Module 3: Accumulative recursion

Topics:
- Accumulative recursion: the idea
- Examples of accumulative recursion
- Designing and debugging accumulatively recursive code
- Introduction to algorithmic efficiency

Readings: HtDP 30, 31

Review: Structural Recursion

- Template for code is based on recursive definition of the data, for example:
  - Basic list template
  - Countdown template for natural numbers

Recall how Structural Recursion works

```
;; factorial: nat -> nat[>0]
;; Produces the product of the
;; integers from 1 to n
;; Examples: (factorial 1) => 1
;;           (factorial 5) => 120
(define (factorial n)
  (cond [(<= n 1) 1]
        [else (* n (factorial
                     (sub1 n)))]))
```

Trace **factorial**

```
(factorial 6)
=> (* 6 (factorial 5))
=> (* 6 (* 5 (factorial 4)))
=> (* 6 (* 5 (* 4 (factorial 3))))
=> (* 6 (* 5 (* 4 (* 3 (factorial 2)))))
=> (* 6 (* 5 (* 4 (* 3 (* 2
                          (factorial 1))))))
=> (* 6 (* 5 (* 4 (* 3 (* 2 1)))))
=> (* 6 (* 5 (* 4 (* 3 2))))
...
=> 720
```


An alternative approach – do one multiplication on each recursive call

```
(define (factorial2 n)
  ;; running-product: nat nat[>0]->nat[>0]
  ;; produces n0!*prod-so-far
  (local
    [(define (running-product
                       n0 prod-so-far)
       (cond
         [(<= n0 1) prod-so-far]
         [else (running-product (sub1 n0)
                  (* prod-so-far n0))]))]
    (running-product n 1)))
```


Trace **factorial2**

```
(factorial2 6)
=> (running-product 6 1)
=> (running-product 5 (* 1 6 ))
=> (running-product 5 6)
=> (running-product 4 (* 6 5))
=> (running-product 4 30)
=> (running-product 3 (* 30 4))
=> (running-product 3 120)
=> (running-product 2 (* 120 3))
=> (running-product 2 360)
=> (running-product 1 (* 360 2))
=> (running-product 1 720)
=> 720
```

Differences and similarities in implementations

- **`factorial2`** needs a helper function to keep track of the work done so far
- Both are correct, but
  - **`factorial`** does all calculations at the end
  - **`factorial2`** does the calculations as we go
    - **`prod-so-far`** is called the "accumulator"
- Mathematically equivalent, but not computationally equivalent.

Accumulative function template

```
(define (acc-fun x)
 (local
  [(define (helper whats-left ans-so-far...)
     (cond
      [base-case-using whats-left
              ... ans-so-far ...]
      [else ...
         (helper ... update-whats-left
                 ... update-ans-so-far )]))]
   ;; call the helper function
   (helper initial-whats-left
          initial-ans-so-far)))
```


Accumulative recursion ...

- Might make better use of space
- Might make code run faster (more later!)

Using an accumulator for **`list-max`**

- Remember the largest value seen so far
- After examining every entry in the list, you have the maximum value
- Filling in the template:
  - **`ans-so-far`**: the largest value in the list so far
  - **`whats-left`**: the unexamined list (i.e. the rest of the list)

Start with the template

```
(define (list-max lst)
  ;; list-max-accum: (listof num) num -> num
  ;; produce the larger of max-so-far and
  ;; the largest value in lon
  (local
    [(define (list-max-accum lon max-so-far)
      (cond [(empty? lon) ... ]
            [else (list-max-accum (rest lon)
                       ... max-so-far...)]))]
     (list-max-accum ... ... )))
```


Continuing with the unknowns

- Completing **`list-max-accum`**:
  - What is the answer if **`lon`** is empty?
  - How to update the value of **`max-so-far`** in the recursive call?
- Completing **`list-max`**:
  - What should the initial values of the parameters be?

Completed `list-max`

```
(define (list-max lst)

  ;; list-max-accum: (listof num) num -> num
  ;; produce the larger of max-so-far and the
  ;; largest value in lon
  (local
    [(define (list-max-accum lon max-so-far)
       (cond [(empty? lon) max-so-far]
             [else (list-max-accum (rest lon)
               (max (first lon) max-so-far))]))]

   (list-max-accum (rest lst) (first lst))))
```


Running Times: An introduction

Suppose you have two algorithms to solve a problem. How can we say one algorithm is faster? What can we compare?

- Running time on a *single* input
- Running time on *all* inputs
- Running time on a *typical* input
- *Average* running time over all inputs
- *Best-case* running time over all inputs
- *Worst-case* running time over all inputs

Measuring Worst Case Running Time for Recursive Code

- Consider n, the size of the input data, e.g.
  - Length of list
  - Natural number being considered
- Determine how many steps, in terms of n, are performed to solve the problem
  - It often helps to determine the number of times the recursive function is called, and
  - How many steps are performed in any one recursive call?

Common Run-time Categories

- Constant running time, denoted $O(1)$
  - Independent of the size of the input
  - e.g.: (first L), (rest L), (cons x L), (abs n), ...
- Linear running time, denoted $O(n)$
  - Proportional to the size of the input
  - For lists, a constant amount of work done for each element in the list
  - e.g.: adding all values in a list, finding the maximum value *(our good version, that is)*, ...

Another Common Run-time Category

- Quadratic running time, denoted $O(n^2)$
  - The running time is proportional to the square of the size of the input
  - For lists, a linear amount of work is done for each element in the list
  - *(we'll see some examples soon)*

Yet another Common Run-time Category

- Exponential running time, denoted $O(2^n)$
  - As the size of the input increases by one, the running time doubles
  - Often observed in recursion when the exact same recursive call is performed multiple times
  - e.g. original version of **`list-max`** from Module 1

Testing Accumulative Recursive Code

- Test each **cond** clause in main body
- Test each **cond** clause in the helper function
- Include data that tests the helper in the base case, near-base case, non-base case
- Be careful: Failing tests could be due to
  - Errors in the helper base case(s)
  - Errors in the helper recursive case(s)
  - Errors in the initial values in call to helper
  - Errors in other parts of the main body

Another accumulative example: Fibonacci numbers

The nth Fibonacci number is the sum of the two previous Fibonacci numbers:

$$fn = f_{n-1} + f_{n-2},$$
$$\text{where } f_0=0, f_1=1.$$

These numbers grow very quickly!

$$f_5 = 5, \quad f_{10} = 55, \quad f_{15} = 610,$$
$$f_{20} = 6765, \quad f_{25} = 75{,}025,$$
$$f_{30} = 832{,}040, \quad f_{35} = 9{,}227{,}465$$

First attempt: straight from the definition

```
;; fib: nat -> nat
;; produce the nth Fibonacci number
;; Examples: (fib 0)=>0, (fib 6)=>8
(define (fib n)
  (cond [(zero? n) 0]
        [(= n 1) 1]
        [else (+ (fib (sub1 n))
                 (fib (- n 2)))]))
```

But, this is ***very*** slow. Why?

- Consider **`(fib 10)`**:
  - **`(fib 9)`** is called 1 times
  - **`(fib 8)`** is called 2 times
  - **`(fib 7)`** is called 3 times
  - **`(fib 6)`** is called ***??*** times
  - ...
  - **`(fib 1)`** is called ***??*** times
- How many times is **`(fib 1)`** called to calculate **`(fib n)`** for any value of **`n`**?
- Running time => Exponential

Use Accumulative Recursion

- Remember the fibonacci numbers by storing them in a list:

  `(list 0 1 1 2 3 5 8...)`
- But
  - Need fast access to two most recent numbers
  - Slow to get to end of list

Use Accumulative Recursion

- Maintain list `L` with most recent at the front
- **`Next`** is **`(+ (first L) (second L))`**
- New list is then **`(cons Next L)`**

- Also, use **`n0`** to keep track of which fibonacci number is at front of list
- Stop when **`n0`** equals **`n`**

```
(define (fib2 n)
  (local
    ;; fib-acc: nat (listof nat) -> nat
    ;; produces nth fib number, where n0th fib
    ;; number is at front of fibs-so-far
    [(define (fib-acc n0 fibs-so-far)
       (cond
         [(= n0 n) (first fibs-so-far)]
         [else (fib-acc (add1 n0)
                 (cons (+ (first fibs-so-far)
                          (second fibs-so-far))
                       fibs-so-far))]))]
    ...))
```


Completing body of **fib2**

- **fib-acc** requires a list of at least length 2
  - Have base case for **n=0** in main body
  - When **n>0**,
    - **fibs-so-far** needs first two fibonacci numbers, **(list (fib 1) (fib 0))** or **(list 1 0)**
    - Initial value for **n0** should be 1, since **(fib 1)** at front of **fibs-so-far**

Completed version of **fib2**

```
(define (fib2 n)
  (local
    [(define (fib-acc n0 fibs-so-far)
       (cond [(= n0 n) (first fibs-so-far)]
             [else (fib-acc (add1 n0)
                     (cons (+ (first fibs-so-far)
                              (second fibs-so-far))
                           fibs-so-far))]))]
    (cond
      [(zero? n) 0]
      [else (fib-acc 1 (list 1 0))])))
```

Tracing **fib2**

```
(fib2 10)
=>(fib-acc 1 (list 1 0))
=>(fib-acc 2 (list 1 1 0))
=>(fib-acc 3 (list 2 1 1 0))
=>(fib-acc 4 (list 3 2 1 1 0))
=>(fib-acc 5 (list 5 3 2 1 1 0))
...
=>(fib-acc 10 (list 55 34 21 13 8 5 3 2 1 1 0))
=>(first (list 55 34 21 13 8 5 3 2 1 1 0))
=>55
```


Running Time of **fib2**

- Consider **(fib2 10)**:
  - **(fib-acc 1 ...)** is called 1 time
  - **(fib-acc 2 ...)** is called 1 time
  - **(fib-acc 3 ...)** is called 1 time
  - **(fib-acc 4 ...)** is called ***??*** times
  - ...
  - **(fib-acc 10 ...)** is called ***??*** times
- How many times is **fib-acc** called to calculate **(fib2 n)** for any value of **n**?
- Running time => Linear

Improving **fib2**

- Anything wrong with **fib2**?
  - Remembered all previous numbers, but only needed last two

Another implementation

```
(define (fib3 n)
  (local
   [(define (fib-acc n0 last-fib
                        prev-fib)
      (cond [(= n0 n) last-fib]
            [else (fib-acc (add1 n0)
                   (+ last-fib prev-fib)
                   last-fib)]))]
   (cond [(zero? n) 0]
         [else (fib-acc 1 1 0)])))
```


Design choices

Two important features of a computer program are
- how much time it takes
- how much memory it uses.

Often these are in opposition:
- We can sometimes make solution faster by storing intermediate results

You can see much more about this topic in a data structures course like CS 234 or 240.

Reversing a List

```
;; invert: (listof any)->(listof any)
;; produces a list with the elements
;; in reverse order of lst
;; Examples:
;; (invert empty) => empty
;; (invert (list 1 2 3)) => (list 3 2 1)
(define (invert lst)
  (cond
    [(empty? lst) empty]
    [else (append (invert (rest lst))
               (list (first lst)))]))
```

Tracing **invert**

```
(invert (list 'a 'b 'c 'd))
⇒ (append (invert (list 'b 'c 'd)) (list 'a))
⇒ (append (append (invert (list 'c 'd)) (list 'b))
  (list 'a))
⇒ (append (append (append (invert (list 'd)) (list
  'c)) (list 'b)) (list 'a))
⇒ (append (append (append (append (invert empty)
  (list 'd)) (list 'c)) (list 'b)) (list 'a))
⇒ (append (append (append (append empty (list 'd))
  (list 'c)) (list 'b)) (list 'a))
⇒ (append (append (append (list 'd) (list 'c))
  (list 'b)) (list 'a))
⇒ (append (append (list 'd 'c) (list 'b)) (list
  'a))
⇒ (append (list 'd 'c 'b) (list 'a))
⇒ (list 'd 'c 'b 'a)
```


Analyzing run-time of **invert**

- For a list of length n,
  - n+1 calls of invert (for lists original list of length n, then a list of length n-1, then n-2, then n-3, etc, down to a list of length 0).
- For each recursive call, **append** is called.
  - What is the running time of **append** ?

Running time of **append**

```
(define (my-append l1 l2)
  (cond
    [(empty? l1) l2]
    [else
      (cons (first l1)
       (my-append (rest l1) l2))]))
```

- How many recursive calls?
- How many steps for each recursive call?
- Total running time?

Back to running time of **invert**

- For a list of length **n**, **n+1** calls of **invert**
- For each recursive call, **append** is called.
  - First call, **n-1** steps (length of first argument to append)
  - Next call, **n-2** steps (length of first argument to append)
  - Next call, **n-3** steps
  - ...
  - Final call, 0 steps (first argument is empty)

⇒ **(n-1) + (n-2) + ... + 1 + 0 = n(n-1)/2**

⇒ Quadratic running time

A better version of **invert**: accumulate the list in reverse order

```
(define (invert2 lst0)
  (local
    [(define (invert-acc lst
                          reversed-so-far)
       (cond [(empty? lst)
                        reversed-so-far]
             [else (invert-acc (rest lst)
                         (cons (first lst)
                         reversed-so-far))]))]
   (invert-acc lst0 empty)))
```


Tracing **invert2**

```
(invert2 (list 3 6 5))
=>(invert-acc (list 3 6 5) empty)
=>(invert-acc (list 6 5) (list 3))
=>(invert-acc (list 5) (list 6 3))
=>(invert-acc empty (list 5 6 3))
=>(list 5 6 3)
```

Using **invert2** to reverse a list with n elements takes O(n) steps.

Testing `invert2`

- Empty list
- List of length 1
- List with a few elements
- A long list

Goals of Module 3

- Understand how to write accumulative recursive functions to build or accumulate a solution going down the recursion.
- Understand constant, linear, quadratic, and exponential running times.
- Understand how to analyze basic recursive code to determine its running time.
- Understand how accumulative recursion may allow for substantial efficiency gains.
- Understand how to test accumulative recursive code.