Where we are

Some implementation tidbits: ARs, call stacks & cons cells

Tail recursion avoids call stack overhead

Accumulator-style recursion typically tail-recursive

Today:

- more tail/accumulator examples
- more on pattern-matching as an elegant generalization of variable binding.
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 `fun f(x) = e`, $e$ is in tail position.
- If `if e1 then e2 else e3` is in tail position, then $e2$ and $e3$ are in tail position (not $e1$). (Similar for case).
- If `let b1 ... bn in e end` is in tail position, then $e$ is in tail position (not any binding expressions).
- Function arguments are not in tail position.
- ...
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.
fun rev1(nil) = nil
| rev1(x::xs) = rev1(xs) @ [x];

Run time?
fun rev1(nil) = nil
| rev1(x::xs) = rev1(xs) @ [x];

Run time?

$O(n^2)$!

L1 @ L2 must copy L1:

fun append([],l2) = l2
| append(x::xs,l2) = x::append(xs,l2);

So rev1([1,2,\ldots,n]) takes time

$1 + 2 + \cdots + n = O(n^2)$. 
A Classic—Reversing a List III

fun rev1(nil) = nil
| rev1(x::xs) = rev1(xs) @ [x];

fun rev2 lst =
    let fun f (nil, acc) = acc
    | f (x::xs, acc) = f(xs,x::acc)
    in
    f(lst,nil)
    end

The standard trick: Do ops on way in, not way out. Instead of operating on recursive result, move operation into the recursive call.

Run time, now?
Deep patterns

Patterns are much richer than we have let on. A pattern can be:

- A variable (matches everything, introduces a binding)
- `_` (matches everything, no binding)
- A constructor and a pattern (e.g., `C p`) (matches a value if the value “is a C” and `p` matches the value it carries)
- A pair of patterns (`(p1, p2)`) (matches a pair if `p1` matches the first component and `p2` matches the second component)
- A record pattern...
- An integer constant...
- ...

CSE 341 Spring 2007, Lecture 7
The truth, the whole truth, and nothing but

It’s really:

- `val p = e`
- `fun f p1 = e1 | f p2 = e2 ... | f pn = en`
- `case e of p1 => e1 | ... | pn => en`

Inexhaustive matches may raise exceptions and are bad style.

Example: could write `Rope pr` or `Rope (r1, r2)`

Fact: Every ML function takes exactly one argument!
Some function examples

- \( \text{fun } f_1() = 34 \)
- \( \text{fun } f_2(x,y) = x + y \)
- \( \text{fun } f_3 \, \text{pr} = \text{let } \text{val } (x,y) = \text{pr} \text{ in } x + y \text{ end} \)

Is there any difference to callers between \( f_2 \) and \( f_3 \)?

In most languages, “argument lists” are syntactically separate, \textit{second-class} constructs.

Can be useful: \( f_2(\text{if } e_1 \text{ then } (3,2) \text{ else } \text{pr}) \)