# CS3110 Prelim 1, Question 5d

We're given the following code, and want to fill in the ???:

```
let map lst f = (List.fold_left (fun acc a bs -> ???) (fun x -> x) lst) []
```

Our first step in solving any "fill in the blank" question should be figure out the type of ???. But wait—folding function seems to have 3 arguments. Shouldn't it only have 2?

Remember that because of currying, we can think of any function that takes three arguments as a function that takes 2 arguments and returns a function. We could have written the fold function instead as:

```
(fun acc a ->
  fun bs -> ???)
```

This is more easily recognizable as a thing which takes in two arguments (an accumulator and an element a of the list `lst`) and returns a function `fun bs -> ???`.

What do we know about the types of fold? We know whatever function we take to fold, it has to take in an accumulator and an element of the list, and return another thing of the same type as the accumulator. Therefore, we know that the type of `fun bs -> ???` is the type of the accumulator.

Great! The other thing we know about fold is that the base of the fold (i.e., `fun x -> x`) is also the same type as the accumulator. So, we know that the accumulator is a function type, and the output type is the same as its input type (it has type `'c -> 'c` for some type `'c`).

Finally, the result type of `List.fold_left` is the same type as the accumulator. Therefore, the type of

```
(List.fold_left (fun acc a bs -> ???) (fun x -> x) lst)
```

is the same as the accumulator: `'c -> 'c`. We then apply that function to `[]`, and the end result is supposed to match the output type of map.

Since map is supposed to give us a `'b list`, we know that the type of the accumulator has to be `'b list -> 'b list`.

**To summarize: the type of the accumulator has to be** `'b list -> 'b list`

In particular, that means that `bs` has to be of type `'b list` (that's why we called it `bs`), and also the type of ??? has to be `'b list`. In fact, if you got this far and figured out that ??? should have type `'b list`, then you should have gotten 4/8 points.

Now, how do we go about actually filling in ??? ? Let's think about what fold actually does. If we're sitting at the $k$th element of the list, then `acc` is the result of applying the fold function to the first $k-1$ elements of the list.

However, remember that we're building a list. And so, if we're building a list by using ::, we can't tack on the $k$th element until we have the result of mapping the tail of the list. So, somehow we need to get a result we haven't computed yet — although, once we have it, we know what to do with it: cons `(f a)` onto the front of it, and return it.

This is exactly why our accumulator is a function: the argument `bs` is going to be the result of computing map on the rest of the list (and which we are going to compute in the future) and once we have that, we can continue with the computation.

**This gives us the following invariant for the accumulator: if we're currently at the $k$th element of the list, `acc` is a function which, when you give it the result of properly computing map on the $k \dots n$ elements of the list, gives you the correct answer for mapping the *entire* list.**

So, let's fill out ???. We know that we're sitting at the $k$th element of the list. By our invariant, if we're given an argument `bs`, and we can assume that `bs` is the **correct result for mapping the rest of the list**. So, to extend this to the correct result for the $k$th element and greater, we just have to stick `(f a)` to the front of `bs`.

Finally, (using the invariant) if we give `acc` the result for the $k$th element and greater, it will actually return the correct result for the entire list. So, the correct value of ??? is

```
acc ((f a)::bs)
```

Overall, the result of the fold function (which gets turned into the accumulator at the $k+1$th element) is a function which, if you hand it the correct result for the $k+1$ and greater part of the list, returns the correct result for the entire list. This is the invariant again, just now for $k+1$!

Note how we used the invariant for the accumulator at the $k$th element to prove the invariant at the $k+1$ element. This is just plain old induction!

We have to make sure the base case works: if we haven't seen any elements yet (which is the usual base case for fold) then we need a function which, given the correct answer for the entire list, returns the correct answer for the entire list. Obviously, `fun x -> x` does exactly that, so the base case works.