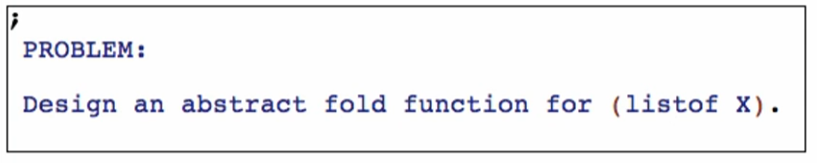
Abstract functions can be produced directly from templates. This can be wonderfully useful, especially for types involving mutual reference.

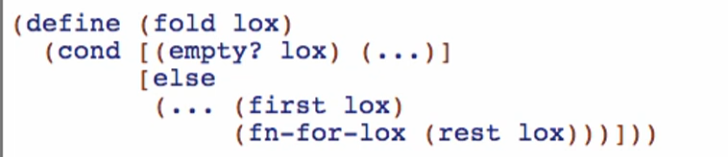


Fold function

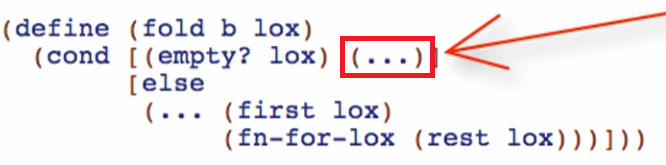
* Abstract function based directly on the template or templates in the case of mutual reference

**Working backwards**

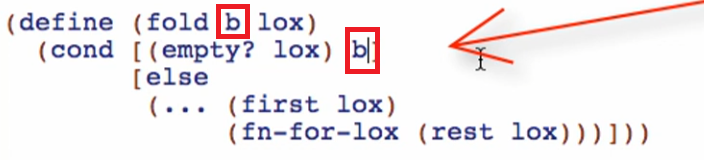
Copy template and rename the function fold



Work for the first (…)

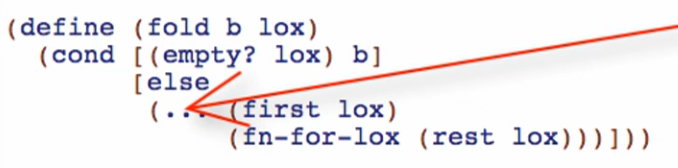


Notice that this is a base case result, so we can use “b” as a parameter for fold and replace (…) with b

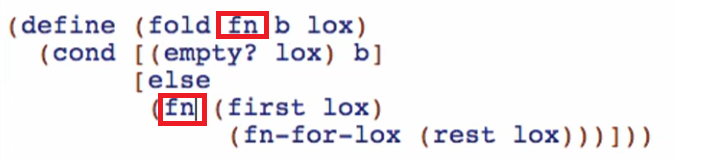


Because b is the final result not a function of no args

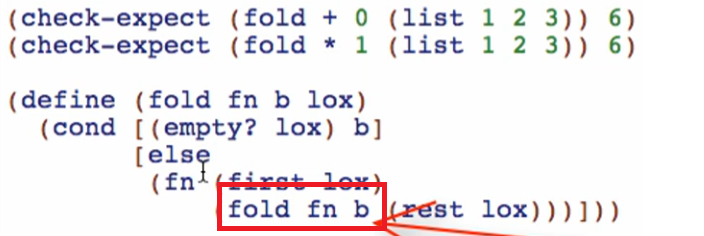
Work for the second (…)



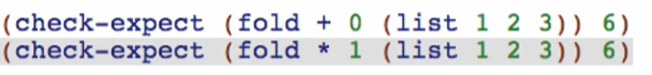
Notice that this is the combination function, the one that combines the first to the result of natural recursion of rest. We can call it “fn” and replace that (…) with “fn” parameter



Note: don’t forget to rename the natural recursion and add the additional parameters to the natural recursion



Examples



Check if well-formed and passing

Purpose



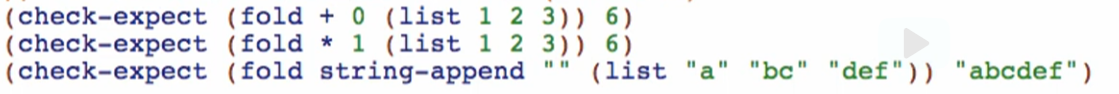
* An abstract function based directly off the template, where each … is replaced by a parameter not a function of no arguments

Signature

lox is of course (listof X)



Checking the produced value

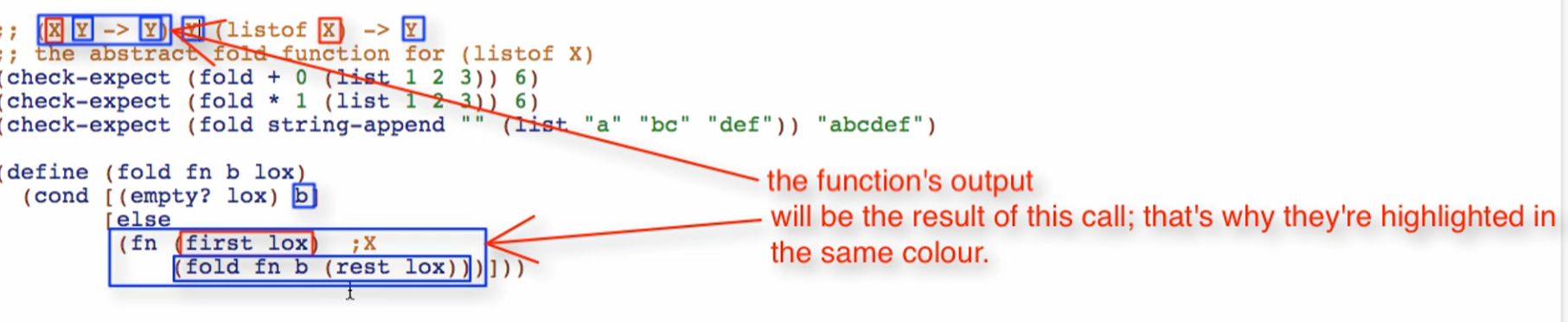


Add another check-expect if we can produce not only numbers

For now, we can say that the type of value produced is the same as the type of values in the list



Try adding another check-expect if we can produce another type that it isn’t the same type as the elements on the list param. OR you can look at the code:



This is like the reduce Function in Java and JS. The return type (Y) is based on the accumulator/base in this case “b”, then the function should also return that same return type (Y). The function should consume (prevValue + currValue) where prevValue is a type of an element of the list, and currValue is the type of the accumulator or base.