CMPU 101 § 55

Computer Science I

Problem-Solving and Abstraction

Prof. Jonathan Gordon

Lecture 11

Accumulators and tail recursion

An ***accumulator*** is an argument of a function that's used to "accumulate" the result of an ongoing computation.

Imagine you have a long list of numbers to add up:

```
    37
 2,384
74,656
   192
 7,720
   584
   968
 4,003
 8,192
 3,373
     8
   592
    17 +
______
```

Besides reaching for a calculator, what could you do?

- It’s difficult to add up all the digits in each column without making a mistake!
- but you could add the first pair of numbers and then add each number to that, so you’re only ever adding two numbers – the running total and the next number!

Imagine you have a long list of numbers to add up:

37	
2,384	2,421
74,656	77,077
192	77,269
7,720	84,989
584	85,573
968	86,541
4,003	90,544
8,192	98,736
3,373	102,109
8	102,116
592	102,708
17	+ 102,725
102,725	

We can write a function, `sum`, to add up a list of numbers the same way, using an accumulator for our running total.

```
> (sum '(1 0 -3 4 5))
7

> (sum '())
0
```

We can write:

- a “helper function” that includes an accumulator as an extra input
- a simple “wrapper function” that calls the helper function with a properly initialized accumulator argument

```
(define sum
  (lambda (lst)
    (sum-helper ...?...)))


(define sum-helper
  (lambda (lst acc)
    ...?...))
```

The accumulator serves to collect the sum of the elements in the list.

For the base case, we can just return the contents of the accumulator.

The recursive step involves adding the first element to the accumulator.

```
(define sum-helper
  (lambda (lst acc)
    ;; If the list is empty...
    (if (null? lst)
        ;; then we're done, so report the
        ;; accumulated value...
        acc
        ;; otherwise, keep accumulating
        (sum-helper (rest lst)
                    (+ (first lst) acc)))))
```

Now the simple wrapper function, sum.

This function takes only a list of numbers as input.

The person using the sum function doesn't need to know about the helper function or the initial value of the accumulator.

```
(define sum
  (lambda (lst)
    ;; Accumulator is initially 0
    (sum-helper lst 0)))
```

Trace of evaluation of **sum** with helper and accumulator

```
(sum        '(1 2 3 4 5))
(sum-helper '(1 2 3 4 5)  0)
(sum-helper '(2 3 4 5)    1)
(sum-helper '(3 4 5)      3)
(sum-helper '(4 5)        6)
(sum-helper '(5)          10)
(sum-helper '()           15)
15
```

Normally when we write a recursive function, the recursive call is embedded in another function call, e.g.,

```
(define foo
  (lambda (lst)
    (if ...?...
        '()
        (cons (bar (first lst))
              (foo (rest lst))
```

```
(define foo
  (lambda (lst)
    (if ...?...
        '()
        (cons (bar (first lst))
              (foo (rest lst))
```

This can't evaluate any of the `cons` calls until it finishes going through the entire list and produces the `()` to `cons` the first result onto.

```
(cons 'i (cons 'am (cons 'waiting ())))
```

In the recursive case of `sum-helper`, the return value is given by a recursive function call to the same helper function.

- There are no earlier recursive calls waiting on a later recursive function call to return.
- Whatever the last recursive function call returns, that value is returned as the value of the helper function.

When this is the case, the recursion is called *tail recursion*.

- Tail recursion can be implemented very efficiently!

Factorials revisited

Factorial

The *factorial* notation $n!$ represents the product of all positive integers from 1 to n, inclusive,

$$n! = 1 \times 2 \times 3 \times \cdots \times (n-1) \times n$$

The factorial function can be defined recursively:

Recursive step: $n! = n \cdot (n-1)!$

Base step: $n = 1$, in which case $n! = 1! = 1$

Normal recursive definition of **`factorial`**

```
(define factorial
  (lambda (n)
    (if (< n 2)
        1
        (* n (factorial (- n 1))))))
```

```
(factorial 3)
(* 3 (factorial 2))
(* 3 (* 2 (factorial 1)))
(* 3 (* 2 (* 1 (factorial 0))))
(* 3 (* 2 (* 1 1)))
(* 3 (* 2 1))
(* 3 2)
6
```

Procedure calls and return values

$$\begin{tikzcd}
\texttt{(factorial 3)} \arrow[d] & 6 \\
\texttt{(factorial 2)} \arrow[d] & 2 \arrow[u] \\
\texttt{(factorial 1)} \arrow[d] & 1 \arrow[u] \\
\texttt{(factorial 0)} & 1 \arrow[u]
\end{tikzcd}$$

Tail-recursive definition of **`factorial`**

```
(define factorial-acc-helper
  (lambda (n acc)
    ;; If we're down to 0 or 1...
    (if (< n 2)
        ;; we're done; return acc...
        acc
        ;; otherwise, keep accumulating
        (factorial-acc-helper
          (- n 1)
          (* acc n)))))

(define factorial-acc
  (lambda (n)
    ;; Initialize accumulator to 1,
    ;; the multiplicative identity
    (factoral-acc-helper n 1)))
```

The Fibonacci sequence

The *Fibonacci sequence* is a famous problem published in 1202 in the *Liber abaci* by Italian merchant and mathematician Leonardo di Pisa[1].

Like all important mathematical sequences, it's about rabbits.

Illustration by Pär Gullberg

1: Fibonacci? *Figlio dei Bonacci* or "son of the Bonaccis"

Rabbit breeder's problem

Start with a pair of newborn rabbits. (*1 pair*)

At one month, they are too young to breed. (*1 pair*)

At two months, they produce one pair of offspring. (*2 pairs*)

At three months, they produce another pair of offspring. (*3 pairs*)

At four months, each pair alive at two months produces a new pair of offspring. (*5 pairs*)

At *n* months, each pair alive at *n*−2 months produces a new pair of offspring.

Rabbit breeding records

Months	Pairs	Total
0	xx	1
1	xx	1
2	xx xx	2
3	xx xx xx	3
4	xx xx xx xx xx	5
5	xx xx xx xx xx xx xx xx	8

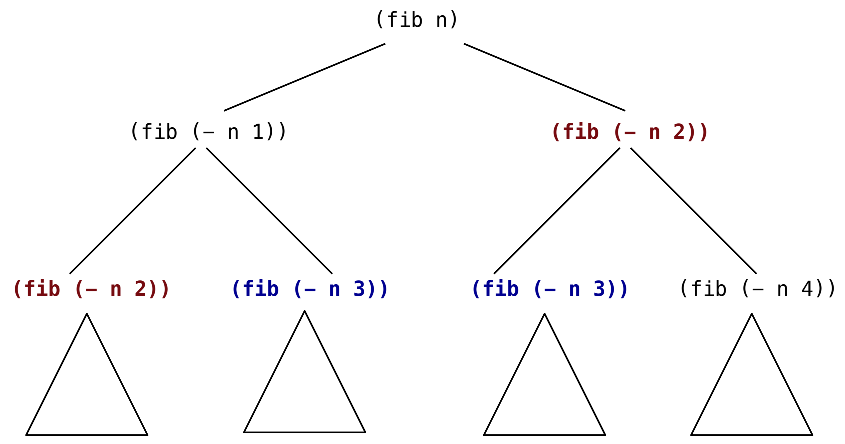
A recursive formula for the Fibonacci numbers

$$\textit{Fibonacci}(n) = \begin{cases} 1 & \textit{if } n = 0 \\ 1 & \textit{if } n = 1 \\ \textit{Fibonacci}(n-1) + \textit{Fibonacci}(n-2) \end{cases}$$

Recursive procedure for computing Fibonnaci(*n*)

```
(define fib
  (lambda (n)
    (if (< n 2)
        1
        (+ (fib (- n 1))
           (fib (- n 2))))))
```

`fib` call tree



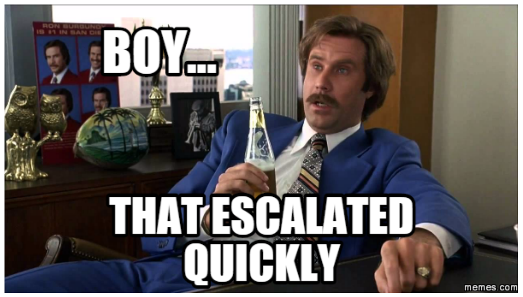
A cause for concern

An evaluation of `(fib n)` leads to two evaluations of `(fib (- n 2))`.

…And many evaluations of `(fib i)` for smaller values of *i*.

Costs of evaluating `(fib n)`

n	**Calls to `fib`**
1	1
2	3
4	9
8	67
16	3,193
32	7,049,155



The `fib` procedure with two accumulation parameters

```
(define fib
  (lambda (n)
    (if (= n 0)
        1
        (fib-helper (- n 1) 1 1))))

(define fib-helper
  (lambda (remaining i j)
    (if (= remaining 0)
        j
        (fib-helper
          (- remaining 1)
          j
          (+ i j)))))
```

Behavior of **`fib`** procedure using the accumulator method

```
        (fib 4)
           |
   (fib-helper 3 1 1)
           |
   (fib-helper 2 1 2)
           |
   (fib-helper 1 2 3)
           |
   (fib-helper 0 3 5)
```

Comparing costs of evaluating **`(fib n)`** with and without accumulator

n	**Calls w/o acc.**	**Calls w/ acc.**
1	1	2
2	3	3
4	9	5
8	67	9
16	3,193	17
32	7,049,155	33

More practice with tail recursion

An accumulator doesn't have to accumulate just numerical quantities; it can accumulate *any* type of Scheme expression.

Let's try accumulating our favorite type of Scheme expression – a list!

Creating a list with an accumulator

First, let's define our function without using tail recursion or an accumulator.

`create-list` takes a number n as input and returns a list containing the numbers from n down to 1:

```
> (create-list 5)
(5 4 3 2 1)
```

```
(define create-list
  (lambda (n)
    (if (= n 0)
        '()
        (cons n (create-list (- n 1))))))


> (create-list 5)
(5 4 3 2 1)
```

Now with tail recursion and an accumulator:

```
(define create-list-acc-helper
  (lambda (n acc)
    ...?...))

(define create-list-acc
  (lambda (n)
    (create-list-acc-helper ...?...)))
```

Now with tail recursion and an accumulator:

```
(define create-list-acc-helper
  (lambda (n acc)
    ...?...))

(define create-list-acc
  (lambda (n)
    ;; Init. accumulator to empty list
    (create-list-acc-helper n '())))
```

Now with tail recursion and an accumulator:

```
(define create-list-acc-helper
  (lambda (n acc)
    ;; If we're down to 0
    (if (= n 0)
        ;; we're done; return acc
        acc
        ;; otherwise, keep accumulating
        ;; elements.
        (create-list-acc-helper
          (- n 1)
          (cons n acc)))))

(define create-list-acc
  (lambda (n)
    ;; Init. accumulator to empty list
    (create-list-acc-helper n '())))
```

But the output of our tail-recursive function is backwards:

```
> (create-list 5)
(5 4 3 2 1)

> (create-list-acc 5)
(1 2 3 4 5)
```

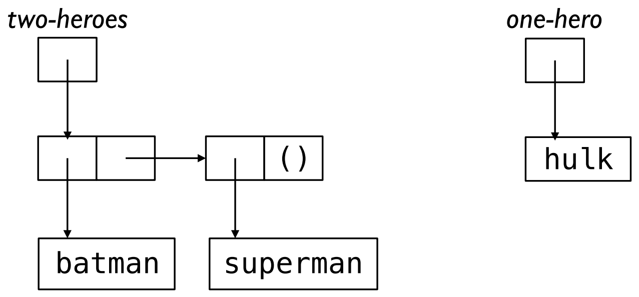
Reversing a list

```
(define reverse (lambda (lst) ...?...))


> (reverse '(a b c))
'(c b a)

> (reverse '(a))
'(a)

> (reverse '())
'()
```

One way to reverse elements in a list would be to use the built-in append function:

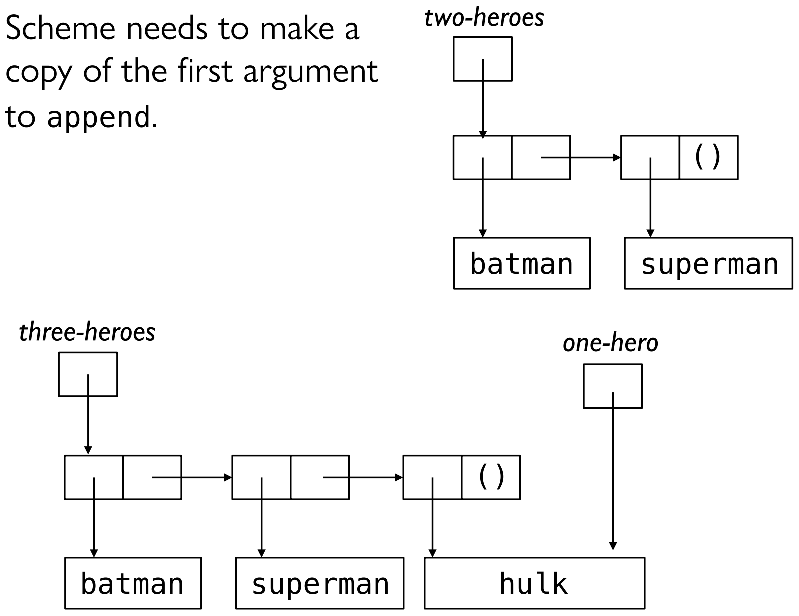
```
(define reverse-v1
  (lambda (listy)
    (if (null? listy)
        '()
        (append (reverse-v1 (rest listy))
                (list (first listy))))))
```

But the way append works is it walks through the first list until it gets to the end, then it points to the second list.



```
> (append two-heroes (list one-hero))
(batman superman hulk)
```

Scheme needs to make a copy of the first argument to append.



This makes for a very inefficient reverse function since we call append as many times as there are elements in the list we're reversing.

A more efficient approach is – again – to use an accumulator!

An accumulator approach to **reverse**

```
(define reverse-helper
  (lambda (lst answer)
    (if (null? lst)
        answer
        (reverse-helper (rest lst)
                        (cons (first lst)
                              answer)))))

(define reverse
  (lambda (lst)
    (reverse-helper lst '())))
```

What is **reverse-helper**?

It takes two arguments, `lst` and `answer`.

It returns: `(append (reverse lst) answer)`.

It does not actually call the append procedure.

Instead it takes the `first` of `lst` and uses `cons` to put it at the beginning of `answer`.

It calls itself recursively on `(rest lst)` and `(cons (first lst) answer)`.

Trace of evaluation of new **reverse** procedure

```
(reverse '(a b c))

(reverse-helper '(a b c)  '())

(reverse-helper '(b c)    '(a))

(reverse-helper '(c)      '(b a))

(reverse-helper '()       '(c b a))

'(c b a)
```

The accumulator method

A recursive function has a special parameter called the “accumulator”.

Each time the function calls itself recursively, it augments the accumulator.

- E.g., consing something into an accumulator list.
- E.g., adding something to an accumulator number.

When recursion stops, the accumulator has the final answer.

Acknowledgments

This lecture incorporates material from:

- Simon Ellis
- Tom Ellman
- Jan Gullberg
- Luke Hunsberger