Introducing General Recursion

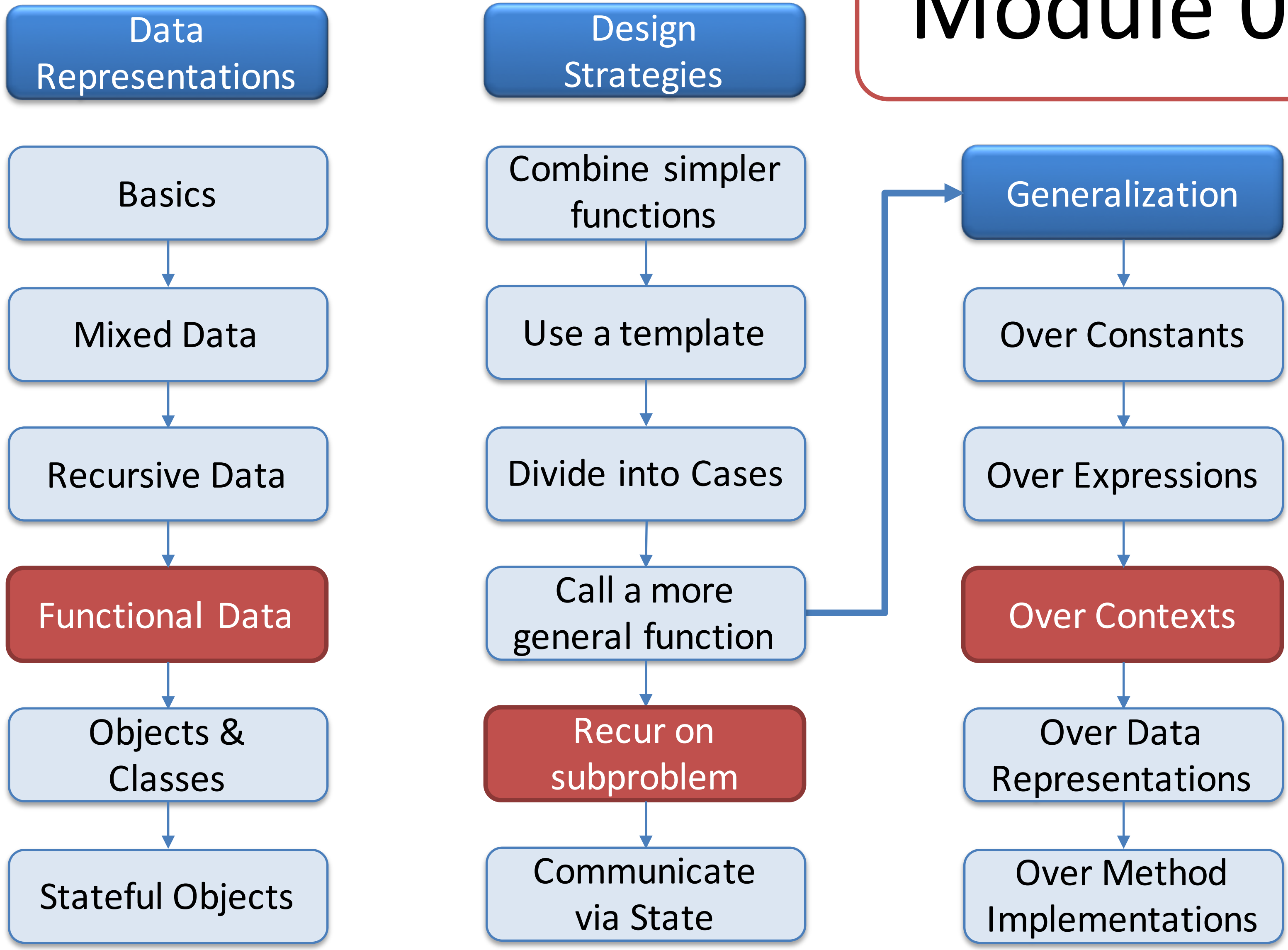
CS 5010 Program Design Paradigms "Bootcamp"

Lesson 8.1

Module Introduction

- So far, we've written our functions using the destructor 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.

Module 08


Data Representations
Basics
Mixed Data
Recursive Data
Functional Data
Objects & Classes
Stateful Objects
Design Strategies
Combine simpler functions
Use a template
Divide into Cases
Call a more general function
Recur on subproblem
Communicate via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

Structural Recursion

- Our destructor templates always recurred on the sub-pieces of our structure.
- We call this *structural recursion.*
- Let's look at an example that doesn't fit into this mold.

An example: **decode**

```
(define-struct diffexp (exp1 exp2))

;; A DiffExp is either
;; -- a Number
;; -- (make-diffexp DiffExp DiffExp)
```

Here is the data definition for diffexps. These are a simple representation of difference expressions, much like the arithmetic expressions we considered in some of the earlier problem sets.

Examples of diffexps

```
(make-diffexp 3 5)
(make-diffexp 2 (make-diffexp 3 5))
(make-diffexp
  (make-diffexp 2 4)
  (make-diffexp 3 5))
```

Writing out diff-exps is tedious at best.

Not very human-friendly...

- How about using more Scheme-like notation, eg:

```
(- 3 5)
(- 2 (- 3 5))
(- (- 2 4) (- 3 5))
```

Task: convert from human-friendly notation to diffexps.

- Info analysis:
  - what's the input?
  - answer: S-expressions containing numbers and symbols

Data Definitions

Here is a formal data definition for the inputs to our function.

```
;; An Atom is one of
;; -- a Number
;; -- a Symbol

;; An SexpOfAtom is either
;; -- an Atom
;; -- a ListOfSexpOfAtom

;; A ListOfSexpOfAtom is either
;; -- empty
;; -- (cons SexpOfAtom ListOfSexpOfAtom)
```

Templates

And the templates that go with it.

```
(define (sexp-fn sexp)
  (cond
    [(atom? sexp) (... sexp)]
    [else (... (los-fn sexp))]))

(define (los-fn los)
  (cond
    [(empty? los) ...]
    [else (... (sexp-fn (first los))
               (los-fn (rest los)))]))
```

Contract and Examples

```
decode : SexpOfAtom -> DiffExp

(- 3 5) => (make-diffexp 3 5)
(- 2 (- 3 5)) => (make-diffexp
                   2
                   (make-diffexp 3 5))
(- (- 2 4) (- 3 5))
  => (make-diffexp
       (make-diffexp 2 4)
       (make-diffexp 3 5))
```

Umm, but not every SexpOfAtom corresponds to a diffexp

```
(- 3)                does not correspond to any diffexp
(+ 3 5)              does not correspond to any diffexp
(- (+ 3 5) 5)        does not correspond to any diffexp
((1))                does not correspond to any diffexp
((- 2 3) (- 1 0))    does not correspond to any diffexp
(- 3 5 7)            does not correspond to any diffexp
```

But here are some other inputs that are legal inputs according to our contract. None of these is the human-friendly representation of any diff-exp.

A Better Contract

```
;; A MaybeX is one of
;; -- false
;; -- X

;; (define (maybex-fn mx)
;;   (cond
;;     [(false? mx) ...]
;;     [else (... mx)]))

decode
  : SexpOfAtom -> MaybeDiffExp
```

To account for this, we change our contract to produce a **MaybeDiffExp** instead of a **DiffExp**.
If the **SexpOfAtom** doesn't correspond to any **DiffExp**, we'll have our decode function return **false**.

Function Definition (1)

```
;; decode : SexpOfAtom -> MaybeDiffExp

;; Algorithm: if the top level of the sexp could be the top level of some
;;  diffexp, then recur, otherwise return false.
;;  If either recursion fails, return false.  If both recursions succeed,
;;  return the diffexp.

(define (decode sexp)
  (cond
    [(number? sexp) sexp]
    [(looks-like-diffexp? sexp)
     (local
       ((define operand1 (decode (second sexp)))
        (define operand2 (decode (third sexp))))
       (if (and (succeeded? operand1)
                (succeeded? operand2))
           (make-diffexp operand1 operand2)
           false))]
    [else false]))
```

Now we can write the function definition.

Function Definition (2)

```
;; looks-like-diffexp? : SexpOfAtom -> Boolean
;; WHERE: sexp is not a number.
;; RETURNS: true iff the top level of the sexp could be the top
;;  level of some diffexp.
;; At the top level, a representation of a
;; diffexp must be either a number or a list of
;; exactly 3 elements, beginning with the symbol -
;; STRATEGY: combine simpler functions
(define (looks-like-diffexp? sexp)
  (and
   (list? sexp)
   ;; at this point we know that
   ;; sexp is a list
   (= (length sexp) 3)
   (equal? (first sexp) '-)))
```

In this function definition, we add an invariant (the **WHERE** clause) to record the assumption that our input is not merely an **SexpOfAtom**, but is rather an **SexpOfAtom** that is not a number. We know this assumption is true, because **looks-like-diffexp?** is only called after **number?** fails.

Function Definition (3)

```
;; succeeded? : MaybeX -> Boolean
;; RETURNS: Is the argument an X?
;; strategy: Use the template for MaybeX
(define (succeeded? mx)
  (cond
    [(false? mx) false]
    [else true]))
```

And we finish with the help function **succeeded?** .

Something new happened here

- We recurred on the subpieces, but
  - we didn't use the predicates from the template
  - we didn't recur on all of the subpieces
- This is not structural recursion following the template.
- It's "divide-and-conquer"
- We call this *general 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
- Here it is as a template:

Here the subproblems are easier because they are pieces of the original structure.
We'll talk more about what "easier" means in Lesson 8.2

Pattern for General Recursion (1)

```
;; solve : Problem -> Solution
;; purpose statement...


(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (local
       ((define solution1
          (solve (simpler-instance1 the-problem)))
        (define solution2
         (solve (simpler-instance2 the-problem))))
       (combine-solutions solution1 solution2))]))
```

Instead of using ellipses ("..."'s), we've give each slot a name (displayed in **orange**) so you can see the role it plays.

There is no magic recipe for finding easier subproblems. You must understand the structure of the problem domain.

There's more than one pattern

- The pattern might take different shapes, depending on our problem.
- We might have different numbers of trivial cases, or different numbers of subproblems.
- Let's write this down as a recipe, and then look at some of the possibilities.

The General Recursion Recipe

Question	Answer
1. Are there different cases of your problem, each with a different kind of solution?	Write a **cond** with a clause for each case.
2. How do the cases differ from each other?	Use the differences to formulate a condition per case
3. For each case:	a. Identify one or more instances of your problem that are simpler than the original. b. Document why they are simpler c. Extract each instance and recur to solve it. d. Combine the solutions of your easier instances to get a solution to your original problem.

Writing down your strategy

We’ll write down our strategies as things like

```
  STRATEGY: Recur on <value>
```

or

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

or

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

That’s pretty vague– we’ll see more as we do more examples.

Another General-Recursion Pattern

```
;; solve : Problem -> Solution
;; STRATEGY: Recur on simpler-instance

(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (local
       ((define solution1
         (solve (simpler-instance the-problem))))
       (adapt-solution solution1))]))

simpler-instance : Problem -> Problem
adapt-solution : Solution -> Solution
```

Here's a version with two trivial cases and one difficult case, where the difficult case involves only one subproblem.
Most of our functions involving lists match this pattern.

Yet Another General-Recursion Pattern

```
;; solve : Problem -> Solution
;; STRATEGY: Recur on (generate-subproblems the-problem), then use adapt-
   solutions

(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (local
       ((define new-problems
          (generate-subproblems the-problem)))
       (adapt-solutions
        (map solve new-problems)))]))

generate-subproblem : Problem -> ListOfProblem
adapt-solutions : ListOfSolution -> Solution
```

Here's a version where the difficult case requires solving a whole list of subproblems. A tree where a node has a list of sons may lead to use of this pattern.

..or you could do it without the local defines

```
;; solve : Problem -> Solution

(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (adapt-solution
       (solve
         (simpler-instance the-problem)))]))

simpler-instance : Problem -> Problem
adapt-solution : Solution -> Solution
```

Here's the single-subproblem pattern we saw a couple of slides ago, but done without the local **define**s

Yet Another General-Recursion Pattern

```
;; solve : Problem -> Solution

(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (adapt-solutions
       (map solve
         (generate-subproblems the-problem)))]))

generate-subproblem : Problem -> ListOfProblem
adapt-solutions : ListOfSolution -> Solution
```

Here's the list-of-subproblems pattern done without using local **define**.

What pattern did we use for decode?

```
;; decode followed the first pattern we wrote:

(define (solution the-problem)
  (cond
    [(trivial1? the-problem) (trivial-solution1 the-problem)]
    [(trivial2? the-problem) (trivial-solution2 the-problem)]
    [(difficult? the-problem)
     (local
       ((define solution1
          (solve (simpler-instance1 the-problem)))
        (define solution2
          (solve (simpler-instance2 the-problem))))
       (combine-solutions solution1 solution2))]))
```

Writing this down for **decode**

```
;; decode : SexpOfAtom -> MaybeDiffExp

;; STRATEGY: if the top level of sexp could be the top level of
;; a diffexp, recur on 2nd and 3rd elements.

(define (decode sexp)
  (cond
    [(number? sexp) sexp]
    [(could-be-diffexp? sexp)
     (local
       ((define operand1 (decode (second sexp)))
        (define operand2 (decode (third sexp))))
       (if (and (succeeded? operand1)
                (succeeded? operand2))
           (make-diffexp operand1 operand2)
           false))]
    [else false]))
```

The strategy is a tweet-sized description of how the function works. We'll see more about this later.

Another 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.

merge

```
;; merge : SortedList SortedList -> SortedList
;; merges 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**).

merge-sort

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

Something new happened here

- Merge-sort did something very different: it recurs on two things, neither of which is **(rest lon)** .
- We recurred on
  - `(even-elements lon)`
  - `(odd-elements  lon)`
- Neither of these is a sublist of **lon**
  - So this is definitely general recursion, not structural recursion.

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

  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 an insertion sort, which takes **O(n^2)**.

Lesson Summary

- We've seen three examples of functions that do not fit the structural recursion pattern.
- We introduced "general recursion", a new class of templates that give the writer more flexibility in writing functions that divide and conquer.
- We wrote a recipe for writing general-recursion templates.

Next Steps

- Study the files 08-1-decode.rkt and 08-2-merge-sort.rkt in the Examples folder.
- Do Guided Practice 8.1
- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson