Generative recursion

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

Topics:

• What is generative recursion? [2–4]
• Termination [5–10]]
• Hoare’s Quicksort [11-17]
• Modifying the design recipe [18]
• Example: breaking strings into lines [19-28]
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))]
        [else (euclid-gcd m (remainder n m))]))

E.g.
gcd(100, 85) ⇒ gcd(85, 15) ⇒ gcd(15, 10) ⇒ gcd(10, 5)
⇒ gcd(5, 0) ⇒ 5
Why does this work?

**Correctness:** Follows from Math 135 proof of the identity.

**Key Idea:** **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 Racket 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?

Key Observation: 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.
Example: termination for simple recursion

\[
\text{(sum-list (list 3 6 5 4))} \Rightarrow \\
(\text{+} 3 \text{(sum-list (list 6 5 4))}) \Rightarrow \\
(\text{+} 3 (\text{+} 6 \text{(sum-list (list 5 4))})) \Rightarrow \ldots
\]

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

Key Point: For generatively recursive functions, we need to make a similar argument.
Example: termination of **euclid-gcd**

The function **euclid-gcd** terminates for any natural number.

*Key Idea:* 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 $m$. 
Example: termination of euclid-gcd

The *second argument always gets smaller* in the recursive application (since $m > n \mod m$), 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)))]))

Key Point: It is a decades-old open research problem (i.e. we currently don’t know the answer) to discover whether or not (collatz n) terminates for all values of n.
We can see better what \texttt{collatz} is doing by producing a list.

\begin{verbatim}
;; (collatz-list n) produces the list of the intermediate
;; results calculated by the collatz function.
;; collatz-list: Nat \rightarrow (listof Nat)
;; requires: n \geq 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)))])))
\end{verbatim}
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 or equal to* the pivot (or just greater than the pivot if duplicated values are not allowed).
Quicksort: example

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

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

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

\((\text{append } (\text{list } 2 \ 4) \ (\text{list } 9) \ (\text{list } 12 \ 15 \ 20))\)
Quicksort: selecting the pivot

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 or equal to the pivot).
quick-sort implementation

;; (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)))]))
Quicksort: termination

Quicksort terminates because *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

\[
\text{(filter (lambda (x) (> = x pivot)) lon)}
\]

instead of the correct

\[
\text{(filter (lambda (x) (> = x pivot)) (rest lon))}
\]
Quicksort in Racket

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) <) ⇒ '(1 2 3 4 5)
(quicksort '(1 5 2 4 3) >) ⇒ '(5 4 3 2 1)
```
Quicksort efficiency

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.

*Note:* 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: new lines

The character sets used in computers include “control” characters.

In Racket the character `#\newline`, signals the start of a new line of text.

In a string ‘\’ and ‘n’ appearing consecutively in a string constant are interpreted as a single newline character (i.e. as `#\newline`).

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.
New line: need for generative recursion

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

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

```
one
two
three
```
```
one
two
three
```

The string version is "\n" and the char version is #\newline.

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

So start with helper functions first-line that divide the list of characters into the first line and the rest of the lines.
Example: breaking strings into lines

- **As a string** the newline is written as “\n”, e.g. “ab\ncd”.
- **As a list of characters** the newline is written as \newline, e.g. (list a b \newline c d).
- Slide 23: The definition of **first-line**, i.e. produce the **first line** from the list of characters, loc.
- Slide 24: The definition of **rest-of-lines**, i.e. produce the **rest of the lines** from the list of characters, loc.
- Slide 25: The template that uses **first-line** and **rest-of-lines**.
- Slide 26: The function that uses **first-line** and **rest-of-lines** to break up the list of characters into a list of lines.
- Slide 29: a simple recursion version.
(define (first-line loc)
  (cond [(empty? loc) empty]
        [(char=? (first loc) newline) empty]
        [else (cons (first loc) (first-line (rest loc)))]))
(define (rest-of-lines loc)
    (cond [(empty? loc) empty]
          [(char=? (first loc) #\newline) (rest loc)]
          [else (rest-of-lines (rest loc))])))
List of lines template

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

\[
(\text{define } (\text{loc} \rightarrow \text{lol} \text{ loc})
\]
\[
(\text{local}
\]
\[
[(\text{define } \text{fline} (\text{first-line loc}))
\]
\[
(\text{define } \text{rlines} (\text{rest-of-lines loc}))]
\]
\[
(\text{cond } [(\text{empty? } \text{loc}) \text{ empty}]
\]
\[
[\text{else } \ldots \ \text{fline} \ldots (\text{loc} \rightarrow \text{lol} \text{ rlines}) \ldots ])]\)
;; 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.
Simple recursion version

(define (list→lines loc)
  (cond [(empty? loc) (list empty)] ; 2
    [(and (empty? (rest loc)) (char=? #\newline (first loc))) ; 3
      (list empty)] ; 4
    [else
      (local [(define r (list→lines (rest loc)))] ; 6
        (cond
          [(char=? (first loc) #\newline) (cons empty r)] ; 8
          [else (cons (cons (first loc) (first r)) ; 9
            (rest r))]])))) ;10

CS 135 Fall 2019 14: Generative recursion 28
Simple recursion version

**Key Idea:** Each time you see a newline start a new list to add the characters to.

E.g. for `(list \a \b \newline \c \d)`

- Base case: start with `(list empty)`.
- Cons d then c to the inner list, i.e. `(list (list c d))`.
- For newline, cons a new empty list c to the inner list, i.e. `(list empty (list c d))`.
- Cons b then a to the first inner list, i.e. `(list (list a b) (list c d))`.
Simple recursion version

• Line 2: Base case: create a list of lines with one empty line in it.

• Lines 3-4: If the last character is a newline, do the same as above.

• Line 6: define $r$ locally to avoid repeated recursive applications

• Line 8: if the char is a newline then cons a new line (i.e. a new list) to the list of lines.

• Lines 9-10: otherwise add the char, i.e. (first loc), to the first list in the list of lines.
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.
Module 14 Summary

Simple and Generative Recursion

1. Simple recursion is a way of deriving code whose form parallels a data definition. [2]

2. **Generative recursion** is more general: the recursive cases are generated as a function of the arguments. [2]

3. An application **terminates** if it can be reduced to a value in finite time. [4]

4. A simple recursive function always makes recursive applications on smaller instances, whose size is bounded below by the base case. [5]
Module 14 Summary

Generative Recursion

5. The depth of recursion is the number of recursive applications of the function until the base case is reached. [5]

6. For generative recursion, an argument has to be made that the recursion will terminate. [6]

7. Calculating the euclidean-gcd of two numbers is an example of generative recursion. [7–8]

8. It is not currently known if the collatz function terminates for all non-zero natural numbers. [9–10]
Module 14 Summary

Quicksort

9. A **divide and conquer** algorithm is an algorithm that
   (a) divides a problem into smaller subproblems;
   (b) recursively solves each one;
   (c) combines the solutions to solve the original problem. [11]

10. **quicksort** is a divide and conquer algorithm.

11. The subproblems are the elements less than the pivot and those
greater than or equal to the pivot. [11]

12. Termination occurs because both subproblems are smaller than
   the original list. [15]
Module 14 Summary

Generative Recursion and Breaking up a String

13. For generative recursion, design recipe often requires more documentation about how the function works. [18]

14. Generative recursion can be used to break a string up into a list of lines, i.e. \texttt{loc} \rightarrow \texttt{lol}. [19–27]