General Recursion

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 8.1

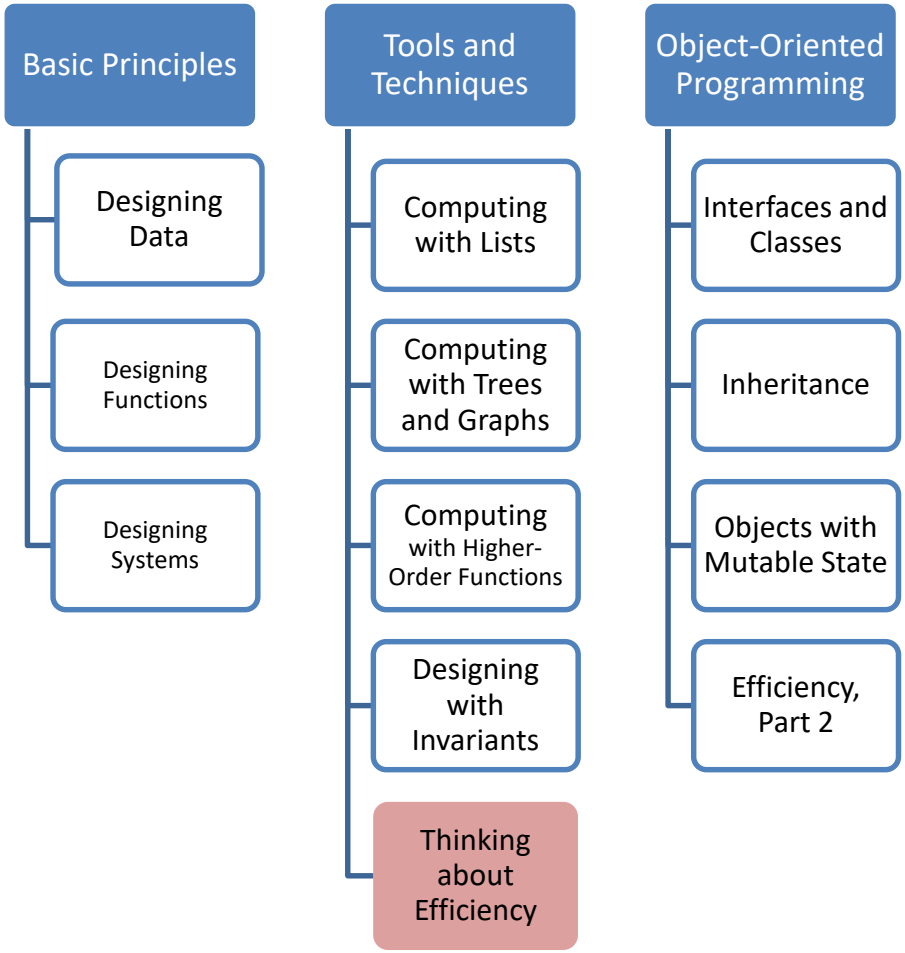
Module Introduction (1)

- This module covers two topics
- First, we talk about *general recursion*, in which our functions recur not on a sub-piece of the input data, but on a sub-problem of the original problem.
  - We talk about how to determine whether a sub-problem is simpler than the original, and how to document that fact in our design.
  - General Recursion and Invariants make a powerful combination

Module Introduction (2)

- Then, we talk about the important topic of *algorithmic complexity*.
  - We talk about how to describe the time your algorithm will take on a given input
  - We talk about what things are worth optimizing for efficiency
    - Spoiler alert: the answer is very, very few
- We'll also introduce more Java examples, to show you how these ideas apply to conventional programs with assignment statements.

Module 08


Basic Principles
Designing Data
Designing Functions
Designing Systems
Tools and Techniques
Computing with Lists
Computing with Trees and Graphs
Computing with Higher-Order Functions
Designing with Invariants
Thinking about Efficiency
Object-Oriented Programming
Interfaces and Classes
Inheritance
Objects with Mutable State
Efficiency, Part 2

General Recursion

- So far, we've written our functions using the observer template to recur on the sub-pieces of the data. We sometimes call this *structural recursion.*
- In this module, we'll see some examples of problems that don't fit neatly into this pattern.
- We'll introduce a new family of strategies, called *general recursion*, to describe these examples.
- General recursion and invariants together provide a powerful combination.

Structural Recursion

- Our observer templates always recurred on the sub-pieces of our structure.
- This is sometimes called *structural recursion*.
- But that's not the only way to use recursion.

Divide-and-Conquer (General Recursion)

- How to solve the problem:
  - If it's easy, solve it immediately
  - If it's hard:
    - Find one or more easier problems whose solutions will help you find the solution to the original problem.
    - Solve each of them
    - Then combine the solutions to get the solution to your original problem

An example: merge sort

- Let's turn to a different example: merge sort, which you should know from your undergraduate data structures or algorithms course.
- Divide the list in half, sort each half, and then merge two sorted lists.
- First we write **merge**, which merges two sorted lists.

But first, a data definition

```
;; A SortedList is a list of Reals,
;;  sorted by <.  Duplicates are
;;  allowed.
```

Just following the Recipe….

merge

```
;; merge : SortedList SortedList -> SortedList
;; RETURNS: the sorted merge of its two arguments
;; strategy: recur on (rest lst1) or (rest lst2)
(define (merge lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(< (first lst1) (first lst2))
     (cons (first lst1) (merge (rest lst1) lst2))]
    [else
     (cons (first lst2) (merge lst1 (rest lst2)))]))
```

If the lists are of length n, this function takes time proportional to **n**. We say that the time is O(**n**).

Why does this function halt?

- Our standard argument is: the input gets smaller at every recursive call, so eventually it can't get any smaller.
- But what's the "input" here? And what do we mean by "smaller"?
- "smaller" is easy: we are recurring on the **rest** of a list, so probably "smaller" should mean "smaller length"

Why does this function halt? (2)

- **(length lst2)** doesn't get smaller at every call
  - look at the first recursive call).
- **(length lst1)** doesn't get smaller at every call
  - look at the second recursive call
- But the *sum* of **(length lst1)** and **(length lst2)** does get smaller at every call
  - at each call, either **(length lst1)** or **(length lst2)** decreases by 1, so their sum is guaranteed to decrease by 1!

Halting Measure (1)

- Remember, part of design is getting knowledge out of our heads and on to a piece of paper. How do we document our knowledge about why our function halts?
- We document this knowledge as a *halting measure.*
- A halting measure is an integer-valued quantity that can't be less than zero, and which decreases at each recursive call in your function.
- The halting measure is a way of explaining how each of the subproblems are easier than the original.

Halting Measure (2)

- Since the measure is integer-valued, and it decreases at every recursive call, your function can't make more recursive calls than what the halting measure says.
- In particular, it must halt!

Possible halting measures

- the value of a NonNegInt argument
- the size of an s-expression
- the length of a list
- the number of elements of some set
- a non-negative integer quantity that depends on one of the quantities above

So for **merge**, we write:

```
;; merge : SortedList SortedList -> SortedList
;; merges its two arguments
;; strategy: recur on (rest lst1) or (rest lst2)
;; HALTING MEASURE: (length lst1) + (length lst2)
(define (merge lst1 lst2)
  (cond
    [(empty? lst1) lst2]
    [(empty? lst2) lst1]
    [(< (first lst1) (first lst2))
     (cons (first lst1) (merge (rest lst1) lst2))]
    [else
     (cons (first lst2) (merge lst1 (rest lst2)))]))
```

Checking the halting measure for **merge**

- Proposed halting measure:
  - **`(length lst1) + (length lst2)`**
- Justification:
  - **`(length lst1)`** and **`(length lst2)`** are both always non-negative, so their sum is non-negative.
  - At each recursive call, either **`lst1`** or **`lst2`** becomes shorter, so either way the sum of their lengths is shorter.
- So **`(length lst1) + (length lst2)`** is a halting measure for **merge**.

merge-sort

```
;; merge-sort : RealList -> SortedList
(define (merge-sort lst)
  (cond
    [(empty? lst) lst]
    [(empty? (rest lst)) lst]
    [else
      (local
       ((define evens (even-elements lst))
        (define odds  (odd-elements lst)))
       (merge
        (merge-sort evens)
        (merge-sort odds)))]))
```

Now we can write merge-sort. merge-sort takes its input and divides it into two approximately equal-sized pieces.

Depending on the data structures we use, this can be done in different ways. We are using lists, so the easiest way is to take every other element of the list, so the list **(10 20 30 40 50)** would be split into **(10 30 50)** and **(20 40)** .

We sort each of the pieces, and then merge the sorted results.

This is *really* different

- Merge-sort just did something very different from anything we've seen before: it recurred on two things, neither of which is **(rest lst)** .
- We recurred on
  - **`(even-elements lst)`**
  - **`(odd-elements  lst)`**
- Neither of these is a sublist of lst **.**
- But each of these is guaranteed to be shorter than lst.
  - Really?? Let's check it...

Is **(even-elements lst)** really always shorter than lst ?

- Hmm, we'd better look at even-elements and odd-elements a little more closely.
- We didn't write formal purpose statement for these functions, but we can look at some plausible examples:

Examples for even-elements and odd-elements

```
(even-elements (list 10 20 30 40))
  = (list 20 40)
```

We didn't specify whether the elements of the list should be counted from 0 or 1. Let's choose to count from 1.

```
(even-elements empty) = empty
```

No doubt about this one! But wait: this already falsifies our hypothesis that **(even-elements lst)** is always shorter than lst

When is **(even-elements lst)** shorter than **lst**?

- When **(even-elements lst)** and **(odd-elements lst)** are called, we know that **lst** has length at least 2.
- That means the first element of **lst** is NOT in **(even-elements lst)**. So **(even-elements lst)** is shorter than **lst**.
- Furthermore, the second element of lst is NOT in **(odd-elements lst)** . So **(odd-elements lst)** is shorter than **lst**.
- Summary: if **(length lst)** ≥ 2, **(even-elements lst)** and **(odd-elements lst)** are both strictly shorter than **lst**
- Luckily, that's all we need!

Also need to confirm that our implementations of **odd-elements** and **even-elements** satisfies this. (Demonstration deferred)

Halting measure for **`merge-sort`**

- Proposed halting measure: **`(length lst)`**
- Justification of halting measure:
  - **`(length lst)`** is always a non-negative integer.
  - At each recursive call, **`(length lst)`** ≥ 2
  - If **`(length lst)`** ≥ 2, then
    `(length (even-elements lst))` and
    `(length (odd-elements lst))`
    are both ***strictly less*** than **`(length lst)`.**
    - **`[As shown on Preceding slide]`**
  - So **`(length lst)`** is a halting measure for **`merge-sort`**.

Running time for merge sort

- Splitting the list in this way takes time proportional to the length n of the list. The call to merge likewise takes time proportional to **n**. We say this time is **O(n)**.
- If **T(n)** is the time to sort a list of length **n**, then **T(n)** is equal to the time **2*T(n/2)** that it takes to sort the two sublists, plus the time **O(n)** of splitting the list and merging the two results:
- So the overall time is

$$\mathbf{T(n) = 2*T(n/2) + O(n)}$$

- When you take algorithms, you will learn that all this implies that **T(n) = O(n log n).** This is better than a selection sort, which takes **O(n^2)**.
- This is all for the worst case: we will talk about all this more precisely in the second half of this module.

A Numeric Example

```
fib : NonNegInt -> NonNegInt
(define (fib n)
  (cond
   [(= n 0) 1]
   [(= n 1) 1]
   [else (+ (fib (- n 1))
            (fib (- n 2)))]))
```

Here's the standard recursive definition of the fibonacci function

A Numeric Example (2)

```
fib : NonNegInt -> NonNegInt
(define (fib n)
  (cond
    [(= n 0) 1]
    [(= n 1) 1]
    [else (+ (fib (- n 1))
             (fib (- n 2)))]))
```

Let's check to see that the recursive calls obey the contract.

When we get to the recursive calls, if **n** is a NonNegInt, and it is not 0 or 1, then it must be greater than or equal to 2, so **n-1** and **n-2** are both NonNegInt's.

So the recursive calls don't violate the contract.

Halting measure for **fib**

- Proposed halting measure: **n**
- Justification for halting measure:
  - **n** is always a non-negative integer (by the contract)
  - At each recursive call, **n-1** and **n-2** are both non-negative integers, and each is strictly smaller than **n.** So **n** decreases at each recursive call.
- So **n** is a halting measure for fib.

What about (fib -1)?

```
(fib -1)
= (+ (fib -2) (fib -3))
= (+ (+ (fib -3) (fib -4))
     (+ (fib -4) (fib -5))
= etc.
```

Oops! This doesn't terminate!

What does this tell us?

- First, it tells us that using general recursion we can write functions that may not terminate.
- Is there something wrong with our termination argument?
- No, because the termination argument only says what happens when **n** is a NonNegInt
- -1 is a contract violation, so anything could happen.
- If we want to make the contract Int -> Int , then we need to document the non-termination behavior:

Documenting non-termination

```
fib : Integer -> Integer
Halting Measure:
  If n is non-negative, then n is a
halting measure.
  If n is negative, the function
fails to halt.
```

What do I need to deliver?

- You must write down a halting measure for each function that uses general recursion.
- You don't have to write down a justification for halting measure but you should be prepared to explain it at codewalk.
- If your function does not terminate on some input problems, you should write down a description of the inputs on which your program fails to halt.

Wait, isn't that a lot of work?

- Most of your functions will recur on a substructure of the input data. We call this structural recursion.
- If you just use structural recursion, you don't need to supply a halting measure, because structural recursions always halt. (See Lesson 5.5)

Most of the time, identifying the halting measure is easy

- It's usually something like
  - "The value of **n**" (a NonNegInt)
  - "the length of **lst**"
  - "the size of the unknown region" (see Lesson 8.3 on Binary Search)
- Only rarely will be it be something more complicated.

Structural Recursion vs. General Recursion

`(... (f (rest lst)))` is structural

`(f (... (rest lst)))` is general

You can usually tell just from the function definition whether it is structural or general recursion.

In the first example here, **f** is called on **(rest lst)**, which is a component of the list, and is therefore smaller than **lst**. This is what the observer template for lists tells us.

In the second example, **f** is being called some other value that happens to be computed from **(rest lst)**, but that's not the same as **(rest lst)**. So this example is general recursion. There's no telling how big **`(... (rest lst))`** is. If we call **f** on it, we'd better have a halting measure and a justification to ensure that the measure of **`(... (rest lst))`** is smaller than the measure of **lst** .

How to write down the design strategy

You can write down a general-recursion strategy as something like

STRATEGY: Recur on <value>

or

STRATEGY: Recur on <value>; halt when <condition>

or

STRATEGY: Recur on <values>; <describe how answers are combined>

These are just patterns; in general, a strategy is a tweet-sized description of how the function works. At this point in the course, we'll give you a lot of freedom in doing this. There's no hard-and-fast right and wrong for these: the question is whether the description is likely to be useful to the reader.

What do I need to deliver?

- You must write down a halting measure for each function that uses general recursion.
- You don't have to write down justification for your halting measure, but you should be prepared to explain it at codewalk.
- If your function does not terminate on some input problems, you should write down a description of the inputs on which your program fails to halt.

Lesson Summary

- We've introduced *general recursion,* also known as *divide-and-conquer* .
- In general recursion, we solve the problem by combining solutions to easier subproblems.
- In each use of general recursion, you must propose a *halting measure* that documents the "difficulty" of each instance of the problem.
- You must be able to justify the proposed halting measure by explaining why the measure of each subproblem is smaller than the measure of the original problem.
- Structural decomposition is a special case where the data definition guarantees the subproblem is easier, so it’s not necessary to document a halting measure.

Next Steps

- Study the files 08-1-merge-sort.rkt in the Examples folder.
- Do Guided Practices 8.1 and 8.2
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson