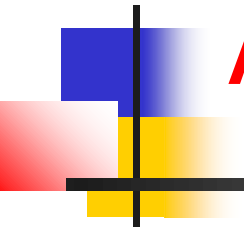
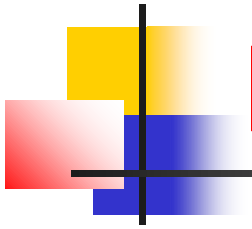


Accumulators and Tail Calls

Corky Cartwright

Department of Computer Science

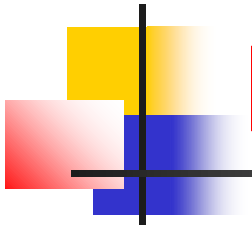
Rice University



Plan for today

Provide a more precise technical motivation for accumulator concepts

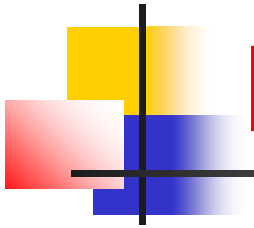
- Recognizing the need for accumulators
- Avoiding non-tail recursive calls
- Avoiding unnecessary traversals of a list (which often correlates with using accumulators)
- The book inexplicably does not discuss tail calls/tail recursion, which was a major focus of the pre-book Comp 210 course. Tail-recursion is *extremely* important because it corresponds to iteration (loops).



Last class: Accumulators

Looked at three examples

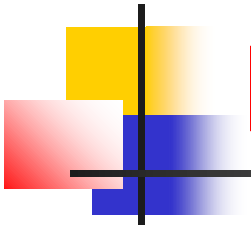
- Computing the partial sums for a sequence
- Reversing a list
- Flattening a general list



Last class: Accumulators

In coding an accumulator based solution to a programming problem, the programmer must answer the following three questions:

- How to use the accumulated value to produce the final answer.
- How to update the accumulated value when we make a recursive call.
- How to initialize the accumulated value.



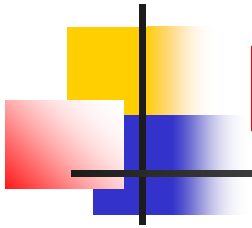
Design Recipe

When do we need an accumulator?

We pointed out some tell-tail(!) signs:

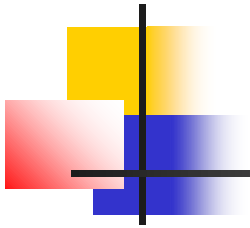
- We miss some context (or history) information
- The result of the recursive call is processed by another function (a non-tail call!)
- Difficulty in ensuring termination (uncommon because setting a flag in each node [structure] as it is visited is usually preferable

This list is not exhaustive



Factorial

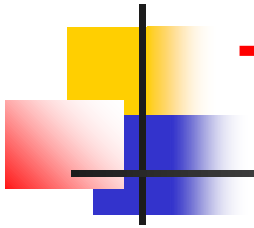
- Even factorial could benefit from an accumulator!
- Why? The recursive call on fact is embedded inside another call (on *) which means the computer must maintain a calling stack to manage the recursion! Call-stack maintenance is NOT free.
- Which one is faster?



Dueling Factorials

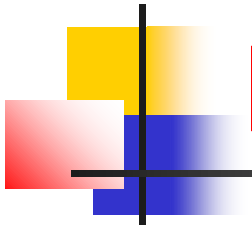
```
(define (fact n)
    (cond [(= n 0) 1]
          [else (* n (fact (sub1 n)))]))
(define (fact-help n ans)
    (cond [(= n 0) ans]
          [else (fact-help (sub1 n) (* n
  ans))]))
(define (fast-fact n) (fact-help n 1))
```

The recursive call on **`fact-help`** is in *tail position* in the function body, meaning that no subsequent processing is performed on the result of the call in evaluating the body.



Tail Position

- Consider our law for evaluating Scheme expressions. When a function call (application) embedded in the body of a function definition (for **`f`**) must be the last form to be reduced in evaluating a call **`(`*`f`*** $\mathbf{v_1}$ … $\mathbf{v_n}$**`)`**, that call is in *tail position*.
- Note that a function definition may have several different embedded calls that are in tail position. Why? Because conditionals create multiple control paths through the function. Think about our reduction rules for **`cond`**.



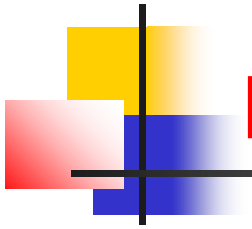
Examples

- Which function calls are in tail position?

```
(define (fact n)
   (cond [(= n 0) 1]  ;; no function call in result
         [else (* n (fact (sub1 n)))]))
```

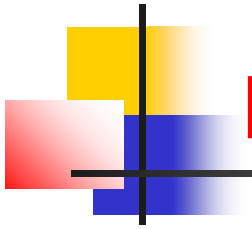
```
(define (fact-help n ans)
   (cond [(= n 0) ans]  ;; no function call in result
[else (fact-help (sub 1) (* n ans))]))
```

```
(define (fast-fact n) (fact-help n 1))
```



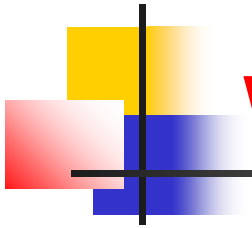
Reduction pattern (factorial)

```
     (fact 3)
=>*  (* 3 (fact 2))
=>*  (* 3 (* 2 (fact 1)))
=>*  (* 3 (* 2 (* 1 (fact 0))))
=>*  (* 3 (* 2 (* 1 1)))
=>   (* 3 (* 2 1))
=>   (* 3 2)
=>   6
```



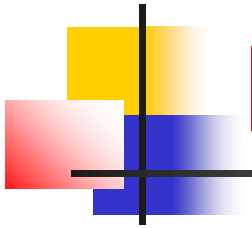
Reduction pattern (fast-fact)

```
     (fast-fact 3)
=>   (fact-help 3 1)
=>*  (fact-help 2 (* 3 1))
=>   (fact-help 2 3)
=>*  (fact-help 1 (* 2 3))
=>   (fact-help 1 6)
=>*  (fact-help 0 (* 1 6))
=>   (fact-acc 0 6)
=>*  6
```



Why are non-tail calls expensive?

- Tail-calls compile to jumps (branches). Tail recursive calls compile to loop (backwards) branches. Branches are only slightly more expensive than ordinary instructions. In fact they can actually be cheaper with the right hardware support.
- Non-tail calls compile (translate) to expensive "jump to subroutine" instructions (which are covered in both Elec 220 and Comp 221).
- A "jump to subroutine" instruction allocates space in the control stack (a chunk of memory reserved by the OS for each computation) to save information about the machine state at the point of the call, saves critical information and then jumps to the subroutine. The saved information enables a subsequent "subroutine return" instruction to restore the machine state at the the point of call (with the returned value in a designated register).
- In contrast, a tail call *never needs to return* to the point of call. It can return to the caller's point of call. Hence, It simply stores the result in the appropriate register and follows the control dictated by the context of the point-of-call.



Naive Reverse

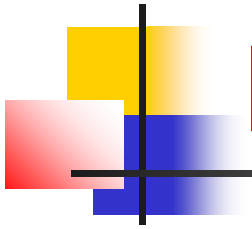
```
(define (rev l)
  (cond [(empty? l) empty]
        [else
          (append (rev (rest l))
                  (list (first l)))]))
```

The subcomputation in red must be performed after the call on **`rev`** completes

Tail-recursive Reverse

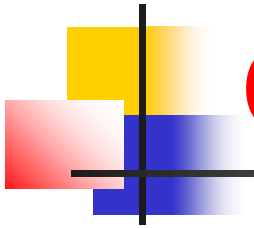
```
(define (rev-help l ans)
  (cond [(empty? l) ans]
        [else
          (rev-help (rest l)
                    (cons (first l) ans))]))
(define (fast-rev l) (rev-help l empty))
```

The recursive call on **`rev-help`** can be implemented by branching back to the code for the enclosing **`cond`** operation reusing the stack frame for the calling invocation of **`rev-help`**. The initial call on **`rev-help`** (in **`reverse`**) can be "inlined". So the optimized machine translation of this call looks just like it does for corresponding loop code.



Example reductions

- Try
  `(reverse '(a b c))`
  using the DrScheme stepper.
- Try
  `(fast-rev '(a b c))`

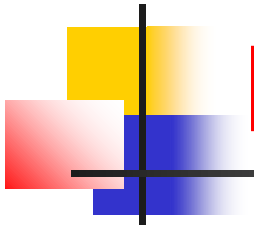


Conversion to tail form

Always possible? Always profitable?

- Always possible, but requires passing closures (functions) as arguments, which requires heap allocation. Called conversion to continuation-passing style (CPS). May or may not be profitable. Covered in depth in Comp 311.

What about our examples? They did not use closures. We relied on associativity of *, conversion of **append** to **cons**. Tail calls are very profitable when they substitute a cheap operation like **cons** for an expensive one like append. They can potentially produce a huge space improvement if no closure allocation is required and significant time improvement if the cost of corresponding operations is unchanged.



For Next Class

- Homework (correct version on wiki) due Monday