CSE 341:
Programming Languages

Spring 2005
Lecture 7 — More on Tail Recursion & Accumulators; Deep Patterns
Where we are

Two implementation tidbits: call stack & cons cells

Tail recursion avoids call stack overhead

Accumulator-style recursion typically tail-recursive

Today:

• one more tail/accumulator example

• more on pattern-matching as an elegant generalization of variable binding.

• first-class functions (closures, functions as values)—A really key idea in computer science
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.
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., \texttt{C p}) (matches a value if the value "is a \texttt{C}" and \textit{p} matches the value it carries)
- A pair of patterns ((\textit{p1}, \textit{p2})) (matches a pair if \textit{p1} matches the first component and \textit{p2} matches the second component)
- A record pattern...
- An integer constant...
- ...

CSE 341 Spring 2005, Lecture 7
The truth, the whole truth, and nothing but the truth

It’s really:

- \texttt{val p = e}
- \texttt{fun f p1 = e1 | f p2 = e2 \ldots | f pn = en}
- \texttt{case e of p1 \Rightarrow e1 | \ldots | pn \Rightarrow en}

Inexhaustive matches may raise exceptions and are bad style.

Example: could write \texttt{Rope pr} or \texttt{Rope (r1,r2)}

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

- `fun f1 () = 34`
- `fun f2 (x,y) = x + y`
- `fun f3 pr = let val (x,y) = pr in x + y end`

Is there any difference to callers between `f2` and `f3`?

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

Can be useful: `f2 (if e1 then (3,2) else pr)`