ME-2017-10-04

13 Recursion

Recursion is a central concept in F# and used to control flow in loops without the `for` and `while` constructions. Figure 13.1 illustrates the concept of an infinite loop with recursion.

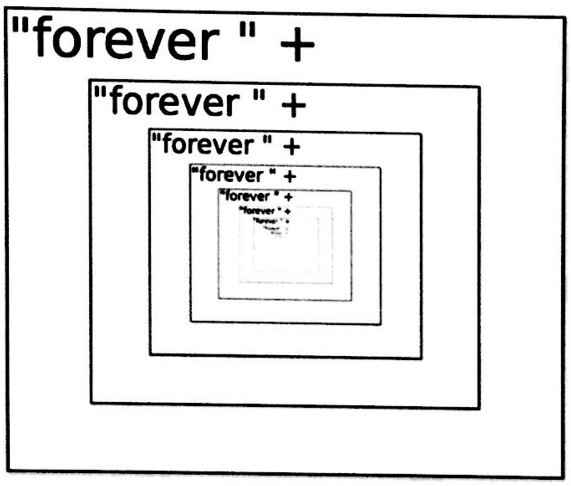


Figure 13.1: An infinitely long string of "forever forever forever...", conceptually calculated by `let rec forever () = "fsharp " + (forever ()).`

13.1 Recursive functions

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

Listing 13.1 Syntax for defining one or more mutually dependent recursive functions.

```
let rec <ident> = <expr> {and <ident> = <expr>} [in] <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, then they must be defined jointly using the *and* keyword. · rec · and

An example of a recursive function that counts from 1 to 10 similarly to Listing 8.5 is given in Listing 13.2.

**Listing 13.2 countRecursive.fsx:
Counting to 10 using recursion.**

```
let rec prt a b =
  if a > b then
    printf "\n"
  else
    printf "%d " a
    prt (a + 1) b

prt 1 10
```

```
$ fsharpc --nologo countRecursive.fsx && mono countRecursive.exe
1 2 3 4 5 6 7 8 9 10
```

Here the prt function calls itself repeatedly, such that the first call is `prt 1 10`, which calls `prt 2 10`, and so on until the last call `prt 11 10`. Each time `prt` is called, new bindings named `a` and `b` are made to new values. This is illustrated in Figure 13.2. The old values are no longer accessible as indicated by subscript in the figure. E.g., in $\texttt{prt}_3$ the scope has access to $\texttt{a}_3$ but not $\texttt{a}_2$ and $\texttt{a}_1$. Thus, in this program, process is similar to a `for` loop, where the counter is `a` and in each loop its value is reduced.

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

Listing 13.3 Recursive functions consists of a stopping criterium, a stopping expression, and a recursive step.

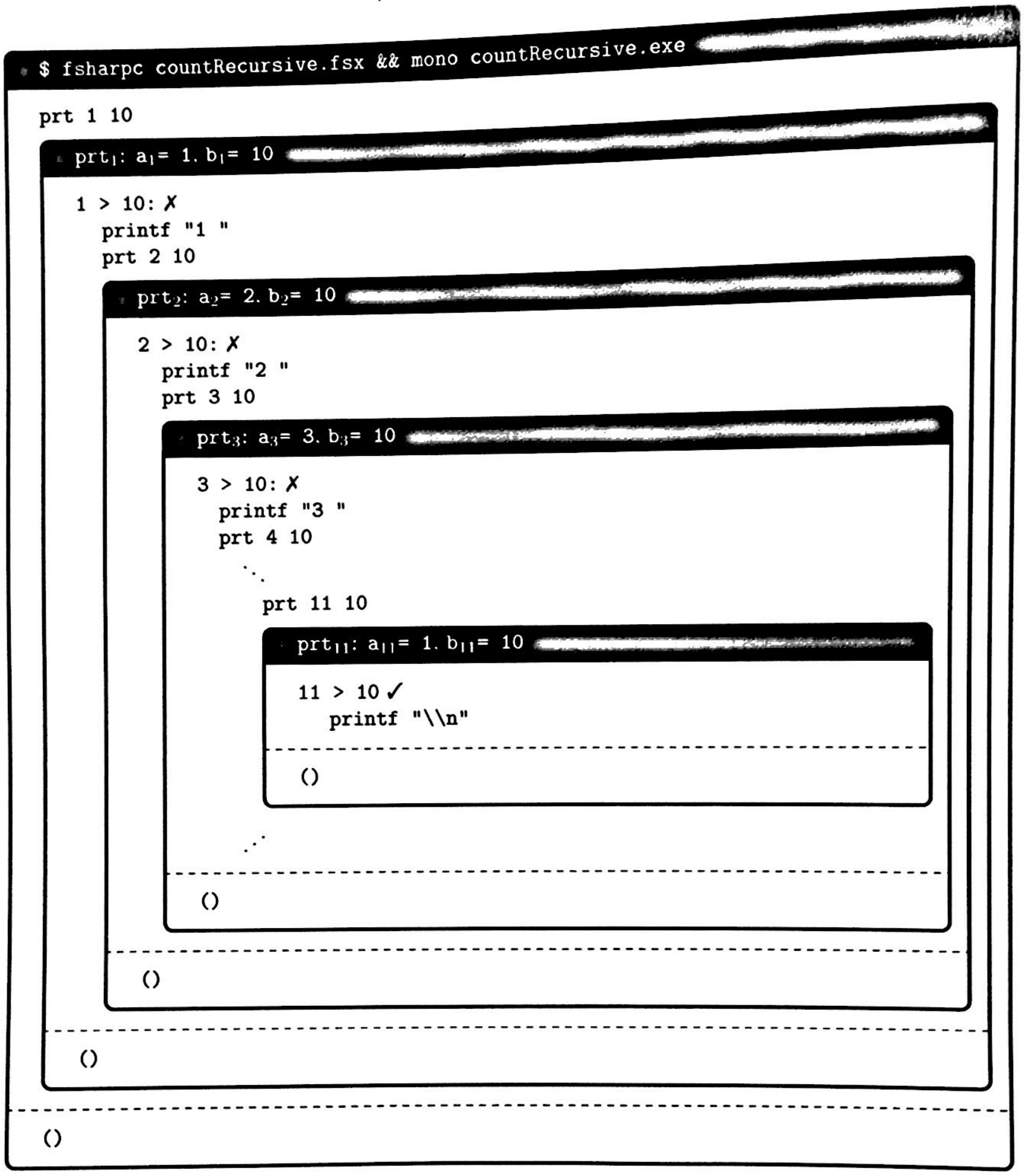
```
let rec f a =
  if <stopping condition>
  then <stopping step>
  else <recursion step>
```

The `match ... with` are also very common conditional structures. In Listing 13.2 `a > b` is the *stopping condition*, `printfn "\n"` is *stopping step*, and `printfn "\%d " a; prt (a + 1) b` is the *recursion step*.

· stopping conditi
· stopping step
· recursion step

13.2 The call stack and tail recursion

Fibonacci's sequence of numbers is a recursive sequence of numbers with relations to the Golden ratio and structures in biology. Fibonacci's 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. A direct implementation of this is given in Listing 8.7.



```
$ fsharpc countRecursive.fsx && mono countRecursive.exe

prt 1 10
  prt_1: a_1= 1, b_1= 10
    1 > 10: ✗
      printf "1 "
      prt 2 10
      prt_2: a_2= 2, b_2= 10
        2 > 10: ✗
          printf "2 "
          prt 3 10
          prt_3: a_3= 3, b_3= 10
            3 > 10: ✗
              printf "3 "
              prt 4 10
                ⋱
                  prt 11 10
                  prt_11: a_11= 1, b_11= 10
                    11 > 10 ✓
                      printf "\\n"
                    ----------------
                    ()
                ⋰
            ----------------
            ()
        ----------------
        ()
    ----------------
    ()
----------------
()
```

Figure 13.2: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `prt`, where new values overshadow old. All return `unit`.

Listing 13.4 fibRecursive.fsx:
The n'th Fibonacci number using recursive.

```
let rec fib n =
  if n < 1 then
    0
  elif n = 1 then
    1
  else
    fib (n - 1) + fib (n - 2)

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

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

Here we extended the sequence to $0, 1, 1, 2, 3, 5, \ldots$ and starting sequence $0, 1$ allowing us to define all $\text{fib}(n) = 0$, $n < 1$ and $\text{fib}(n) = \text{fib}(n-2) + \text{fib}(n-1)$, $n > 1$. Thus, our function is defined for all integers, and the irrelevant negative arguments fails gracefully by returning 0. This is a general advice: **make functions that fails gracefully.**

what if n=0?

Advice No! *

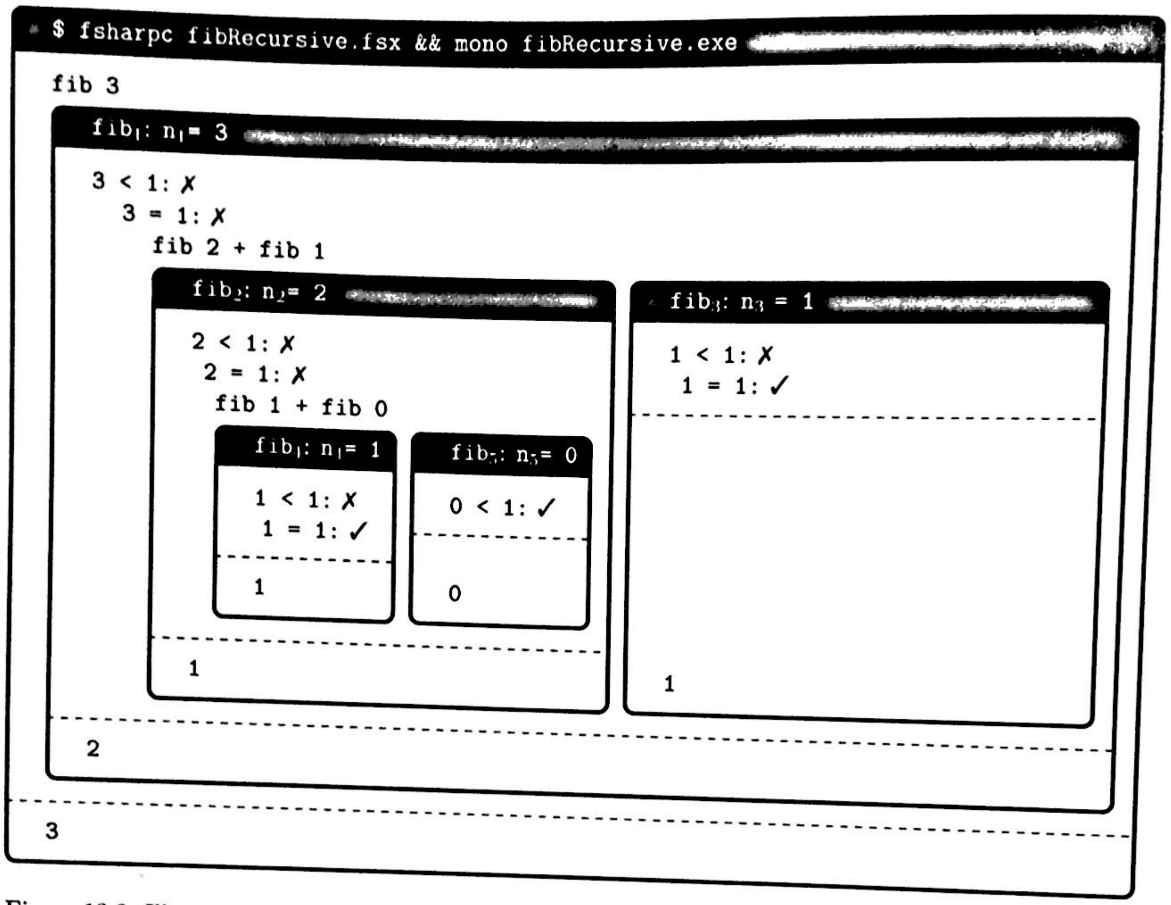
A visualization of the calls and the scopes created by `fibRecursive` ~~is~~ are shown in Figure 13.3. The figure illustrates that each recursive step results in two calls to the function, thus creating two new scopes. And it gets worse. Figure 13.4 illustrates the tree of calls for `fib 5`. Thus a call 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 13.4, 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 $\alpha \geq \frac{1+\sqrt{5}}{2} \sim 1.6$[1]. Each call takes time and requires memory, and we have thus created a slow and somewhat memory intensive function. This is a hugely ineffective implementation of calculating entries into Fibonacci's sequence, since many of the calls are identical. E.g., in Figure 13.4 `fib 1` is called five times. Before we examine a faster algorithm, we first need to discuss how ~~the compiler implements~~ 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, ~~then~~ a new *stack frame* is stacked ~~on~~ onto (pushed) the call stack including its arguments, local storage such as mutable ~~values~~ data, 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, ~~then~~ the call stack is identical to its state prior to the call. ~~In~~ Figure 13.5 ~~is~~ shows snapshots of the call stack, when calling `fib 5` in Listing 13.4. The call first stacks a frame onto the call stack with everything needed to execute and return ~~to~~ right after

· call stack
· The Stack
· stack frame

[1]Jon: https://math.stackexchange.com/questions/674533/prove-upper-bound-big-o-for-fibonaccis-sequence

* I once heard someone argue for a "safe" version of division – not acceptable. In the fib(n) case, the solution may be ok...



```
$ fsharpc fibRecursive.fsx && mono fibRecursive.exe
```

fib 3
$\texttt{fib}_1$: n_1= 3
3 < 1: ✗
3 = 1: ✗
fib 2 + fib 1
$\texttt{fib}_2$: n_2= 2
2 < 1: ✗
2 = 1: ✗
fib 1 + fib 0
$\texttt{fib}_1$: n_1= 1
1 < 1: ✗
1 = 1: ✓
1
$\texttt{fib}_5$: n_5= 0
0 < 1: ✓
0
1
$\texttt{fib}_3$: n_3 = 1
1 < 1: ✗
1 = 1: ✓
1
2
3

Figure 13.3: Illustration of the recursion used to write the sequence "1 2 3 ... 10" in line 8 in Listing 13.2. Each frame corresponds to a call to `fib`, where new values overshadow old.

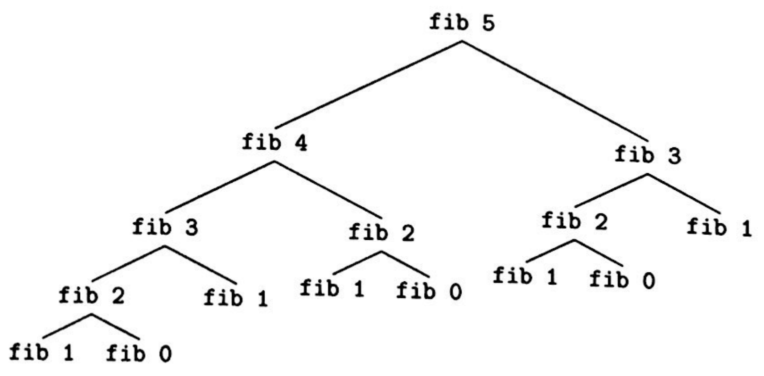


Figure 13.4: The function calls involved in calling `fib 5`.

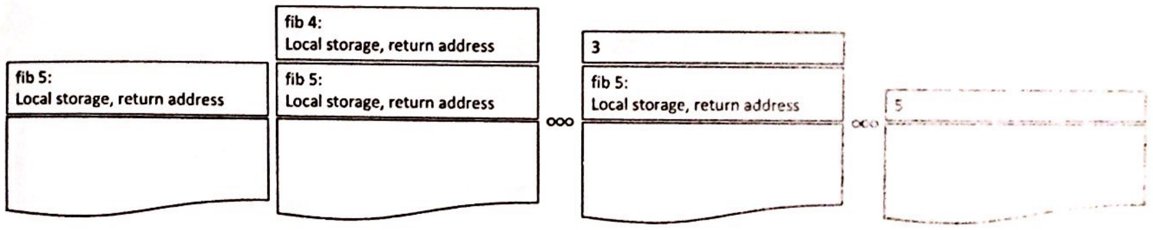

fib 5:
Local storage, return address
fib 4:
Local storage, return address
fib 5:
Local storage, return address
3
fib 5:
Local storage, return address
5

Figure 13.5: A call to `fib 5` in Listing 13.4 starts a sequence of function calls and stack frames on the call stack.

the point of calling. 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`, ~~then~~ 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, ~~then~~ 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 13.4, $\mathcal{O}(\alpha^n)$, $\alpha = \frac{1+\sqrt{5}}{2}$ stacking operations ~~is~~ are performed for a call to `fib n`. The $\mathcal{O}(f(n))$ 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, $|g(n)| \leq M|f(n)|$, $n \geq n_0$. As indicated by the tree in Figure 13.4, the call tree is maximally n high, which corresponds to a maximum of n additional stack frames as compared to the starting point. , for all $n \geq n_0$,

· Landau symbol

The implementation of Fibonacci's sequence in Listing 13.4 can be improved to run faster and use less memory. One such algorithm is given in Listing 13.5

Listing 13.5 fibRecursiveAlt.fsx:
A fast, recursive implementation of Fibonacci's numbers. Compare with Listing 13.4.

```
let fib n =
  let rec fibPair n pair =
    if n < 2 then pair
    else fibPair (n - 1) (snd pair, fst pair + snd pair)
  if n < 1 then 0
  elif n = 1 then 1
  else fibPair n (0, 1) |> snd

printfn "fib(10) = %d" (fib 10)
```

```
$ fsharpc --nologo fibRecursiveAlt.fsx && mono fibRecursiveAlt.exe
fib(10) = 55
```

Calculating the 45th Fibonacci number a Macbook Pro, with a 2.9 Ghz Intel Core i5 using Listing 13.4 takes about 11.2s, while using Listing 13.5 is about 224 times faster and only takes 0.050s. The reason is that `fib` in Listing 13.5 calculates every number in 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 `helper` 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 13.5 also uses much less memory than Listing 13.4, since its recursive call is the last expression in the function, and since the return value of two recursive calls to `helper` is the same as the return value of the last. In fact, the return value of any number of recursive calls to `helper` 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 `helper` calls itself, then its frame is no longer needed, and may be replaced by the new `helper` with the slight modification, that the return point should be to `fib` and not the end of the previous `helper`. Once the recursion reaches the stopping criteria, then instead of popping a long list of calls of `helper` 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 is replaced by one. Hence, it is a wise choice to **prefer tail-recursion whenever possible.**

· tail-recursion

Advice

13.3 Mutual 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 statements: `even 0 = true`, `odd 0 = false`, and `even n = odd (n-1)`:

· mutually recurs

what about odd(n), n>0 ?

Listing 13.6 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 w = 5;
printfn "%*s %*s %*s" w "i" w "even" w "odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursive.fsx && mono mutuallyRecursive.exe
    i  even   odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

Notice that in the lightweight notation the and must be on the same indentation level as the original `let`.

Without the and keyword, F# will ~~return an error~~ issue a compile error at the definition of even. However, it is possible to implement mutual recursion by using functions as an argument, e.g.,

Listing 13.7 mutuallyRecursiveAlt.fsx:
Mutual recursion without the keyword needs a helper function.

```
let ~~rec~~ evenHelper (notEven: int -> bool) x =
  if x = 0 then true
  else notEven (x - 1)

let rec odd x =
  if x = 0 then false
  else evenHelper odd (x - 1);;

let even x = evenHelper odd x

let w = 5;
printfn "%*s %*s %*s" w "i" w "Even" w "Odd"
for i = 1 to w do
  printfn "%*d %*b %*b" w i w (even i) w (odd i)
```

```
$ fsharpc --nologo mutuallyRecursiveAlt.fsx
$ mono mutuallyRecursiveAlt.exe
    i  Even   Odd
    1 false  true
    2  true false
    3 false  true
    4  true false
    5 false  true
```

But, Listing 13.6 is clearly to be preferred over Listing 13.7.

In the 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:

Listing 13.8 parity.fsx:
A better way to test for parity without recursion.

```
let even x = (x % 2 = 0)
let odd x = not (even x)
```

which is to be preferred anytime as the solution to the problem.

13.4 To Do

- Add an intermezzo, giving examples of how to write a recursive program by thinking the problem has been solved.