Tail calls

If the result of \( f(x) \) is the result of the enclosing function body, then \( f(x) \) is a **tail call**.

More precisely, a tail call is a call in **tail position**:

- In \( \text{fun } f(x) = e \), \( e \) is in tail position.
- If \( \text{if } e1 \text{ then } e2 \text{ else } e3 \) is in tail position, then \( e2 \) and \( e3 \) are in tail position (not \( e1 \)). (Similar for case).
- If \( \text{let } b1 \ldots bn \text{ in } e \) and is in tail position, then \( e \) is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...

Implementing calls

Consider

\[
\begin{align*}
\text{fun } \text{len } [] & = 0 \\
\text{len } (x::xs) & = 1 - \text{len } xs;
\end{align*}
\]

\[
\text{val } \text{theLength } = \text{len } [1,2,3,4,5];
\]

Q: How do you implement function call?
A: A “Call Stack”

Compare:

\[
\begin{align*}
\text{fun } \text{last } [x] & = x \\
\text{last}(x::xs) & = \text{last } xs;
\end{align*}
\]

\[
\text{val } \text{theLast } = \text{last } [1,2,3,4,5];
\]

So what?

Why does this matter?

- Implementation takes space proportional to depth of function calls ("call stack" must "remember what to do next")
- But in functional languages, implementation must ensure tail calls eliminate the caller’s space
- Accumulators are a systematic way to make some functions tail recursive
- “Self” tail-recursive is very loop-like because space does not grow.