Recap
When is a recursive function call tail-recursive?

When it is the last expression to be evaluated before the return from the function.
When is a recursive function tail-recursive?

When every possible control flow path contains either:
1) exactly one recursive call that is tail-recursive, or
2) no recursive call at all.

In other words, a single non-tail-recursive call is sufficient to render a function non-tail-recursive.
Is this function tail-recursive? Why?

```haskell
len :: [a] -> Int
len [] = 0
len (_:xs) = 1 + len xs
```

No, the function is not tail-recursive, since the `+` function must be evaluated after `len xs`. 
Is this function tail-recursive? Why?

\[
\text{max'} :: [\text{Int}] \rightarrow \text{Int} \\
\text{max'} (x:[]) = x \\
\text{max'} (x:xs) = \begin{cases} 
    x & \text{if } x \geq \text{max'} xs \\
    \text{else } \text{max'} xs & \text{else}
\end{cases}
\]

No, the function is **not** tail-recursive, since the result from recursive call \(\text{max'} xs\) is required to decide which branch is executed, and thus it cannot be the last expression to be evaluated.
What is the key benefit of a statically, weakly typed language?

Execution speed: no runtime checks are required.
Which data type fundamentally requires runtime checks in a strongly typed language?

Disjoint union types; tag checks must occur at runtime.