Chapter 8

Recursion Revisited

Abstract Recursion is a central concept in functional programming and is used to control flow. A recursive function must obey the 3 laws of recursion:

1. The function must call itself.
2. It must have a base case.
3. The base case must be reached at some point, in order to avoid an infinite loop

In Figure 8.1 is an illustration of the concept of an infinite loop with recursion. In

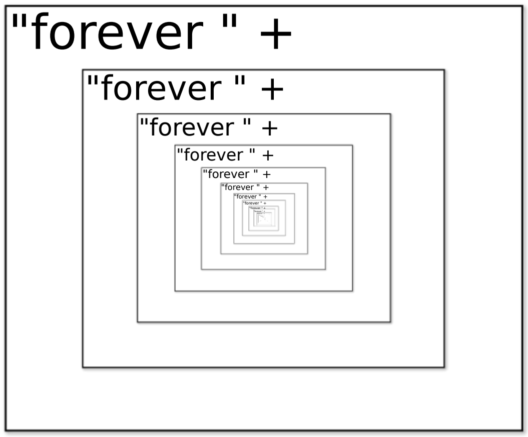


Fig. 8.1 An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "forever " + (forever ()).`

this chapter, you will learn

- How to define recursive functions and types
- About the concept of the Call stack and tail recursion
- How to trace-by-hand recursive functions

8.1 Recursive Functions

A *recursive function* is a function that calls itself, and the syntax for defining recursive functions is an extension of that for regular functions:

Listing 8.1: Syntax for defining one or more mutually dependent recursive functions.

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

From a compiler point of view, the *rec* is necessary, since the function is used before the compiler has completed its analysis. If two functions are mutually recursive, i.e., they call each other, then they must be defined jointly using the *and* keyword.

An example of a recursive function that counts from 1 to 10 is given in Listing 8.2. Here the `count` function calls itself repeatedly, such that the first call is `count 1 10`, which calls `count 2 10`, and so on until the last call `count 11 10`. Each time `count` is called, new bindings named `a` and `b` are made to new values. This is illustrated in Figure 8.2. The old values are no longer accessible, as indicated by subscripts in the figure. E.g., in $\texttt{count}_3$, the scope has access to $\texttt{a}_3$ but not $\texttt{a}_2$ and $\texttt{a}_1$. Thus, in this program, the process is similar to a `while` loop, where the counter is `a`, and in each loop its value is reduced.

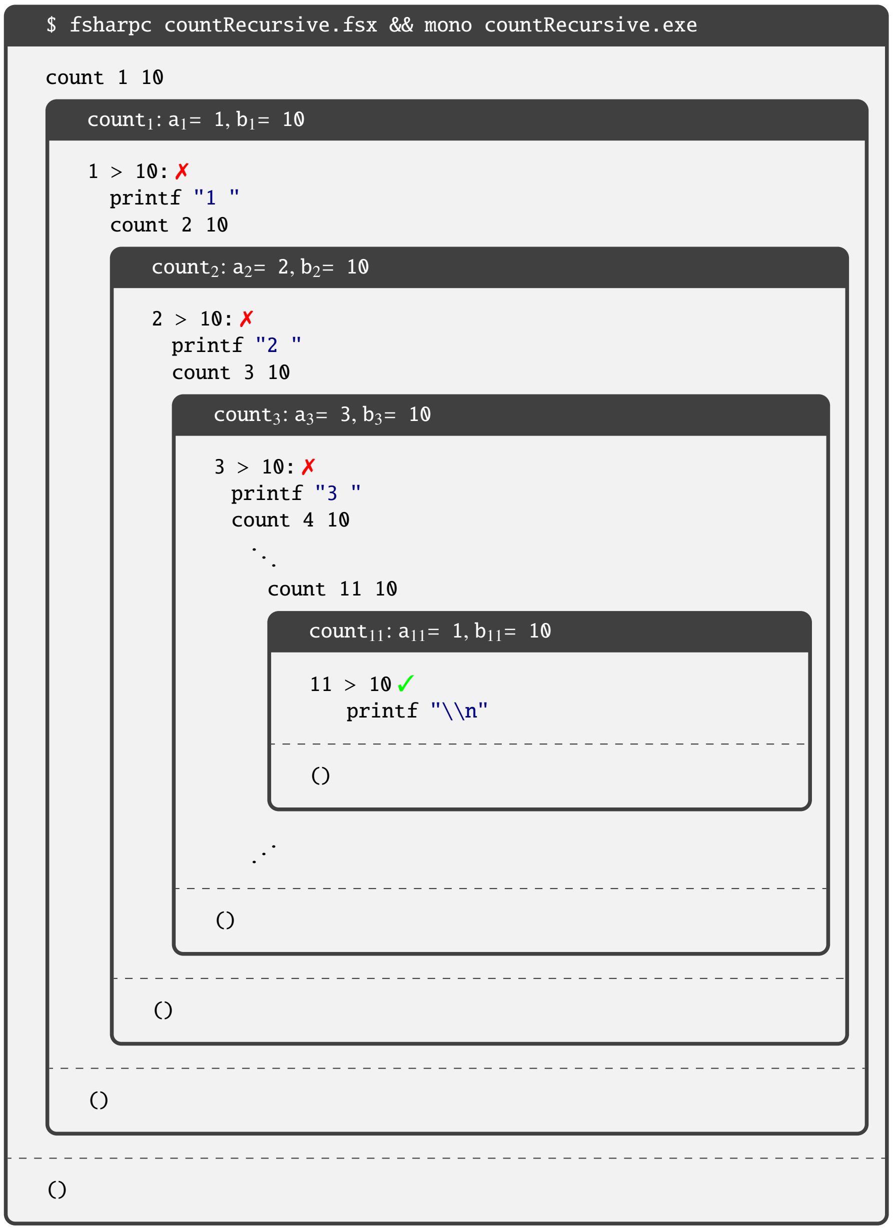
Listing 8.2 countRecursive.fsx:
Counting to 10 using recursion.

```
let rec count a b =
  match a with
    i when i > b ->
      printfn ""
    | _ ->
      printf "%d " a
      count (a + 1) b

count 1 10
```

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

The structure of the function is typical for recursive functions. They very often follow the following pattern.



```
$ fsharpc countRecursive.fsx && mono countRecursive.exe
count 1 10
  count₁: a₁= 1, b₁= 10
  1 > 10: ✗
    printf "1 "
    count 2 10
      count₂: a₂= 2, b₂= 10
      2 > 10: ✗
        printf "2 "
        count 3 10
          count₃: a₃= 3, b₃= 10
          3 > 10: ✗
            printf "3 "
            count 4 10
              ⋱
                count 11 10
                  count₁₁: a₁₁= 1, b₁₁= 10
                  11 > 10 ✓
                      printf "\\n"
                  - - - - - - - - - -
                  ()
              ⋰
          - - - - - - - - - -
          ()
      - - - - - - - - - -
      ()
  - - - - - - - - - -
  ()
- - - - - - - - - -
()
```

Fig. 8.2 Illustration of the recursion used to write the sequence “1 2 3 ... 10” in line 9 in Listing 8.2. Each frame corresponds to a call to `count`, where new values overshadow old ones. All calls return `unit`.

Listing 8.3: Recursive functions consist of a stopping criterium, a stopping expression, and a recursive step.

```
let rec f a =
  match a with
    <stopping pattern> ->
      <stopping step>
    | _ ->
      <recursion step>
```

The `if–then` is also a very common conditional structures. In Listing 8.2, `a > b` is the *stopping condition*, `printfn ""` is the *stopping step*, and `printfn "\%d " a; count (a + 1) b` is the *recursion step*.

A trick to designing recursive algorithms is to **assume that the recursive function already works**. For example, ⋆

```
let rec sum n = match n with 0 -> 0 | _ -> n + sum (n-1)
```

we call `sum` in the recursive step under the assumption, that it correctly sums up values from 0 to $n - 1$. Thus, we are free only to consider how to calculate the sum from 0 to n given that we have n and the result of `sum (n-1)`.

Reconsider the Divide-by-two algorithm in Section 3.7. The algorithm consists of two elements. Given an initial unsigned integer `n`:

1. calculate `n%2u`. This will be the right-most digit in the binary representation of `n`,

2. solve the same problem but now for `n/2u` until `n=0`.

Hence, we have divided the total problem into a series of simple and identical steps. Following the design steps from Section 6.1, we decide to call the function `divideByTwo`, and define it as a mapping from unsigned integers to strings, `divideByTwo (n: uint): string`. From the above, we identify `n=0` as the base case, `divideByTwo (n/2u)` as the recursive call, and `divideByTwo (n/2u) + string (n%2u)` as the recursive step. The resulting algorithm can in brief be written as,

8.2 The Call Stack and Tail Recursion

Recursion is a powerful tool for designing compact and versatile algorithms. However, they can be slow and memory intensive. To understand this, we must understand

Listing 8.4 divideByTwoRecursive.fsx:
Implementing the divide-by-two algorithm using recursion.

```
let rec divideByTwo (n: uint) : string =
  match n with
    0u -> ""
    | _ -> (divideByTwo (n/2u)) + (string (n%2u))

printfn "13u_10 = %A_2" (divideByTwo 13u)
```

```
$ dotnet fsi divideByTwoRecursive.fsx
13u_10 = "1101"_2
```

how function calls are implemented in a typical language using the Call stack, and what can be done to make recursive functions efficient.

Consider Fibonacci's sequence of numbers which is a recursive sequence of numbers with relations to the Golden ratio and structures in biology. The Fibonacci sequence is the sequence of numbers $1, 1, 2, 3, 5, 8, 13, \ldots$. The sequence starts with $1, 1$ and the next number is recursively given as the sum of the two previous ones. A direct implementation of this is given in Listing 8.5. Here we extended the sequence to

Listing 8.5 fibRecursive.fsx:
The n'th Fibonacci number using recursion.

```
let rec fib (n: uint) =
  if n < 2u then n
  else fib (n - 1u) + fib (n - 2u)

printfn "%A" (List.map fib [0u .. 10u])
```

```
$ dotnet fsi fibRecursive.fsx
[0u; 1u; 1u; 2u; 3u; 5u; 8u; 13u; 21u; 34u; 55u]
```

$0, 1, 1, 2, 3, 5, \ldots$ with the starting sequence $0, 1$, allowing us to define a function for all non-negative integers without breaking the definition of the sequence. This ★ is a general piece of advice: **make functions which give sensible output for any argument from its input domain.**

A visualization of the calls and the scopes created by `fibRecursive` is shown in Figure 8.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 8.4 illustrates the tree of calls for `fib 5`. Thus, a call to the function `fib` generates a tree of calls that is five levels deep and has `fib(5)` number of nodes. In general for the program in Listing 8.5, a call to `fib(n)` produces a tree with `fib(n)` $\leq c\alpha^n$ calls to the function for some positive constant c and $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$. Each call takes time and requires memory, and we have thus created a slow and somewhat memory-intensive function.

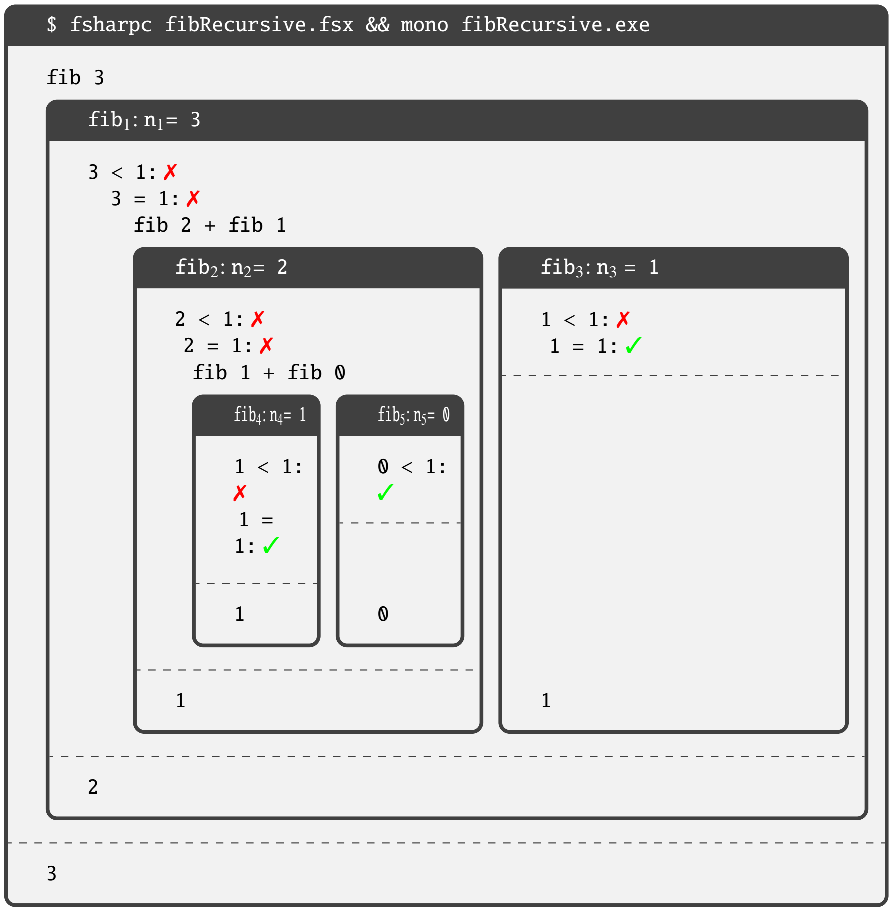


Fig. 8.3 Illustration of the recursion used to write the sequence “1 2 3 ... 10” in line 9 in Listing 8.2. Each frame corresponds to a call to `fib`, where new values overshadow old ones.

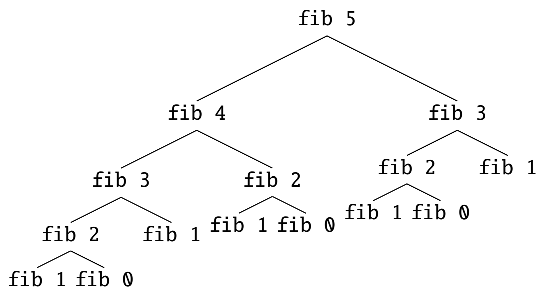


Fig. 8.4 The function calls involved in calling `fib 5`.

This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence since many of the calls are identical. E.g., in Figure 8.4, `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how F# executes function calls.

When a function is called, then memory is dynamically allocated internally for the function on what is known as the *call stack*. Stacks are used for many things in programming, but typically the call stack is considered special since it is almost always implicitly part of any program execution. Hence, it is often just referred to as *The Stack*. When a function is called, a new *stack frame* is stacked (pushed) on the call stack, including its arguments, local storage such as mutable values, and where execution should return to when the function is finished. When the function finishes, the stack frame is unstacked (popped) and in its stead, the return value of the function is stacked. This return value is then unstacked and used by the caller. After unstacking the return value, the call stack is identical to its state prior to the call. Figure 8.5 shows snapshots of the call stack when calling `fib 5` in Listing 8.5. The call first stacks a frame onto the call stack with everything needed to execute the

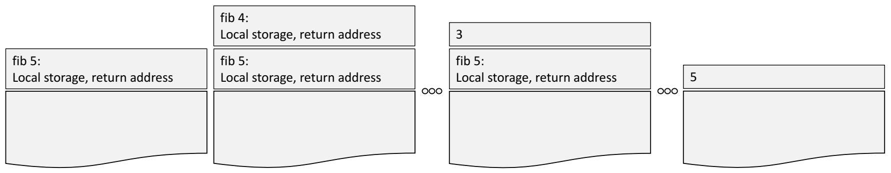

fib 5:
Local storage, return address
fib 4:
Local storage, return address
fib 5:
Local storage, return address
3
fib 5:
Local storage, return address
5

Fig. 8.5 A call to `fib 5` in Listing 8.5 starts a sequence of function calls and stack frames on the call stack.

function body plus a reference to where the return to, when the execution is finished. Then the body of `fib` is executed, which includes calling `fib 4` and `fib 3` in turn. The call to `fib 4` stacks a frame onto the call stack, and its body is executed. Once execution is returned from the call to `fib 4`, the result of the function is on top of the stack. It is unstacked, saved and the call to `fib 3` is treated equally. When the end of `fib 5` is reached, its frame is unstacked, and its result is stacked. In this way, the call stack is returned to its original state except for the result of the function, and execution is returned to the point right after the original call to `fib 5`. Thus, for Listing 8.5 $\mathcal{O}\left(\alpha^n\right),\ \alpha = \frac{1+\sqrt{5}}{2}$ stacking operations are performed for a call to `fib n`. The $\mathcal{O}\left(f(n)\right)$ is the *Landau symbol* used to denote the order of a function, such that if $g(n) = \mathcal{O}(f(n))$ then there exists two real numbers $M > 0$ and a n_0 such that for all $n \geq n_0$, $|g(n)| \leq M|f(n)|$. As indicated by the tree in Figure 8.4, the call tree is at most n high, which corresponds to a maximum of n additional stack frames as compared to the starting point.

The implementation of Fibonacci's sequence in Listing 8.5 can be improved to run faster and use less memory. One such algorithm is given in Listing 8.6 Calculating the 45th Fibonacci number a MacBook Pro, with a 2.9 GHz Intel Core i5 using Listing 8.5 takes about 11.2s while using Listing 8.6 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 8.6 calculates every number in

Listing 8.6 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 8.5.

```
let fib (n: uint) =
  let rec fibPair n pair =
    if n < 2u then pair
    else fibPair (n - 1u) (snd pair, fst pair + snd pair)

  if n < 2u then n
  else fibPair n (0u, 1u) |> snd

printfn "fib(10) = %A" (fib 10u)
```

```
$ dotnet fsi fibRecursiveAlt.fsx
fib(10) = 55u
```

the sequence once and only once by processing the list recursively while maintaining the previous two values needed to calculate the next in the sequence. I.e., the function `fibPair` transforms the pair `(a,b)` to `(b,a+b)` such that, e.g., the 4th and 5th pair `(3,5)` is transformed into the 5th and the 6th pair `(5,8)` in the sequence. What complicates the algorithm is that besides the transformation, we must keep track of when to stop, which here is done using a counter variable, that is recursively reduced by 1 until our stopping criterium.

Listing 8.6 also uses much less memory than Listing 8.5, since its recursive call is the last expression in the function, and since the return value of two recursive calls to `fibPair` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `fibPair` is the return value of the last. This structure is called *tail-recursion*. Compilers can easily optimize the call stack usage for tail recursion, since when in this example `fibPair` calls itself, then its frame is no longer needed, and may be replaced by the new `fibPair` with the slight modification, that the return point should be to `fib` and not the end of the previous `fibPair`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `fibPair` frames, then there is only one, and the return value is equal to the return value of the last call and the return point is to `fib`. Thus, many stack frames in tail recursion are replaced by one. Hence, **prefer tail-recursion whenever possible.** ⋆

8.3 Mutually Recursive Functions

Functions that recursively call each other are called *mutually recursive* functions. F# offers the *`let`* – *`rec`* – *`and`* notation for co-defining mutually recursive functions. As an example, consider the function `even : int -> bool`, which returns true if its argument is even and false otherwise, and the opposite function `odd : int -> bool`. A mutually recursive implementation of these functions can be developed from the following relations: `even 0 = true`, `odd 0 = false`, and for $n > 0$, `even n = odd (n-1)`, which implies that for $n > 0$, `odd n = even (n-1)`: Notice that in the lightweight notation the `and` must

Listing 8.7 mutuallyRecursive.fsx:
Using mutual recursion to implement even and odd functions.

```
let rec even x =
  if x = 0 then true
  else odd (x - 1)
and odd x =
  if x = 0 then false
  else even (x - 1)

let res = List.map (fun i -> (i, even i, odd i)) [1..3]
printfn "(i, even, odd):\n%A" res
```

```
$ dotnet fsi mutuallyRecursive.fsx
(i, even, odd):
[(1, false, true); (2, true, false); (3, false, true)]
```

be on the same indentation level as the original `let`. Without the `and` keyword, F# will issue a compile error at the definition of `even`.

In the example above, we used the `even` and `odd` function problems to demonstrate mutual recursion. There is, of course, a much simpler solution, which does not use recursion at all: A better way to test for parity without recursion. This is to be

Listing 8.8 parity.fsx:
parity

```
let even x = (x % 2 = 0)
let odd x = not (even x)
let res = List.map (fun i -> (i, even i, odd i)) [1..3]
printfn "(i, even, odd):\n%A" res
```

```
$ dotnet fsi parity.fsx
(i, even, odd):
[(1, false, true); (2, true, false); (3, false, true)]
```

preferred anytime as the solution to the problem.

8.4 Recursive types

Data types can themselves be recursive. As an example, the linked list in Figure 7.2. Pattern matching must be used in order to define functions on values of a discriminated union. E.g., in Listing 8.10 we define a function that traverses a list and prints the content of the elements. Discriminated unions are very powerful and can often be used instead of class hierarchies. Class hierarchies are discussed in Section 16.1.

Listing 8.9 discriminatedUnionList.fsx:
A discriminated union modelling for linked lists.

```
type Lst = Ground | Element of int*Lst

let lst = Element (1, Element (2, Element (3, Ground)))
printfn "%A" lst
```

```
$ dotnet fsi discriminatedUnionList.fsx
Element (1, Element (2, Element (3, Ground)))
```

Listing 8.10 discriminatedUnionPatternMatching.fsx:
Traversing a recursive list type with pattern matching.

```
type Lst = Ground | Element of int*Lst
let rec traverse (l : Lst) : string =
    match l with
      Ground -> ""
      | Element(i,Ground) -> string i
      | Element(i,rst) -> string i + ", " + (traverse rst)

let lst = Element (1, Element (2, Element (3, Ground)))
printfn "%A" (traverse lst)
```

```
$ dotnet fsi discriminatedUnionPatternMatching.fsx
"1, 2, 3"
```

8.5 Tracing Recursive Programs

Tracing by hand is a very illustrative method for understanding recursive programs. Consider the recursive program in Listing 8.11. The program includes a function for calculating the greatest common divisor of 2 integers, and calls this function with the numbers 10 and 15. Following the notation introduced in Section 4.5, we write:

Listing 8.11 gcd.fsx:
The greatest common divisor of 2 integers.

```fsharp
 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)
```

```
1 $ fsharpc --nologo gcd.fsx && mono gcd.exe
2 gcd 10 15 = 5
```

Step	Line	Env.	Bindings and evaluations
0	-	E_0	()
1	1	E_0	gcd = $\big((a, b), \text{gcd-body}, ()\big)$
2	9	E_0	$a = 10$
3	10	E_0	$b = 15$

In line 11, `gdc` is called before any output is generated, which initiates a new environment E_1 and executes the code in gcd-body:

Step	Line	Env.	Bindings and evaluations
4	11	E_0	gcd a b = ?
5	1	E_1	$\big((a = 10, b = 15), \text{gcd-body}, ()\big)$

In E_1 we have that $a < b$, which fulfills the first condition in line 2. Hence, we call gdc with switched arguments and once again initiate a new environment,

Step	Line	Env.	Bindings and evaluations
6	2	E_1	a<b = true
7	3	E_1	gcd b a = ?
8	1	E_2	$\big((a = 15, b = 10), \text{gcd-body}, ()\big)$

In E_2, `a < b` in line 2 is false, but `b > 0` in line 4 is true, hence, we first evaluate `a % b`, call `gcd b (a % b)`, and then create a new environment,

Step	Line	Env.	Bindings and evaluations
9	2	E_2	a<b = false
10	4	E_2	b>0 = true
11	5	E_2	a % b = 5
12	5	E_2	gcd b (a % b) = ?
13	1	E_3	$\big((a = 10, b = 5), \text{gcd-body}, ()\big)$

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

Step	Line	Env.	Bindings and evaluations
14	2	E_3	a<b = false
15	4	E_3	b>0 = true
16	5	E_3	a % b = 0
17	5	E_3	gcd b (a % b) = ?
18	1	E_4	$\big((a = 5, b = 0)$, gcd-body, $()\big)$

This time both `a < b` and `b > 0` are false, so we fall through to line 7 and return the value of `a` from E_4, which is `5`:

Step	Line	Env.	Bindings and evaluations
19	2	E_4	a<b = false
20	4	E_4	b>0 = false
21	7	E_4	return = 5

We scratch E_4, return to E_3, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the previously evaluated value,

Step	Line	Env.	Bindings and evaluations
22	5	E_3	gcd b (a % b) = 5
23	5	E_3	return = 5

Like before, we scratch E_3, return to E_2, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the just evaluated value,

Step	Line	Env.	Bindings and evaluations
24	5	E_2	gcd b (a % b) = 5
25	5	E_2	return = 5

Again, we scratch E_2, return to E_1, replace the ?-mark with `5`, and continue the evaluation of line 5. Since this is also a branch of the last statement in `gdc`, we return the just evaluated value,

Step	Line	Env.	Bindings and evaluations
26	3	E_1	gcd a b = 5
27	3	E_1	return = 5

Finally, we scratch E_1, return to E_0, replace the ?-mark with `5`, and continue the evaluation of line 11:

Step	Line	Env.	Bindings and evaluations
28	11	E_0	gcd a b = 5
29	11	E_0	output = "gcd a b = 5"
30	11	E_0	return = ()

Note that the output of `printfn` is a side-effect while its return-value is unit. In any case, since this is the last line in our program, we are done tracing.

8.6 Key Concepts and Terms in This Chapter

In this chapter, we have taken a second look at recursion. You have seen:

- how to define *recursive functions* and *mutually recursive functions*.
- how the *call stack* influences the resources used by recursive functions, and how *tail recursion* is a method for making recursive function efficient.
- *recursive discriminated unions* and how they can be used to model linked lists.
- how to trace-by-hand recursive functions.