Solving Your Problem by Generalization

CS 5010 Program Design Paradigms

Lesson 7.1

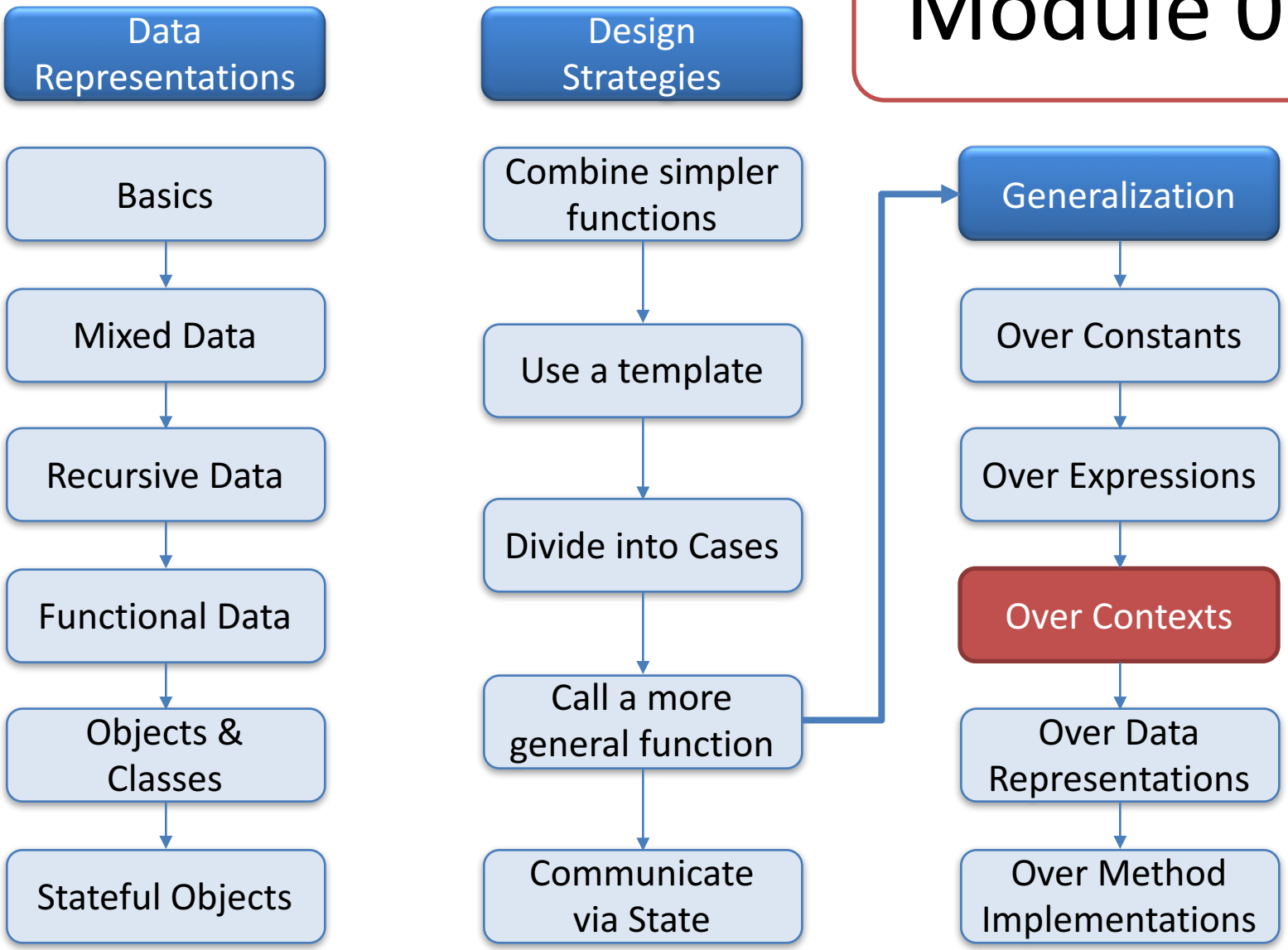
Module Introduction

- Some problems are not easily solved by simply using a template.
- We show how to solve many such problems by introducing new variables called *context variables*.
- We introduce *invariants* as a way of recording the assumptions that a function makes about its context.

Module Outline

- At the end of this module, you should be able to
  - use generalization within a problem to solve the problem
  - use context arguments to generalize over problem contexts
  - write invariants to document the meaning of a context argument
  - explain how invariants divide responsibility between a function and its callers

Module 07


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
Communicate via State
Generalization
Over Constants
Over Expressions
Over Contexts
Over Data Representations
Over Method Implementations

Lesson Introduction

- In Module 5, we learned about generalizing functions in order to avoid code duplication and establish single points of control.
- In this lesson, we'll extend those techniques to situations where the problem itself demands to be generalized before you can solve it.
- Let's look at an example.

Example 1: number-list

```
number-list : ListOfX -> NumberedListOfX
RETURNS: a list like the original, but with the
   elements numbered consecutively, starting
   from 1


(number-list (list 22 44 33))
  = (list (list 1 22) (list 2 44) (list 3 33))
(number-list    (list 44 33))
  = (list (list 1 44) (list 2 33))
```

Here's an example of a problem that's hard to solve using our usual template for lists.

Example 1: number-list

```
A NumberedX        is a (list Int X)
A NumberedListOfX  is a ListOfNumberedX

Example:
(list 14 "abc") is a NumberedString
(list 36 "u")   is a NumberedString
(list
  (list 14 "abc")
  (list 36 "u")
  (list 14 "abc"))   is a NumberedListofString
```

Here are the data definitions for this problem

Let's try using the template for ListOfX

```
(define (number-list lst)
  (cond
    [(empty? lst) empty]
    [else (cons
            (list 1 (first lst))
            (number-list (rest lst)))]))
```

Well, that's clearly wrong! What could work?

We need a help function, to number the rest of the list starting from 2

Try #2

```
(define (number-list lst)
  (cond
    [(empty? lst) empty]
    [else (cons
            (list 1 (first lst))
            (number-list-starting-from-2
              (rest lst)))]))
```

Well, this looks promising. All we have to do now is write **number-list-starting-from-2**

number-list-starting-from-2

```
(define (number-list-starting-from-2 lst)
  (cond
    [(empty? lst) empty]
    [else (cons
            (list 2 (first lst))
            (number-list-starting-from-3
              (rest lst)))]))
```

Oh, dear. Now we have to write **number-list-starting-from-3**

number-list-starting-from-3

```
(define (number-list-starting-from-3 lst)
  (cond
    [(empty? lst) empty]
    [else (cons
            (list 2 (first lst))
            (number-list-starting-from-4
              (rest lst)))]))
```

You should be able to guess where this is going...

Let's generalize!

Add an extra parameter for the starting point of the numbering.

```
;; number-list-from : ListOfX NonNegInt-> NumberedListOfX
;; RETURNS: a list with same elements as lst, but
;;  numbered starting at n.
;; EXAMPLE: (number-list-from (list 88 77) 2)
;;          = (list (list 2 88) (list 3 77))
```

Now the problem is easy

```
;; STRATEGY: Use template for ListOfX
;;      on lst
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
      (cons
        (list n (first lst))
        (number-list-from
          (rest lst)
          (+ n 1)))]))
```

And we can recover the original

```
;; STRATEGY:
;; Call a more general function

(define (number-list lst)
  (number-list-from lst 1))
```

Let’s look again at number-elements

- Let's look at **number-elements** again, in a different way that may give us some more insight.

Let's watch this work

```
(number-list (list 11 22 33))
= (number-list-from (list 11 22 33) 1)
= (cons (list 1 11)
    (number-list-from (list 22 33) 2)
= (cons (list 1 11)
    (cons (list 2 22)
      (number-list-from (list 33) 3)))
= (cons (list 1 11)
    (cons (list 2 22)
      (cons (list 3 33)
        (number-list-from empty 4)))
= (cons (list 1 11)
    (cons (list 2 22)
      (cons (list 3 33)
        empty)))
```

Here's an example of **number-list** in action. In each call of **number-list-from**, I've marked the arguments in red. What do you notice about the first argument of each call? What do you notice about the second argument of each call? What is the relationship between the arguments of each call and the original list, **(list 11 22 33)** ?

What's going on here?

- **`(number-list-from lst n)`** is called on the **`n`**-th sublist of the original.
- So **`n`** is the number of elements in the original that are above **`lst`**
- This is deep knowledge about this function, which we need to capture and document if we are going to explain this code to anybody

We document this as an **invariant**

```
;; number-list-from
;;    : ListOfX NonNegInt-> NumberedListOfX
;; GIVEN: a sublist slst and an integer n
;; WHERE: slst is the n-th sublist
;;  of some list lst0
;; RETURNS: <to be filled in>
```

We don't know what that list **lst0** was; all we know is that we are looking at its **n**-th sublist. We document this knowledge by writing it in a WHERE clause.

The WHERE clause is called an "invariant". It is the responsibility of each caller of this function to make sure that the WHERE clause is satisfied.

The function itself can assume that the WHERE clause is true, just as it assumes that its arguments satisfy its contract.

Now let’s write the rest of the purpose statement

- The function has lost track of the original list; it only knows its position in the original.
- Need to document the connection in the purpose statement.
- Here's the new purpose statement:

New Purpose Statement

```
;; number-list-from
;;   : ListOfX NonNegInt -> NumberedListOfX
;; GIVEN: a sublist slst and an integer n
;; WHERE: slst is the n-th sublist of some list lst0
;; RETURNS: a copy of slst numbered according to its
;;  position in lst0.
;; strategy: Use template for ListOfX on slst
```

First, we document that we are looking at a sublist of some list

We don't know what that list was; all we know is that we are looking at its **n**-th sublist. We document this knowledge by writing it in a WHERE clause.

This is called the *accumulator invariant*

The extra argument n keeps track of the context: where we are in **lst0**

Structural Arguments and Context Arguments

- In this example, **slst** is a *structural argument*: it is the argument that we are doing structural decomposition on.
- **n** is a *context argument*: it tells us something about the context in which we are working. It generally changes at each recursive call, because the recursive call is solving the problem in a new or bigger context.
- The **WHERE** clause tells us how to *interpret* the context argument as a context.

Is the invariant satisfied at the recursive call?

```
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
      (cons
        (list n (first lst))
        (number-list-from (rest lst) (+ n 1)))]))
```

FACT:
if
lst is the **n**th sublist of the original,
then
(rest lst) is its **(n+1)**-st sublist.

So, if the current call satisfies the invariant, then the recursive call also satisfies the invariant.

Context Arguments and Accumulators

- The book calls context arguments "accumulators".
- For each function you write, you need to be clear on what the structural argument is.
  - You've been doing that already in the strategy
- Unlike the book, we are not going to make a big deal over what is or is not a context argument/accumulator.
- We are also not going to have "+ accumulator" as a strategy or have templates for "structural decomposition + accumulator".

One less thing for you to stress over!

This isn't completely new:

Here are some examples of **WHERE** clauses that we've seen (or might have seen) before:

```
-- A Ring is a (make-ring Real Real)
   WHERE inner < outer

-- An TelephoneBook is a ListOfEntries
   WHERE the entries are sorted by name
```

More examples of **WHERE** clauses

```
unpaused-world-after-tick
  : World -> World
GIVEN: a World
WHERE: the world is not paused
RETURNS: the state of the world after the next tick

ball-normal-motion-after-tick
  : Ball -> Ball
GIVEN: a Ball
WHERE: we know the ball will not hit the wall on the next
       tick
RETURNS: the state of the ball after the next tick.
```

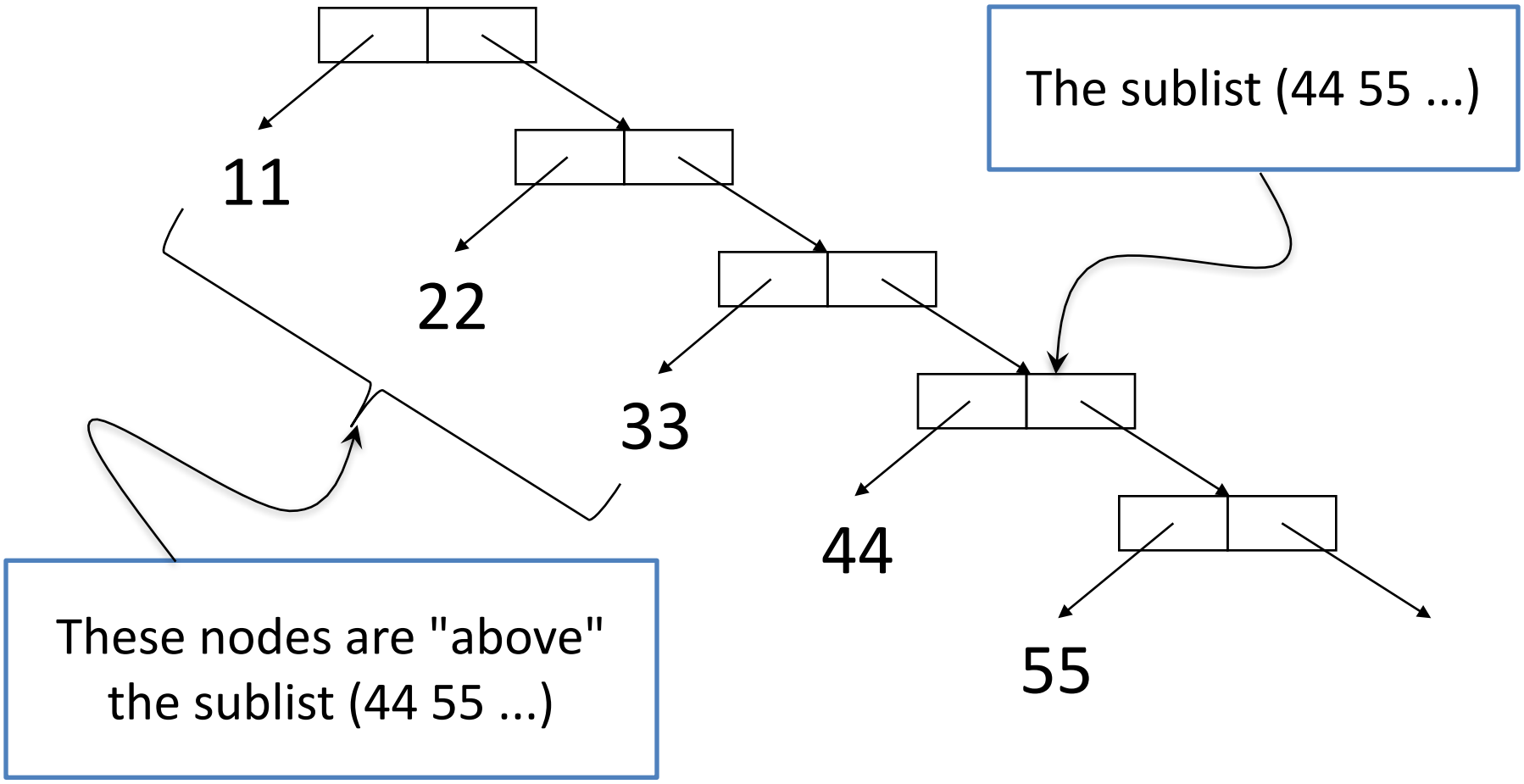
In each case, it is the responsibility of the caller to make sure the invariant is satisfied before the function is called.

And conversely, the function gets to assume that the invariant is satisfied.

Recipe for context arguments

Recipe for context arguments
Is information being lost when you do a structural recursion? If so, what?
Formulate a generalized version of the problem that works on a substructure of your original. Add a context argument that represents the information "above" the substructure. Document the purpose of the context argument as an invariant in your purpose statement.
Design and test the generalized function.
Define your original function in terms of the generalized one by supplying an initial value for the context argument.

Wait: what do we mean by "above"?



Summary

- Sometimes you need more information than the usual template gives you
- So generalize the problem to include the extra information as a parameter
- Design the generalized function
- Then define your original function in terms of the generalized one.

Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Go on to the next lesson