**[Checklist for Programming with Recursion](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8949" \o "Checklist for Programming with Recursion)**

**by Geoffrey Smith**

Writing recursive programs is quite different from writing imperative programs. Here we present a *checklist* that gives a systematic way to *derive* a correct recursive function.

A recursive function f will always work by *case analysis* on the input. On some inputs, f will compute the answer directly, without calling itself recursively---these are called *base cases*. On other inputs, f will call itself recursively to help it to compute the answer---these are called *non-base cases*. (Obviously, there must be at least one base case, because if f *always* calls itself recursively, it will never halt.) Your function definition must include enough cases to cover all possible inputs; pleasantly, F# automatically checks this for you if you use *pattern matching*.

To develop your cases systematically, it helps to keep in mind the following [*Checklist for Programming with Recursion*](https://moodle.cis.fiu.edu/v2.1/mod/page/view.php?id=8949):

1. Make sure that each base case returns the correct answer.
2. Make sure that each non-base case returns the correct answer, *assuming that each of its recursive calls returns the correct answer*.
3. Make sure that each recursive call is on a *smaller* input.

**An Example Derivation Using the Checklist**

Suppose that we wish to define an F# function that takes as input a pair of lists and appends them:

> append ([1;3;5],[2;4;6]);;

val it : int list = [1; 3; 5; 2; 4; 6]

We need to develop base and non-base cases. Under Step 1, each base case that we include needs to return the correct answer. Of course, Step 1 doesn't tell us *how* to do this, so we need to identify cases where the answer is easy to compute.

For append, one obvious base case is the case when the first list is empty; in this case, we should just return the second list. Using F# pattern matching, we can implement this by

let rec append = function

| ([], ys) -> ys

The case when the second list is empty could also be included as a base case, but let's hold off on that right now. (We would like to find a solution with as few cases as necessary!)

Turning to non-base cases, Step 2 of the checklist is more interesting. It says that when we handle a case using recursion, we can *assume* that the recursive calls that we make return the correct results! Being able to assume this may seem like magic to you---it is extremely powerful. It means that all that we need to do is to write code that *transforms* the partial solution returned by the recursive call into the complete solution for this case.

For append, consider the case when the first list is nonempty, with head x and tail xs, and the second list is ys. In this case, we can recursively call append (xs,ys) and we can *assume* that this returns the correct result. Now we can see that this result is almost the answer that we are trying to compute---all that we need to do is to cons x to the front of the partial solution, and we are done! In F# we can code this like this:

| (x::xs, ys) -> x :: append (xs,ys)

To address Step 3 of the checklist, we must verify that each recursive call gets an input that is *smaller* than the original input. (The precise meaning of *smaller* can vary; in the case of lists, we usually mean *shorter*.) Here notice that the list xs used in the recursive call is shorter by 1 than the original input x::xs, so Step 3 is satisfied. (More precisely, we call append recursively on the *pair* (xs,ys); this pair is smaller than the original input pair (x::xs, ys) because its first list is shorter by 1 and its second list is of equal size.)

Finally, notice that the two cases we have defined for append cover all possible inputs: the first list can be either empty or not, and the second list can be arbitrary. Hence we do not need any more cases, and our definition is complete.

**Using Concrete Examples in Step 2**

In carrying out Step 2 of the checklist, it is often helpful to think about a concrete example. In designing append, for instance, we could think about the example

append([1;3;5],[2;4;6]).

One possible recursive call we could make is

append([3;5],[2;4;6])

and, under step 2 of the checklist, we can assume that this call returns the correct answer, namely [3;5;2;4;6]. Since the final answer that we want in this case is[1;3;5;2;4;6], it is easy to see that all that we need to do is to cons 1 (the head of the first list) with the result returned by the recursive call.

Notice that if instead we had considered the recursive call

append([1;3;5],[4;6])

then we would have gotten back [1;3;5;4;6], which we cannot easily transform into [1;3;5;2;4;6].

**Caution**: In working with such concrete examples, we must make sure that the transformation code that we write works in general, and not just on the specific example that we are considering.

**"Smaller" Inputs in Step 3**

The purpose of Step 3 is to ensure that we cannot have an infinite recursion. As a result, we can define *smaller* to be *any* measure of the "size" of the input that satisfies the following crucial property: *it must not be possible for values to keep getting smaller forever*! Notice that this property is satisfied on lists if we define *smaller* to mean *shorter*. After all, a list of length *k* can be made shorter at most *k* times before reaching an empty list, which cannot be made shorter. In contrast, this property is *not* satisfied on rational numbers if we define *smaller* to mean the usual *less than*. The problem is that we can have an infinite descending chain:

1 > 1/2 > 1/4 > 1/8 > 1/16 > 1/32 > 1/64 > ...

In practice, Step 3 is usually routine (as when a function calls itself recursively on the tail of its input list), but sometimes it can be subtle.

**Remark**: Formally, Step 3 requires that every recursive call is on a value that is smaller than the original input with respect to some *well-founded partial order* ≤.

Recall that a *partial order* ≤ is a binary relation on a set *S* that is

* *reflexive*: for all *a*, *a*≤*a*
* *transitive*: for all *a*, *b*, and *c*, if *a*≤*b* and *b*≤*c*, then *a*≤*c*
* *antisymmetric*: for all *a* and *b*, if *a*≤*b* and *b*≤*a*, then *a*=*b*.

And ≤ is *well-founded* if every nonempty subset of *S* has a *minimal* element, that is, an element with nothing smaller than it.

**Correctness of the Checklist**

Why does the checklist work? The answer is that the checklist is really an informal proof by *mathematical induction* that your program works correctly. In particular, the assumption (in Step 2) that each recursive call works correctly is exactly the *induction hypothesis* in the induction step of a proof by induction.

Here's a more intuitive way to see why the checklist works. Suppose that the implementation of a function f satisfies the three steps of the checklist. We argue by contradiction that f must work correctly. For suppose that f works incorrectly on at least one input. In that case, we can choose *e* to be an input of *minimum size* on which fworks incorrectly. Now, *e* cannot be a base case, or else Step 1 would fail. So *e* must be a non-base case. But when f makes recursive calls in processing *e*, those calls must be on inputs *smaller* than *e*, by Step 3. Hence, by the minimality of *e*, f must work correctly on those recursive calls. But then, by Step 2, f must in fact work correctly on *e*!

**Analyzing Code with the Checklist**

I will sometimes ask you to *analyze* an F# function definition with respect to the checklist. This involves answering three questions:

1. Is there any circumstance in which a base case fails to return the correct result for the input?
2. Is there any circumstance in which the code for a non-base case can fail to transform correct results returned by the recursive calls into the correct result for the input?
3. Is there any circumstance in which the definition can make a recursive call on an input that is not smaller than the original input?

Let's consider some examples. First consider a function to calculate the factorial of a non-negative integer:

let rec fact = function

| 0 -> 0

| n -> n \* fact(n-1)

If you try this code, you will see that wrongly returns 0 for every input. Let's analyze it using the checklist. Step 1 *fails*, since 0! is by definition 1, not 0. Step 2, however, is fine---if n is positive, then by definition n! is n\*(n-1)!. So, assuming that the recursive call returns the correct answer, the given non-base case code returns the correct answer. (Note that because of the faulty base case, the recursive call *won't* actually return the correct answer. But that doesn't matter---Step 2 is still satisfied. This might seem nonsensical to you, but notice that to *debug* the broken definition, it suffices to correct the base case; no change to the non-base case is needed.) Step 3 is also fine, since the recursive call is on n-1, which is smaller than n.

Next consider a function that is supposed to sort a list of integers into nondecreasing order:

let rec sort = function

| [] -> []

| [n] -> [n]

| n::ns -> n :: sort ns

Step 1 is fine, because lists of length 0 or 1 are automatically sorted. Step 2 fails, because the value returned, n :: sort ns, is not necessary correctly sorted even assuming that sort ns returns the correct answer. The reason, of course, is that we have not established that n is less than or equal to the elements of ns. And, indeed, it is easy to come up with a counterexample: on sort [2;1], the recursive call sort [1] correctly returns [1], but this is incorrectly transformed to [2;1], which is not in order. Step 3 is fine, since ns is shorter than n::ns.

As a final (somewhat silly) example, suppose we want a function f to compute something. Consider the definition

let rec f x = f x

This definition doesn't include any base cases. So it *automatically* satisfies Step 1, because there is no base case that ever returns a wrong answer! (A key point to grasp is that Step 1 is **not** concerned with whether the definition includes "enough" base cases, but only with the correctness of the base cases that *are* included.) The definition also trivially satisfies Step 2, since its non-base case code certainly returns the correct answer, assuming that the recursive call returns the correct answer. But, of course, it fails Step 3, since the recursive call is on the original input, rather than on a smaller value. Of course, this code just goes into an infinite loop. (By the way, it's interesting to look at the *type* that F# infers for f.)

We'll discuss more interesting examples of using the Checklist later; I hope you will find it useful in your programming.