Homework

Task 1. Analyze framework loading

1. Create plain new Console application (e.g. **`dotnet new console`**)
2. Publish framework dependent version of it (using your IDE or by **`dotnet publish`**)
3. Use Process Monitor to see what I/O operations are made when published app is executed
    - probably it will be **`\bin\Release\net6.0\publish\HelloWorld.exe`**
4. Publish self-contained version of it
5. Again, use Process Monitor to confirm an app does NOT use anything beside local runtime
    - probably it will be **`\bin\Release\net6.0\win-x64\publish\HelloWorld.exe`**

Answer: In case of framework dependent deployment, it should load **`hostfxr.dll`** and the rest from one of the global installations like **`C:\Program Files\dotnet\shared\Microsoft.NETCore.App\5.0.6\`**. In case of self-contained indeed it should only use libraries from the application folder.

Task 2. Play with the Fibonacci

1. Open in sharplab.io and input F# code calculating *Fibonacci sequence* **`n`**-th element:

```
namespace Fibonacci.FSharp

module Math =
  let Fib n =
      let rec FibAcc a b n =
          match n with
          | 0 -> a
          | n -> FibAcc b (a+b) (n-1)
      FibAcc 0 1 n
```

   See how it is compiled to IL. Does it use **`tail.`** call or not? Try to understand the generated code.

   Answer: We see that **`Math`** module has been compiled into **`Math`** class and **`Fib`** function to a **`public static Fib`** method, using a private static **`FibAcc@4`** method, so there is one-to-one correspondence between F# and IL concepts. But! **`FibAcc@4`** should be recursive (calling itself), while we don't see any **`call`**/**`jump`** instructions inside it. We will solve this mystery in a second. Nevertheless, without calls, there is no need for **`tail`** instruction.

Task 2. Play with the Fibonacci (cont.)

See how it looks under C#.

Answer: In C# decompilation it is easier to see that the recurrsion has been translated into a single **`while(true)`** loop with a proper exit condition.

In the end, see the JIT result.

Answer: Well, I don't know how for you, but for me JIT result is just outstanding 😍! High level concepts of F# has been translated into really concise set of assembly instructions. Here some comments:

```
Fibonacci.FSharp.Math.FibAcc@4(System.Int32, System.Int32, System.Int32)
    L0000: test r8d, r8d          ; if r8d was 0 (so r8d represents n)
    L0003: ja short L0008         ; ... jump to L0008
    L0005: mov eax, ecx           ; otherwise set result to ecx
    L0007: ret                    ; and return
    L0008: dec r8d                ; n--
    L000b: add ecx, edx           ; ecx += edx
    L000d: xchg rdx, rcx          ; temp = rdx; rdx = rcx; rcx = temp;
    L0010: jmp short L0000        ; jump to the begining

Fibonacci.FSharp.Math.Fib(System.Int32)
    L0000: mov r8d, ecx
    L0003: xor ecx, ecx           ; set ecx to 0
    L0005: mov edx, 1             ; set edx to 1
    L000a: jmp 0x00007ffcbc1a0420 ; jump into FibAcc@4
```

Task 2. Play with the Fibonacci (cont.)

1. Write C# version of the same code.
    - experiment with making **`FibAcc`** a local function or static local function. What is being generated?
    - experiment with using pattern matching (C# **`switch`** expression) versus plain, old **`if`**. What is being generated?

Answer: Local function with pattern matching:

```
public class Math {
    public static int Fib(int n)
    {
        static int FibAcc(int a, int b, int n) => n switch
        {
            0 => a,
            _ => FibAcc(b, a + b, n - 1)
        };
        return FibAcc(0, 1, n);
    }
}
```

It produces nice code - local function is translated into **`<Fib>g__FibAcc|0_0`** method which calls itself recursively - we will see **`call int32 Math::'<Fib>g__FibAcc|0_0'(int32, int32, int32)`** inside of it. However, we don't see **`tail`** call here - the recursive call is not the last in the method, so it cannot happen.

Task 2. Play with the Fibonacci (cont.)

Answer (cont.): Local function with a simple confition:

```
public class Math {
    public static int Fib(int n)
    {
        static int FibAcc(int a, int b, int n)
        {
            if (n == 0)
                return a;
            return FibAcc2(b, a + b, n - 1);
        }
        return FibAcc(0, 1, n);
    }
}
```

It produces very similar IL code - local function is again translated into **<Fib>g__FibAcc|0_0** method which calls itself recursively. We still don't see **tail** call here although it could happen - it is the very last call before function ends.

Task 2. Play with the Fibonacci (cont.)

BUT! If you compare ASM results for those two methods, you will see a very important difference! The first version (using pattern matching) is indeed using a recursive call. BUT, the second version (with a simple `if`) was optimized to use just a single loop! That's pretty awesome for me! Not the compiler, but the JIT was able to get rid of recursion and replace it with a single loop 🤯.

In that manner, emitting or not a `tail` does not matter, if JIT was able to optimize recursion (a call) in the first place. Awesome!

Task 2. Play with the Fibonacci (cont.)

1. (hardcore🔥) Try to write in sharplab.io the CIL version of it, using the **`tail.`** opcode.

Answer: We can force using tail call by **`tail.`** prefix.

And as you will see in the ASM result, this... does not matter because again, JIT is getting rid of the recursion (call) in the first place :)

Remember, this is the case for this particular scenario! As a very optional fun, not even a homework, if you have enough time and feel interested, try to write a program that:

- written in C# produces real recursive call (not a loop-optimized counterpart)
- in its IL counterpart does use recursive call if **`tail.`** is not used and uses jumps/loops if **`tail.`** is used

Good luck! 💪