Question 1 [15 points] This is a question on higher-order function. Define a function `repeat` of type
\[
\text{val repeat : f:('a -> 'a) -> n:int -> ('a -> 'a)}
\]
This function takes a function \( f \) as argument and then takes a non-negative integer \( n \) and \textit{returns a function}. The returned function will apply the original function \( f \), \( n \) times. Here is a short transcript.

\[
\begin{align*}
\text{> let inc n = n + 1;;} \\
\text{val inc : n:int -> int} \\
\text{> let plusfive = repeat inc 5;;} \\
\text{val plusfive : (int -> int)} \\
\text{> plusfive 3;;} \\
\text{val it : int = 8} \\
\text{> let never = repeat inc 0;;} \\
\text{val never : (int -> int)} \\
\text{> never 3;;} \\
\text{val it : int = 3}
\end{align*}
\]

Solution:

\[
\text{let rec repeat f n =} \\
\quad \text{if (n = 0) then fun x -> x} \\
\quad \text{else fun x -> f ((repeat f (n - 1)) x)}
\]

I explicitly wrote an example showing the zero case so we penalized people who wrote things like \( f \) as what is returned for \( n = 0 \). The bracketing is important. Look at the following

\[
\text{let rec wrong_repeat f n =} \\
\quad \text{if (n = 0) then fun x -> x} \\
\quad \text{else fun x -> f (repeat f (n - 1)) x}
\]

\text{mid_sols.fs(7,27): error FS0001: Type mismatch. Expecting a} \\
\text{'a -> 'a}  \\
\text{but given a} \\
\text{('a -> 'a) -> 'b -> 'c}  \\
\text{The resulting type would be infinite when unifying ''a' and ''a -> 'a'}

We did not penalize you harshly for this. Wrapping the function returned in a \textit{fun} is \textit{very} important.

\[
\text{let rec wrong_repeat2 f n =} \\
\quad \text{if (n = 0) then fun x -> x}
\]
else f (repeat f (n - 1))

Type mismatch. Expecting a

'a -> 'a

but given a

('a -> 'a) -> 'b -> 'b

The resulting type would be infinite when unifying ''a' and ''a -> 'a'

This was penalized. It is not just a question of “syntax” of F#. It is a genuine misunderstanding about higher-order functions.

Question 2 [20 points]

A bit is either the value 0 or 1. A bit-sequence is a list of zeros and ones. I would like to define a function bits of type

val bits : n:int -> int list list

This function, when given n, creates a list of all possible bit-sequences of length n.

Solution

let rec bits n =
    if (n = 0) then [[]]
    elif (n = 1) then [[0];[1]]
    else
        let temp = bits (n - 1)
        (List.map (fun x -> 0 :: x) temp) @ (List.map (fun x -> 1 :: x) temp)

There are, of course, many ways of writing this. Some of you used fold, a bit of overkill for this simple case but fine if you did it correctly. Some of you wrote :: instead of @, we took two marks off for that. Others wrote various degrees of convoluted answers. We tried to give partial credit based on whether we thought you had some idea of the structure of the solution. We gave 5 for a garbled solution that showed some knowledge of how the correct solution should look. Some solutions showed basic lack of understanding about how to approach a recursive program. Only one person had apparently never heard of recursion and wrote while loops; giving a program that was far from working correctly. We generously gave zero for this.

Question 3 [15 points]

When the following expression is evaluated; the actual result is 7, as shown in the execution script below. Draw an environment diagram showing the bindings just as the execution of f(4) is about to begin.

let x = 1 in
let f =
    let x = (let u = 2 in u + x) in
    fun z -> x + z
f(4);;

val x : int = 1
val f : (int -> int)
val it : int = 7

A few people got this totally right. Most of you got this right except for not removing the u. We took 3 points off for that. A number of you left in the u and then made the closure pointer point to it, instead of to (x, 3) We took 3 more points off for that. A few of you wrote strange things like binding z to x + z or other such stuff. You never bind a variable to an expression!!!! We took 5 points off for that.