Generative recursion

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

Topics:

- What is generative recursion?
- Termination
- Hoare's Quicksort
- Modifying the design recipe
- Example: breaking strings into lines

What is generative recursion?

Simple and accumulative 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 functions using simple recursion terminate?

A simple 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)) ⇒
(+ 3 (sum-list (list 6 5 4))) ⇒
(+ 3 (+ 6 (sum-list (list 5 4)))) ⇒ ...
```

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, which makes the second argument smaller.

After that, the second argument is always smaller than the first argument in any recursive application, due to the application of the remainder modulo m.

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)
```

instead of the correct

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

In the teaching languages, the built-in function quicksort (note no hyphen) consumes two arguments, a list and a comparison function.

(quicksort '(1 5 2 4 3) $<$) $\Rightarrow$ '(1 2 3 4 5)

(quicksort '(1 5 2 4 3) $>$) $\Rightarrow$ '(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.

Modifying the design recipe

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

The purpose statement remains unchanged, but additional documentation is often required to describe **how** the function works.

Examples need to illustrate the workings of the algorithm.

We cannot apply a template, since there is no data definition.

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. It would typically be printed as "ab" on one line and "cd" on the next line.

Consider converting a string such as "one\ntwo\nthree" into a list of strings, (list "one" "two" "three"), one for each line.

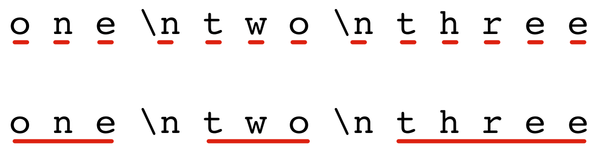
The solution will start with an application of string→list. That’s the only way we’ve studied of working with individual characters in a string.

This problem can be solved using simple recursion on the resulting list of characters – but it’s hard. The “simple” recursion gets bogged down in a lot of little details.

In this case a generative solution is easier.

The generative idea

Instead of thinking of the list of characters as a list of characters, think of it as a list of lines:



A list of lines is either empty or a line followed by a list of lines.

Start with helper functions that divide the list of characters into the first line and the rest of the lines.

```
;; (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 '(#\a #\newline)) '(#\a))
(check-expect (first-line (string→list "abc\ndef")) '(#\a #\b #\c))

(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 '(#\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))]))
```

We can create a “list of lines template” using these helpers.

```
(define (loc→lol loc)
  (local
    [(define fline (first-line loc))
     (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
          [else . . . fline . . . (loc→lol rlines) . . . ])))
```

```
;; list→lines: (listof Char) → (listof Str)
(check-expect (loc→lol (string→list "abc\ndef")) (list "abc" "def"))
(check-expect (loc→lol (string→list "")) (list))
(check-expect (loc→lol (string→list "\ndef")) (list "" "def"))

(define (loc→lol loc)
  (local [(define fline (first-line loc))
          (define rlines (rest-of-lines loc))]
    (cond [(empty? loc) empty]
          [else (cons (list→string fline)
                      (loc→lol rlines))])))
```

Generative recursion

Why is this generative recursion?

loc→lol can be rewritten as

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

The recursive call to loc→lol is *not* using the data definition for a list of characters. It often gets many steps closer to the base case in one recursive application.

It *is* using a data definition of a “list of lines”, but that’s a higher-level abstraction that we imposed on top of the (listof Char), our actual argument.

The key part of the generative recursion pattern is that the argument to loc→lol is being generated by rest-of-lines.

With generative recursion we needed that “aha” that transformed the problem into a list of lines. That “aha” is often difficult to see.

Was it worth it? Consider the solution using “simple” recursion on the next slide. This still needs a wrapper function to do both pre- and post-processing.

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

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.