

```
fun append (xs,ys) =  
  if xs=[]  
  then ys  
  else (hd xs)::append(tl xs,ys)  
  
fun map (f,xs) =  
  case xs of  
    [] => []  
  | x::xs' => (f x)::(map(f,xs'))  
  
val a = map (increment, [4,8,12,16])  
val b = map (hd, [[8,6],[7,5],[3,0,9]])
```

Programming Languages

Dan Grossman
2013

Tail Recursion: Perspective and Definition

Always tail-recursive?

There are certainly cases where recursive functions cannot be evaluated in a constant amount of space

Most obvious examples are functions that process trees

In these cases, the natural recursive approach is the way to go

- You could get one recursive call to be a tail call, but rarely worth the complication

Also beware the wrath of premature optimization

- Favor clear, concise code
- But do use less space if inputs may be large

What is a tail-call?

The “nothing left for caller to do” intuition usually suffices

- If the result of $\mathbf{f\ x}$ is the “immediate result” for the enclosing function body, then $\mathbf{f\ x}$ is a tail call

But we can define “tail position” recursively

- Then a “tail call” is a function call in “tail position”

...

Precise definition

A tail call is a function call in *tail position*

- If an expression is not in tail position, then no subexpressions are
- In **fun f p = e**, the body **e** is in tail position
- If **if e1 then e2 else e3** is in tail position, then **e2** and **e3** are in tail position (but **e1** is not). (Similar for case-expressions)
- If **let b1 ... bn in e end** is in tail position, then **e** is in tail position (but no binding expressions are)
- Function-call *arguments* **e1 e2** are not in tail position
- ...