Generative and accumulative recursion

Readings: Sections 25, 26, 27, 30, 31

- Some subsections not explicitly covered in lecture
- Section 27.2 technique applied to strings

What is generative recursion?

Structural recursion, which we have been using so far, is a way of deriving code whose form parallels a data definition.

Generative recursion is more general: the recursive cases are **generated** based on the problem to be solved.

The non-recursive cases also do not follow from a data definition.

It is much harder to come up with such solutions to problems.

It often requires deeper analysis and domain-specific knowledge.

Example revisited: GCD

```
;; (euclid-gcd n m) computes gcd(n,m) using Euclidean algorithm
;; euclid-gcd: Nat Nat → Nat
(define (euclid-gcd n m)
  (cond [(zero? m) n]
        [else (euclid-gcd m (remainder n m))]))
```

Why does this work?

Correctness: Follows from Math 135 proof of the identity.

Termination: An application terminates if it can be reduced to a value in finite time.

All of our functions so far have terminated. But why?

For a non-recursive function, it is easy to argue that it terminates, assuming all applications inside it do.

It is not clear what to do for recursive functions.

Termination of recursive functions

Why did our structurally recursive functions terminate?

A structurally recursive function always makes recursive applications on smaller instances, whose size is bounded below by the base case (e.g. the empty list).

We can thus bound the **depth** of recursion (the number of applications of the function before arriving at a base case).

As a result, the evaluation cannot go on forever.

(sum-list (list 3 6 5 4)) $\Rightarrow$

(+ 3 (sum-list (list 6 5 4))) $\Rightarrow$

(+ 3 (+ 6 (sum-list (list 5 4)))) $\Rightarrow \ldots$

The depth of recursion of any application of sum-list is equal to the length of the list to which it is applied.

For generatively recursive functions, we need to make a similar argument.

In the case of euclid-gcd, our measure of progress is the size of the second argument.

If the first argument is smaller than the second argument, the first recursive application switches them.

After that, the second argument is always smaller than the first argument in any recursive application.

The second argument always gets smaller in the recursive application (since $m > n \bmod m$), but it is bounded below by 0.

Thus any application of euclid-gcd has a depth of recursion bounded by the second argument.

In fact, it is always much faster than this.

Termination is sometimes hard

```
;; collatz: Nat → Nat
;; requires: n >= 1
(define (collatz n)
  (cond [(= n 1) 1]
        [(even? n) (collatz (/ n 2))]
        [else (collatz (+ 1 (* 3 n)))]))
```

It is a decades-old open research problem to discover whether or not (collatz n) terminates for all values of n.

We can see better what collatz is doing by producing a list.

```
;; (collatz-list n) produces the list of the intermediate
;;     results calculated by the collatz function.
;; collatz-list: Nat → (listof Nat)
;; requires: n >= 1
(check-expect (collatz-list 1) '(1))
(check-expect (collatz-list 4) '(4 2 1))
(define (collatz-list n)
  (cons n (cond [(= n 1) empty]
                [(even? n) (collatz-list (/ n 2))]
                [else (collatz-list (+ 1 (* 3 n)))])))
```

Hoare's Quicksort

The Quicksort algorithm is an example of **divide and conquer**:

- divide a problem into smaller subproblems;
- recursively solve each one;
- combine the solutions to solve the original problem.

Quicksort sorts a list of numbers into non-decreasing order by first choosing a **pivot** element from the list.

The subproblems consist of the elements less than the pivot, and those greater than the pivot.

If the list is (list 9 4 15 2 12 20), and the pivot is 9, then the subproblems are (list 4 2) and (list 15 12 20).

Recursively sorting the two subproblem lists gives (list 2 4) and (list 12 15 20).

It is now simple to combine them with the pivot to give the answer.

(append (list 2 4) (list 9) (list 12 15 20))

The easiest pivot to select from a list lon is (first lon).

A function which tests whether another item is less than the pivot is (lambda (x) (< x (first lon))).

The first subproblem is then

(filter (lambda (x) (< x (first lon))) lon).

A similar expression will find the second subproblem (items greater than the pivot).

```
;; (quick-sort lon) sorts lon in non-decreasing order
;; quick-sort: (listof Num) → (listof Num)
(define (quick-sort lon)
  (cond [(empty? lon) empty]
        [else (local
                [(define pivot (first lon))
                 (define less (filter (lambda (x) (< x pivot)) (rest lon)))
                 (define greater (filter (lambda (x) (>= x pivot)) (rest lon)))]
                (append (quick-sort less) (list pivot) (quick-sort greater)))]))
```

Termination of quicksort follows from the fact that both subproblems have fewer elements than the original list (since neither contains the pivot).

Thus the depth of recursion of an application of quick-sort is bounded above by the number of elements in the argument list.

This would not have been true if we had mistakenly written

```
(filter (lambda (x) (>= x pivot)) lon)
```

The teaching languages function quicksort (note no hyphen) consumes two arguments, a list and a comparison function.

(quicksort '(1 5 2 4 3) <) ⇒ '(1 2 3 4 5)

(quicksort '(1 5 2 4 3) >) ⇒ '(5 4 3 2 1)

Intuitively, quicksort works best when the two recursive function applications are on arguments about the same size.

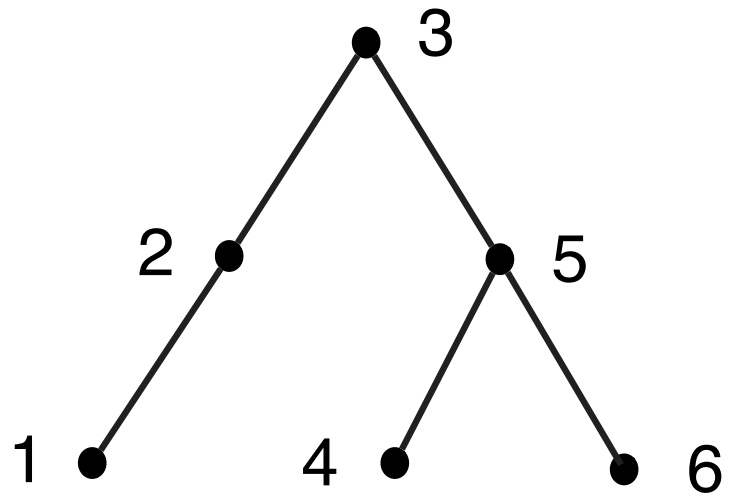
When one recursive function application is always on an empty list (as is the case when quick-sort is applied to an already-sorted list), the pattern of recursion is similar to the worst case of insertion sort, and the number of steps is roughly proportional to the square of the length of the list.

We will go into more detail on efficiency considerations in CS 136.

Here is another way of looking at quicksort.

Consider taking a list and inserting each element into an initially-empty binary search tree as a key with some dummy value, using an accumulator so that the first element of the list is the first inserted.

'(3 2 5 1 6 4)



```
(define (list→bst lon)
  (local [(define (list→bst/acc lon bst-so-far)
            (cond [(empty? lon) bst-so-far]
                  [else (list→bst/acc
                         (rest lon)
                         (add-to-bst (first lon)
                                     "dummy"
                                     bst-so-far))]))]
    (list→bst/acc lon empty)))
```

Inserting the elements of the list in order uses structural recursion on the list, with an accumulator for the BST.

Each insertion uses structural recursion on the BST.

We can recover the original list, but in sorted order, by flattening the tree in such a manner that the key of the root node comes between the flattened result of the left tree and the flattened result of the right tree (instead of before both).

This is called an **inorder traversal**, and computing it also uses structural recursion on the BST.

```
;; (bst-inorder t) flattens a BST into a sorted list
;;      with root between left and right
;; bst-inorder: BST → (listof Num)
(define (bst-inorder t)
  (cond [(empty? t) empty]
        [else (append (bst-inorder (node-left t))
                      (list (node-key t))
                      (bst-inorder (node-right t)))]))
```

```
(define (treesort lon)
  (bst-inorder (list→bst lon)))
```

Quicksort can be viewed as simply eliminating the trees from treesort.

The first key inserted into the BST, which ends up at the root, is the pivot.

(bst-inorder (node-left t)) is exactly the list of items of value less than the pivot.

Quicksort takes a number of structurally-recursive computations and makes them more efficient by using generative recursion.

Modifying the design recipe

The design recipe becomes much more vague when we move away from data-directed design.

The purpose statement should now specify **how** the function works, as well as what it does.

Examples need to illustrate the workings of the algorithm.

The template is less useful. Typically there are tests for the easy cases that don't require recursion, followed by the formulation and recursive solution of subproblems, and then combination of the solutions.

Example: breaking strings into lines

Traditionally, the character set used in computers has included not only alphanumeric characters and punctuation, but “control” characters as well.

An example in Racket is #\newline, which signals the start of a new line of text.

The characters ‘\’ and ‘n’ appearing consecutively in a string constant are interpreted as a single newline character.

For example, the string "ab\ncd" is a five-character string with a newline as the third character.

A line is a maximal sequence of non-newline characters.

Text files often contain embedded newlines to split the file into lines.

The same can be true of strings entered by a user or displayed.

We will develop a function to break a string into lines.

```
(check-expect (string→strlines "abc\ndef") (list "abc" "def"))
```

What should the following produce?

```
(string→strlines "abc\ndef\nhig")
(string→strlines "abc\n\nhig")
(string→strlines "\nabc")
(string→strlines "")
(string→strlines "\n")
(string→strlines "\n\n")
```

Internally, string→strlines will need to convert its argument into a list of characters using string→list. We'll apply a helper function, list→lines, to that list.

What should list→lines produce?

One natural solution is a list, where each item is a list of characters that make up one line in string→strlines's result.

```
;; (list→lines loc) turns loc into list of char lists based on newlines
;; list→lines: (listof Char) → (listof (listof Char))
;; Example:
(check-expect (list→lines '(#\a #\b #\newline #\c #\d))
              '((#\a #\b) (#\c #\d)))
```

Thus string→strlines can be implemented as

```
;; string→strlines: Str → (listof Str)
(define (string→strlines str)
    (local
        ;; list→lines: (listof Char) → (listof (listof Char))
        [(define (list→lines loc)
                 ...)]
    (map list→string
         (list→lines (string→list str)))))
```

We can use structural recursion on lists to write list→lines, but as we will see, the resulting code is awkward.

Rewriting the usual list template just a bit so we can refer to the recursive application several times without recomputing it gives us:

```
(define (list→lines loc)
  (cond [(empty? loc) ...]
        [else
         (local [(define r (list→lines (rest loc)))]
           (... (first loc) ... r ...))]))
```

If loc is nonempty and has #\newline as its first character, we cons an empty line onto the recursive result r.

If loc has, say, #\a as its first character, then this should be put at the front of the first line in the recursive result r.

But what if r is empty and has no first line?

```
(define (list→lines loc)
  (cond [(empty? loc) empty]
        [else (local [(define r (list→lines (rest loc)))]
                (cond [(char=? (first loc) #\newline)
                       (cons empty r)]
                      [(empty? r)
                       (list (list (first loc)))]
                      [else (cons (cons (first loc) (first r))
                                  (rest r))]))]))
```

This approach works, but is pretty low-level and hard to reason about. If we reformulate the problem at a higher level, it turns out to be easier to reason about – but requires generative recursion to implement.

Instead of viewing the data as a sequence of characters, what if we view it as a sequence of lines?

```
(define (list→lines loc)
  (cond [(empty? loc) ...]
        [else (... (first-line loc) ...
                   (list→lines (rest-of-lines loc)) ...)]))
```

It's easy to fill in the "sequence of lines template".

```
;; list→lines: (listof Char) → (listof (listof Char))
(define (list→lines loc)
  (cond [(empty? loc) empty]
        [else (cons (first-line loc)
                    (list→lines (rest-of-lines loc)))]))
```

The two helper functions are also simpler, using structural recursion on lists.

```
;; (first-line loc) produces longest newline-free prefix of loc
;; first-line: (listof Char) → (listof Char)
;; Examples:
  (check-expect (first-line empty) empty)
  (check-expect (first-line '(#\newline)) empty)
  (check-expect (first-line '(#\a #\newline)) '(#\a))

(define (first-line loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) empty]
        [else (cons (first loc) (first-line (rest loc)))]))
```

```
;; (rest-of-lines loc) produces loc with everything
;;     up to and including the first newline removed
;; rest-of-lines: (listof Char) → (listof Char)
;; Examples:
(check-expect (rest-of-lines empty) empty)
(check-expect (rest-of-lines '(#\newline)) empty)
(check-expect (rest-of-lines '(#\a #\newline)) empty)
(check-expect (rest-of-lines '(#\a #\newline #\b)) '(#\b))
```

```
(define (rest-of-lines loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) #\newline) (rest loc)]
        [else (rest-of-lines (rest loc))]))
```

Generalizing string→strlines

We can generalize string→strlines to accept a predicate that specifies when to break the string. We will call the pieces **tokens** instead of lines.

```
;; string→tokenstrs: (Char → Bool) Str → (listof Str)
(define (string→tokenstrs break? str)
  (map list→string
       (list→tokens break? (string→list str))))
(check-expect
  (string→tokenstrs char-whitespace? "One two\nthree")
  (list "One" "two" "three"))
```

```
;; string→tokenstrs: (Char → Bool) Str → (listof Str)
(define (string→tokenstrs break? str)
  (local [(define (first-token loc)
            (cond [(empty? loc) empty]
                  [(break? (first loc)) empty]
                  [else (cons (first loc) (first-token (rest loc)))]))

          (define (remove-token loc)
            (cond [(empty? loc) empty]
                  [(break? (first loc)) (rest loc)]
                  [else (remove-token (rest loc))]))
```

```
;; list→tokens: (listof Char) → (listof (listof Char))
(define (list→tokens loc)
  (cond [(empty? loc) empty]
        [else (cons (first-token loc)
                    (list→tokens (remove-token loc)))]))
] ; end of local
(map list→string
     (list→tokens (string→list str)))))
```

As an example, consider breaking a string up into “words” separated by **whitespace** (spaces, newlines).

In this case, we can use the built-in predicate char-whitespace?.

```
(string→tokenstrs char-whitespace? "This is\na test.")
⇒ '("This" "is" "a" "test.")
```

Tokenizing is especially important in interpreting or **parsing** a computer program or other specification. For instance,

```
(define (square x)
  (* x x))
```

can be broken into twelve **tokens**.

Here, our simple method of using a character predicate will not work.

In later courses, we will discuss tokenizing in order to deal with situations such as a string representing a computer program, a Web page, or some other form of structured data.

More higher-order functions

Because generative recursion is so general, it is difficult to abstract it into one function.

However, we can create higher-order functions for certain patterns.

One example is accumulative recursion, which combines structural recursion with the updating of accumulators.

We need to be a little more specific: structural recursion on a list with one accumulator.

```
;; code from lecture module 9

(define (sum-list lon)
  (local [(define (sum-list/acc lst sofar)
            (cond [(empty? lst) sofar]
                  [else (sum-list/acc (rest lst)
                                      (+ (first lst) sofar))]))]
    (sum-list/acc lon 0)))
```

```
;; code from lecture module 7 rewritten to use local

(define (my-reverse lst0)
  (local [(define (my-rev/acc lst acc)
            (cond [(empty? lst) acc]
                  [else (my-rev/acc (rest lst)
                                    (cons (first lst) acc))]))]
    (my-rev/acc lst0 empty)))
```

The differences between these two functions are:

- the initial value of the accumulator;
- the computation of the new value of the accumulator, given the old value of the accumulator and the first element of the list.

```
(define (my-foldl combine base lst0)
  (local [(define (foldl-acc lst acc)
            (cond [(empty? lst) acc]
                  [else (foldl-acc (rest lst)
                                   (combine (first lst) acc))]))]
    (foldl-acc lst0 base)))

(define (sum-list lon) (my-foldl + 0 lon))
(define (my-reverse lst) (my-foldl cons empty lst))
```

We noted earlier that intuitively, the effect of the application

```
(foldr f b (list x1 x2 . . . xn))
```

is to compute the value of the expression

```
(f x1 (f x2 (. . . (f xn b) . . . )))
```

What is the intuitive effect of the following application of foldl?

```
(foldl f b (list x1 . . . xn-1 xn))
```

The function foldl is provided in Intermediate Student.

What is the contract of foldl?

Goals of this module

You should understand the idea of generative recursion, why termination is not assured, and how a quantitative notion of a measure of progress towards a solution can be used to justify that such a function will return a result.

You should understand the examples given, particularly quicksort, line breaking, and tokenizing using a general break predicate.

You should be comfortable with accumulative recursion, and able to write accumulatively recursive helper functions with associated wrapper functions to hide the accumulator(s) from the user.