and indexing. Arrays are mutable lists, and support higher order structures such as tables and three dimensional arrays.

9.1 Tuples

Tuples are unions of immutable types, · tuple

Listing 9.3:

```
expr = ...
  | exprTuple (*tuple*)

exprTuple = expr | expr "," exprTuple;
```

and the they are identified by the "`,`" lexeme. Most often the tuple is enclosed in parentheses, but that is not required. Consider the tripel, also known as a 3-tuple, `(2,true,"hello")` in interactive mode,

Listing 9.4: Definition of a tuple.

```
> let tp = (2, true, "hello")
- printfn "%A" tp;;
(2, true, "hello")

val tp : int * bool * string = (2, true, "hello")
val it : unit = ()
```

The values `2`, `true`, and `"hello"` are *members*, and the number of elements of a tuple is its *length*. · member
From the response of F# we see that the tuple is inferred to have the type `int * bool * string`, · length
where the "`*`" is cartesian product between the three sets. Notice, that tuples can be products of any types and have lexical scope 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,

Listing 9.5: 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 = ()
```

In this a function is defined that takes 1 argument, a 3-tuple, and which is bound to a tuple with 3 named members. Since we used the `%A` placeholder in the `printfn` function, then the function is generic and can be called with 3-tuples of different types. Note, **don't confuse a function of n arguments with a function of an n-tuple.** The later has only 1 argument, and the difference is the ","s. Another example is `let solution a b c = ...`, which is the beginning of the function definition in Listing 9.1. It is a function of 3 arguments, while `let solution (a, b, c) = ...` would be a function of 1 argument, which is a 3-tuple. Functions of several arguments makes currying easy, i.e., we could define a new function which fixes the quadratic term to be 0 as `let solutionToLinear = solution 0.0`, that is, without needing to specify anything else. With tuples, we would need the slightly more complicated, `let solutionToLinear (b, c) = solution (0.0, b, c)`.

Advice

Tuples comparison are defined similarly as strings. Tuples of different lengths are different. For tuples of equal length, then they 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 the left operand should come before the right, when sorting alphabetically like.

Listing 9.6, tupleCompare.fsx:
Tuples are compared as strings are compared alphabetically.

```
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
```

```
('a', 'b', 'c') < ('d', 'e', 'f') is true
('a', 'b', 'c') < ('a', 'b', 'b') is false
('a', 'b', 'c') < ('a', 'b', 'd') is true
```

The algorithm for deciding the boolean value of `(a1, a2) < (b1, b2)` is as follows: we start by examining the first elements, and if `la1` and `b1` are different, then the `(a1, a2) < (b1, b2)` is equal to `a1 < b1`. If `la1` and `b1` are equal, then we move onto the next letter and repeat the investigation. The "<=", ">", and ">=" operators are defined similarly.

Binding tuples to mutables does not make the tuple mutable, e.g.,

Listing 9.7, tupleOfMutables.fsx:
A mutable change 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
```

```
1, 2, (1, 2)
3, 2, (1, 2)
```

However, it is possible to define a mutable variable of type tuple such that new tuple values can be assigned to it, e.g., in the Fibonacci example, we can write a more compact script by using mutable tuples and the "`fst`" and "`snd`" functions as follows.

Listing 9.8, fibTuple.fsx:
Calculating Fibonacci numbers using a mutable tuple.

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

for i = 0 to 10 do
  printfn "fib(%d) = %d" i (fib i)
```

```
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
```

In this example, the central computation has been packed into a single line, `prev <- (snd prev, (fst prev) + (snd prev))`, where both the calculation of $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$ and the rearrangement of memory to hold the new values $\text{fib}(n)$ and $\text{fib}(n-1)$ based on the old values $\text{fib}(n-2)+\text{fib}(n-1)$. While this may look elegant and short there is the risk of *obfuscation*, i.e., writing compact code that is difficult to read, and in this case, 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.**

· obfuscation

Advice

9.2 Lists

Lists are unions of immutable values of the same type and have a more flexible structure than tuples. Its grammar follows *computation expressions*, which is very rich and shared with arrays and sequences, and we will delay a discussion on most computation expressions to Section 15.1, and here just consider a subset of the grammar:

· list
· computation expressions

Listing 9.9:

```
expr = ...
  | "[" (exprSeq | rangeExpr) "]" (*list*)

exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr [".." expr];
```

E.g., an explicit list `let lst = [1; 2; 3; 4; 5]`, which may be written shortly as *range expression* as `let lst = [1 .. 5]`, and ranges may include a step size `let lst = [1 .. 2 .. 5]`, which is the same as `let lst = [1; 3; 5]`.

· range expression

Lists may be indexed and concatenated much like strings, e.g.,

Listing 9.10, listIndexing.fsx:
Examples of list concatenation, indexing.

```
let printList (lst : int list) =
  for elm in lst do
    printf "%A " elm
  printfn ""

let printListAlt (lst : int list) =
  for i = 0 to lst.Length - 1 do
    printf "%A " lst.[i]
  printfn ""

let a = [1; 2;]
let b = [3; 4; 5]
let c = a @ b
let d = 0 :: c
printfn "%A, %A, %A, %A" a b c d
printList d
printListAlt d
```

```
[1; 2], [3; 4; 5], [1; 2; 3; 4; 5], [0; 1; 2; 3; 4; 5]
0 1 2 3 4 5
0 1 2 3 4 5
```

A list type is identified with the "`list`" keyword, as here a list of integers is `int list`. Above, we used the *"`@`"* and *"`::`"* concatenation operators, the *"`.[]`"* index method, and the *"`Length`"* property. There also exists a family of functions including `List.length`, `List.isEmpty`, `List.item`, `List.head`, `List.tail` for working with list. Notice, as strings, list elements are counted from 0, and thus the last element has `lst.Length - 1`. In `printList` the "`for`"-"`in`" is used, which runs loops through each element of the list and assigns it to the identifier `elm`. This is in contrast to `printListAlt`, which uses uses the "`for`"-"`to`" keyword and explicitly represents the index `i`. Explicit representation of the index makes more complicated programs, and thus increases the chances of programming errors. Hence, **"`for`"-"`in`" is to be preferred over "`for`"-"`to`".** Lists support slicing identically to strings, e.g.,

· "`@`"
· "`::`"
· "`.[]`"
· "`Length`"

Advice

Listing 9.11, listSlicing.fsx:
Examples of list slicing. Compare with Listing 5.33.

```
let lst = ['a' .. 'g']
printfn "%A" lst.[0]
printfn "%A" lst.[3]
printfn "%A" lst.[3..]
printfn "%A" lst.[..3]
printfn "%A" lst.[1..3]
printfn "%A" lst.[*]
```

```
'a'
'd'
['d'; 'e'; 'f'; 'g']
['a'; 'b'; 'c'; 'd']
['b'; 'c'; 'd']
['a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g']
```

The basic properties and members of lists are summarized in Table 9.1- 9.2. In addition, lists have many other built-in functions, such as functions for converting lists to arrays,

Listing 9.24, listConversion.fsx:
The `List` module contains functions for conversion to arrays.

```
let lst = ['a' .. 'c']
let arr = List.toArray lst
printfn "%A, %A" lst arr
```

```
['a'; 'b'; 'c'], [|'a'; 'b'; 'c'|]
```

These and more will be discussed in Chapter F and Part III.[1]

It is possible to make multidimensional lists as lists of lists, e.g.,

Listing 9.25, listMultidimensional.fsx:
A ragged multidimensional list, built as lists of lists, and its indexing.

```
let a = [[1;2];[3;4;5]]
let row = a.Item 0 in printfn "%A" row
let elm = row.Item 1 in printfn "%A" elm
let elm = (a.Item 0).Item 1 in printfn "%A" elm
```

```
[1; 2]
2
2
```

The example shows a *ragged multidimensional list*, since each row has different number of elements. The indexing of a particular element is not elegant, which is why arrays are often preferred in F#.

· ragged multidimensional list

[1] Todo: **Add description of prepend and concatenation operator for lists.**

Function name	Example	Description

`Length`	**Listing 9.12:**
	`> [1; 2; 3].Length;;`
	`val it : int = 3`
	`> let a = [1; 2; 3] in a.Length;;`
	`val it : int = 3`
		The number of elements in a list

`List.Length`	**Listing 9.13:**
	`> List.length [1; 2; 3];;`
	`val it : int = 3`
	`> let a = [1; 2; 3] in List.length a;;`
	`val it : int = 3`
		The number of elements in a list

`List.Empty`	**Listing 9.14:**
	`> let a : int list = List.Empty;;`
	`val a : int list = []`
	`> let b = List<int>.Empty;;`
	`val b : int list = []`
		An empty list of specified type

`IsEmpty`	**Listing 9.15:**
	`> [1; 2; 3].IsEmpty;;`
	`val it : bool = false`
	`> let a = [1; 2; 3] in a.IsEmpty;;`
	`val it : bool = false`
		Compare with the empty list

`List.isEmpty`	**Listing 9.16:**
	`> List.isEmpty [1; 2; 3];;`
	`val it : bool = false`
	`> let a = [1; 2; 3] in List.isEmpty a`
	`;;`
	`val it : bool = false`
		Compare with the empty list

`Item`	**Listing 9.17:**
	`> [1; 2; 3].Item 1;;`
	`val it : int = 2`
	`> let a = [1; 2; 3] in a.Item 1;;`
	`val it : int = 2`
		Indexing

`List.item`	**Listing 9.18:**
	`> List.item 1 [1; 2; 3];;`
	`val it : int = 2`
	`> let a = [1; 2; 3] in List.item 1 a;;`
	`val it : int = 2`
		Indexing

Table 9.1: Basic properties and members of lists. The syntax used in `List<int>.Empty` ensures that the empty list is of type `int`.

Function name	Example	Description

`Head`	**Listing 9.19:**
	`> [1; 2; 3].Head;;`
	`val it : int = 1`
	`> let a = [1; 2; 3] in a.Head;;`
	`val it : int = 1`
		The first element in the list. Exception if empty.

`List.head`	**Listing 9.20:**
	`> List.head [1; 2; 3];;`
	`val it : int = 1`
	`> let a = [1; 2; 3] in List.head a;;`
	`val it : int = 1`
		The first element in the list. Exception if empty.

`List.tail`	**Listing 9.21:**
	`> List.tail [1; 2; 3];;`
	`val it : int list = [2; 3]`
	`> let a = [1; 2; 3] in List.tail a;;`
	`val it : int list = [2; 3]`
		The list except its first element. Exception if empty.

`Cons`	**Listing 9.22:**
	`> list.Cons (1, [2; 3]);;`
	`val it : int list = [1; 2; 3]`
	`> 1 :: [2; 3];;`
	`val it : int list = [1; 2; 3]`
		Append an element to the front of the list

`@`	**Listing 9.23:**
	`> [1] @ [2; 3];;`
	`val it : int list = [1; 2; 3]`
	`> [1; 2] @ [3; 4];;`
	`val it : int list = [1; 2; 3; 4]`
	`> [1; 2] @ [3];;`
	`val it : int list = [1; 2; 3]`
		Concatenate two lists

Table 9.2: Basic properties and members of lists continued from Table 9.1.

9.3 Arrays

One dimensional arrays or just arrays for short are mutable lists of the same type and follow a similar syntax as lists. Its grammar follows *computation expressions*, which will be discussed in Section 15.1. Here we consider a subset of the grammar:

· computation expressions

Listing 9.26:

```
expr = ...
  | "[|" (exprSeq | rangeExpr) "|]" (*array*)

exprSeq =  expr | expr ";" exprSeq;
rangeExpr = expr ".." expr ["..." expr];
```

Thus the creation of arrays is identical to lists, but there is no explicit operator support for appending and concatenation, e.g.,

Listing 9.27, arrayCreation.fsx:
Creating arrays with a syntax similarly to lists.

```
let printArray (arr : int array) =
  for elm in arr do
    printf "%d " elm
  printf "\n"

let printArrayAlt (arr : int array) =
  for i = 0 to arr.Length - 1 do
    printf "%A " arr.[i]
  printfn ""

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

```
[|1; 2|], [|3; 4; 5|], [|1; 2; 3; 4; 5|]
1 2 3 4 5
1 2 3 4 5
```

The array type is defined using the "`array`" keyword or alternatively the "`[]`" lexeme. Arrays cannot be resized, but are mutable,

Listing 9.28, arrayReassign.fsx:
Arrays are mutable in spite the missing "mutable" keyword.

```
let printArray (a : int array) =
  for i = 0 to a.Length - 1 do
    printf "%d " a.[i]
  printf "\n"

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 |]

printArray A
square A
printArray A
```

```
1 2 3 4 5
1 4 9 16 25
```

Notice that in spite the missing "mutable" keyword, the function `square` still had the *side-effect* of squaring alle entries in `A`.

· side-effect

Arrays support *slicing*, that is, indexing an array with a range results in a copy of array with values corresponding to the range, e.g.,

· slicing

Listing 9.29, arraySlicing.fsx:
Examples of array slicing. Compare with Listing 9.11 and Listing 5.33.

```
let arr = [|'a' .. 'g'|]
printfn "%A" arr.[0]
printfn "%A" arr.[3]
printfn "%A" arr.[3..]
printfn "%A" arr.[..3]
printfn "%A" arr.[1..3]
printfn "%A" arr.[*]
```

```
'a'
'd'
[|'d'; 'e'; 'f'; 'g'|]
[|'a'; 'b'; 'c'; 'd'|]
[|'b'; 'c'; 'd'|]
[|'a'; 'b'; 'c'; 'd'; 'e'; 'f'; 'g'|]
```

As illustrated, the missing start or end index implies from the first or to the last element.

Arrays can be converted to lists by,

Listing 9.30, arrayConversion.fsx:
The `Array` module contains functions for conversion to lists.

```
let arr = [|'a' .. 'c'|]
let lst = Array.toList arr
printfn "%A, %A" arr lst
```

```
[|'a'; 'b'; 'c'|], ['a'; 'b'; 'c']
```

There are quite a number of built-in procedures for all arrays many which will be discussed in Chapter F.

Higher dimensional arrays can be created as arrays of arrays (of arrays . . .). These are known as *jagged arrays*, since there is no inherent control of that all sub-arrays are of similar size. E.g., the following is a jagged array of increasing width,

· jagged arrays

Listing 9.31, arrayJagged.fsx:
An array of arrays. When row lengths are of non-equal elements, then it 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"
```

```
1
1 2
1 2 3
```

Indexing arrays of arrays is done sequentially, in the sense that in the above example, 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,

Listing 9.32, arrayJaggedSquare.fsx:
A rectangular array.

```
let pownArray  (arr : int array array) p =
  for i = 1 to arr.Length - 1 do
    for j = 1 to arr.[i].Length - 1 do
      arr.[i].[j] <- pown arr.[i].[j] p

let printArrayOfArrays (arr : int array array) =
  for row in arr do
    for elm in row do
      printf "%3d " elm
    printf "\n"

let A = [|[|1 .. 4|]; [|1 .. 2 .. 7|]; [|1 .. 3 .. 10|]|]
pownArray A 2
printArrayOfArrays A
```

```
  1   2   3   4
  1   9  25  49
  1  16  49 100
```

Notice, the "for"-"in" cannot be used in `pownArray`, e.g., `for row in arr do for elm in row do elm <- pown elm p done done` since the iterator value `elm` is not mutable even though `arr` is an array. In fact, square arrays of dimensions 2 to 4 are so common that F# has built-in modules for their support. In the following, we describe Array2D. The workings of Array3D and Array4D are very similar. An example of creating the same 2 dimensional array as above but as an `Array2D` is,

Listing 9.33, array2D.fsx:
Creating a 3 by 4 rectangular arrays of intigers.

```
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
```

```
[[0; 3; 6; 9]
 [1; 4; 7; 10]
 [2; 5; 8; 11]]
```

Notice that the indexing uses a slightly different notation `[,]` and the length functions are also slightly different. The statement `A.Length` would return the total number of elements in the array, in this case 12. As can be seen, the `printf` supports direct printing of the 2 dimensional array. Higher dimensional arrays support slicing, e.g.,

Listing 9.34, array2DSlicing.fsx:
Examples of Array2D slicing. Compare with Listing 9.33.

```
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.[2,3]
printfn "%A" arr.[1..,3..]
printfn "%A" arr.[..1,*]
printfn "%A" arr.[1,*]
printfn "%A" arr.[1..1,*]
```

```
11
[[10]
 [11]]
[[0; 3; 6; 9]
 [1; 4; 7; 10]]
[|1; 4; 7; 10|]
[[1; 4; 7; 10]]
```

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

Array2D and higher have a number of built-in functions that will be discussed in Chapter F.

[2] Todo: **Array2D.ToString produces `[[ ... ]]` and not `[|[| ... |]|]`, which can cause confusion.**

Chapter 10

Testing programs

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

· bug

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

· functionality

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

· reliability

Reliability: Does the software work reliably over time? E.g., does the route planning software work

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

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

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

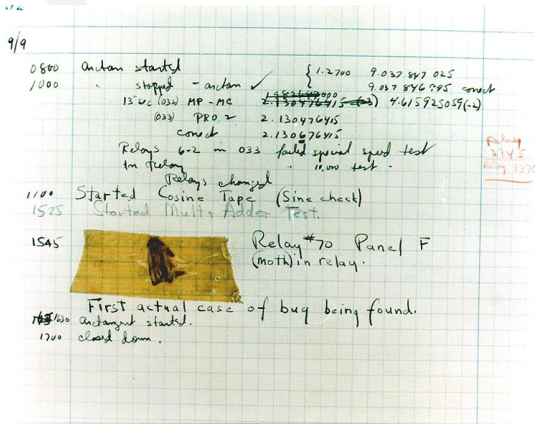

Figure 10.1: The first computer bug caught by Grace Hopper, U.S. Naval Historical Center Online Library Photograph NH 96566-KN.

in case of internet dropouts?

· usability

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

· efficiency

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

· maintainability

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

· portability

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

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

· software testing
· debugging

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

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

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

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

Problem 10.1:

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

Facts about dates in the Gregorian calendar are:

- combinations of dates and weekdays repeat themselves every 400 years;
- the typical length of the months Januar, February, ... follow the knucle rule, i.e., January belongs to the index knuckle, February to the space between the index and the middle finger, and August restarts or starts on the other hand. All knuckle months have 31 days, all spacing months have 30 days except February, which has 29 days on leap years and 28 days all other years.
- A leap year is a multiplum of 4, except if it is also a multiplum of 100 but not of 400.

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

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

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

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

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

10.1 White-box testing

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

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

1. Decide which are the units to test: The program shown in Listing 10.1 has 3 functions, and we will consider these each as a unit, but we might as well just have chosen `date2Day` as a single unit. The important part is that the union of units must cover the whole program text, and

since `date2Day` calls both `januaryFirstDay` and `sum`, designing test cases for the two later is superfluous. However, we may have to do this anyway, when debugging, and we may choose at a later point to use these functions separately, and in both cases we will be able to reuse the testing of the smaller units.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

```
White-box testing of date2Day.fsx
  Unit: januaryFirstDay
    Branch: 0 - true
  Unit: sum
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
  Unit: date2Day
    Branch: 1a - true
    Branch: 1b - true
    Branch: 1c - true
    Branch: 1d - true
```

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

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

10.2 Black-box testing

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

Decide on the interface to use: It is useful to have an agreement with the software developers

about what interface is to be used, e.g., in our case, the software developer has made a function `date2Day d m y`, where `d`, `m`, and `y` are integers specifying the day, month, and year.

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

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

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

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

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

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

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

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

Write a program executing the tests:

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

```
let testCases = [
  ("A complete week",
   [(1, 1, 2016, "Friday");
    (2, 1, 2016, "Saturday");
    (3, 1, 2016, "Sunday");
    (4, 1, 2016, "Monday");
    (5, 1, 2016, "Tuesday");
    (6, 1, 2016, "Wednesday");
    (7, 1, 2016, "Thursday");]);
  ("Across boundaries",
   [(31, 12, 2014, "Wednesday");
    (1, 1, 2015, "Thursday");
    (30, 9, 2017, "Saturday");
    (1, 10, 2017, "Sunday")]);
  ("Across feburary bondary",
   [(28, 2, 2016, "Sunday");
    (29, 2, 2016, "Monday");
    (1, 3, 2016, "Tuesday");
    (28, 2, 2017, "Tuesday");
    (1, 3, 2017, "Wednesday")]);
  ("Leap years",
   [(1, 3, 2015, "Sunday");
    (1, 3, 2012, "Thursday");
    (1, 3, 2000, "Wednesday");
    (1, 3, 2100, "Monday")]);
  ]

printfn "Black-box testing of date2Day.fsx"
for i = 0 to testCases.Length - 1 do
  let (setName, testSet) = testCases.[i]
  printfn "  %d. %s" (i+1) setName
  for j = 0 to testSet.Length - 1 do
    let (d, m, y, expected) = testSet.[j]
    let day = date2Day d m y
    printfn "    test %d - %b" (j+1) (day = expected)
```

Listing 10.5, Output of Listing 10.4.

```
Black-box testing of date2Day.fsx
  1. A complete week
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
    test 6 - true
    test 7 - true
  2. Across boundaries
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
  3. Across feburary bondary
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
    test 5 - true
  4. Leap years
    test 1 - true
    test 2 - true
    test 3 - true
    test 4 - true
```

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

Advice

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

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

10.3 Debugging by tracing

· debugging

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

· mockup code

· tracing

Some development interfaces has built-in tracing system, e.g., `fsharpi` will print inferred types and some binding values. However, often a source of a bug is due to a misunderstanding of the flow of data trough a program execution, and we will in the following introduce *hand tracing* a technique to simulate the execution of a program by hand.

· hand tracing

Consider the program,

Listing 10.6, gcd.fsx:
The greatest common divisor of 2 integers.

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

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

· environment

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

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

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

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

$$E_0 : \\ \text{gcd} \rightarrow \big((a, b), \text{gcd-body}, \varnothing\big) \\ a \rightarrow 10 \\ b \rightarrow 15 \\ \text{line 11: gcd a b} \rightarrow ? \\ E_1 : \big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$$

where the new environment is noted to have gotten its argument names `a` and `b` bound to the values `10` and `15` respectively, and where the return of the function to environment E_0 is yet unknown, so it is

noted as a question mark. In line 2 the comparison `a < b` is checked, and since we are in environment E_1 then this is the same as checking `10 < 15`, which is true so the program executes line 3. Hence, we initiate a new environment E_2 and update our trace as,

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

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

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

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

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

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

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

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

E_2 to 5,

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

and likewise for E_2 and E_1:

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

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

write:

E_0 :
gcd $\rightarrow \big((a, b), \text{gcd-body}, \varnothing\big)$
$a \rightarrow 10$
$b \rightarrow 15$
line 11: gcd a b $\rightarrow$ ↯ 5
line 11: stdout $\rightarrow$ "gcd 10 15 = 5"

~~E_1~~ :$\big((a \rightarrow 10, b \rightarrow 15), \text{gcd-body}, \varnothing\big)$
line 3: gcd b a $\rightarrow$ ↯ 5

~~E_2~~ :$\big((a \rightarrow 15, b \rightarrow 10), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 5
line 5: gcd b (a % b) $\rightarrow$ ↯ 5

~~E_3~~ :$\big((a \rightarrow 10, b \rightarrow 5), \text{gcd-body}, \varnothing\big)$
line 5: a % b $\rightarrow$ 0
line 5: gcd b (a % b) $\rightarrow$ ↯ 5

~~E_4~~ :$\big((a \rightarrow 5, b \rightarrow 0), \text{gcd-body}, \varnothing\big)$

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

F# uses lexical scope, which implies that besides function arguments, we also at times need to consider the environment at place of writing. E.g., for the program

Listing 10.7, lexicalScopeTracing.fsx:
Example of lexical scope and closure environment.

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

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

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

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

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

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

$$
\begin{aligned}
&E_0 : \\
&\quad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope } 2.0 \to ? \\
&E_1 : \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
&\quad a \to 3.0 \\
&\quad \text{f} \to \big(z, \text{a * x}, (a \to 3.0, x \to 2.0)\big) \\
&\quad a \to 4.0 \\
&\quad \text{line 5: f x} \to ? \\
&E_2 : \big(z \to 10.0, \text{a * x}, (a \to 3.0, x \to 2.0\big)
\end{aligned}
$$

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

$$
\begin{aligned}
&E_0 : \\
&\quad \text{testScope} \to \big(x, \text{testScope-body}, \varnothing\big) \\
&\quad \text{line 6: testScope } 2.0 \to \cancel{?}\, 6.0 \\
&\quad \text{line 6: stdout} \to \text{"6.0"} \\
&\cancel{E_1} : \big(x \to 2.0, \text{testScope-body}, \varnothing\big) \\
&\quad a \to 3.0 \\
&\quad \text{f} \to \big(z, \text{a * x}, (a \to 3.0, x \to 2.0)\big) \\
&\quad a \to 4.0 \\
&\quad \text{line 5: f x} \to \cancel{?}\, 6.0 \\
&\cancel{E_2} : \big(z \to 10.0, \text{a * x}, (a \to 3.0, x \to 2.0\big)
\end{aligned}
$$

For mutable bindings, i.e., variables, the scope is dynamic. For this we need the concept of storage, i.e., for the the program

Listing 10.8, 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)
```

We add a storage area to our hand tracing, e.g.,line 6,

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

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

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

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

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

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

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

Hence,

Store :
  $\alpha_1 \rightarrow \cancel{3.0}\ 4.0$
E_0 :
  testScope $\rightarrow (x, \text{testScope-body}, \varnothing)$
  line 6: testScope 2.0 $\rightarrow$?
E_1 : $(x \rightarrow 2.0, \text{testScope-body}, \varnothing)$
  $a \rightarrow \alpha_1$
  f $\rightarrow (z, \text{a * x}, (a \rightarrow \alpha_1, x \rightarrow 2.0))$
  line 5: f x $\rightarrow$?
E_2 : $(z \rightarrow 10.0, \text{a * x}, (a \rightarrow \alpha_1, x \rightarrow 2.0)$

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

Store :
  $\alpha_1 \rightarrow \cancel{3.0}\ 4.0$
E_0 :
  testScope $\rightarrow (x, \text{testScope-body}, \varnothing)$
  line 6: testScope 2.0 $\rightarrow \cancel{?}\ 8.0$
  line 6: stdout $\rightarrow$ "8.0"
$\cancel{E_1}$: $(x \rightarrow 2.0, \text{testScope-body}, \varnothing)$
  $a \rightarrow \alpha_1$
  f $\rightarrow (z, \text{a * x}, (a \rightarrow \alpha_1, x \rightarrow 2.0))$
  line 5: f x $\rightarrow \cancel{?}\ 8.0$
$\cancel{E_2}$: $(z \rightarrow 10.0, \text{a * x}, (a \rightarrow \alpha_1, x \rightarrow 2.0)$

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