Invariants and Context Variables

CS 5010 Program Design Paradigms
"Bootcamp"

Lesson 7.2

Key Points for Lesson 7.2

- Sometimes our function needs more information than simply its place in a decision tree.
- We often capture this information in a *context variable.*
- A context variable is an abstraction of the information that we "pass over" when we recur on a structure.
- The invariant serves as a kind of interpretation for the data in the context variable.

Let’s do an example.

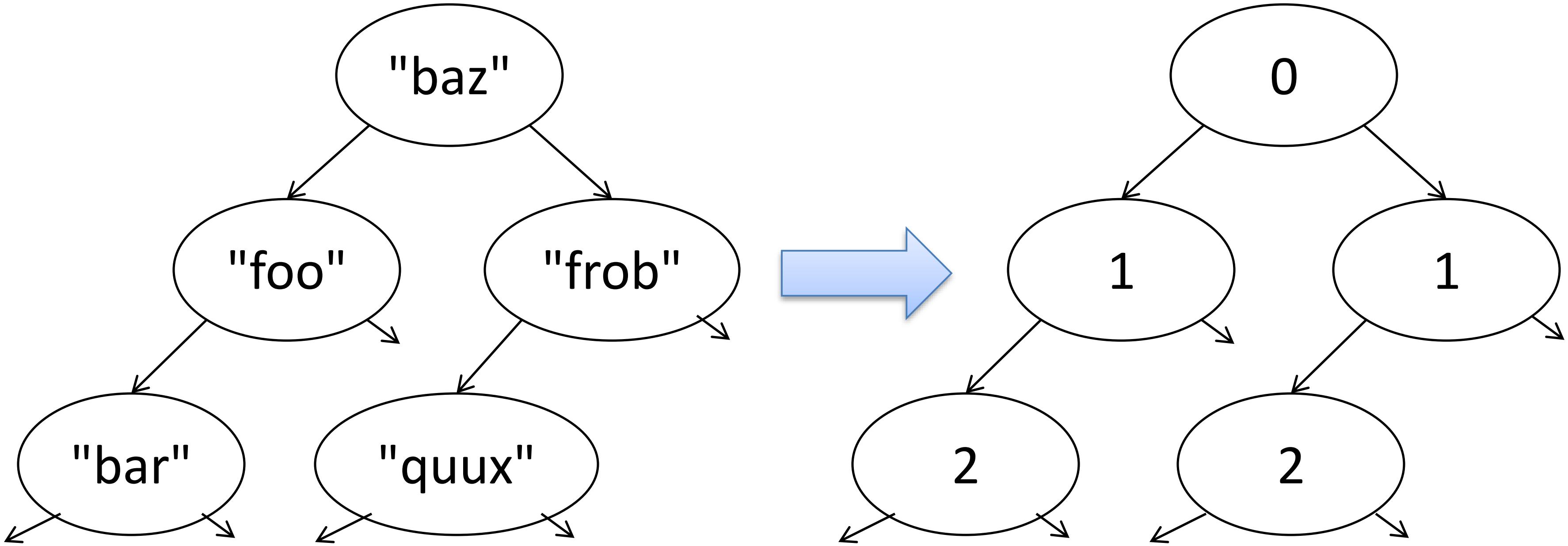
```
(define-struct bintree-node (left data right))

;; A XBintree is either
;; -- empty
;; -- (make-bintree-node XBintree X XBintree)
```

A **XBintree** is a binary tree with a value of type **X** in each of its nodes. For example, you might have **SardineBintree**. This is, of course, a different notion of binary tree than we saw in Lesson 5.1.

Example: mark-depth (2)

```
;; mark-depth : XBintree -> NumberBintree
;; RETURNS: a bintree like the original, but
;; with each node labeled by its depth
```

Example



Here's an example of the argument and result of **mark-depth**. The argument is a **StringBintree** and the result is a **NumberBintree**, just like the contract says.

Observer Template for **XBintree**

```
(define (bintree-fn tree)
  (cond
    [(empty? tree) ...]
    [else (...
           (bintree-fn
             (bintree-node-left tree))
           (bintree-data tree)
           (bintree-fn
             (bintree-node-right tree)))]))
```

If we follow the recipe for writing a template, this is what we get for **XBintree**.

Filling in the template

```
(define (mark-depth tree)
  (cond
    [(empty? tree) ...]
    [else (make-bintree-node
            (mark-depth
               (bintree-node-left tree))
            ...
            (mark-depth
               (bintree-node-right tree)))]))
```

We want to put the depth here.
But how do we know the depth?

We need another argument!

- We'll add another argument to represent the depth that we are in the tree.
- Then we can write:

Function Definition

```
(define (mark-depth-2 tree d)
  (cond
    [(empty? tree) empty]
    [else (make-bintree-node
            (mark-depth-2 (bintree-node-left tree)
                          (+ d 1))
            d
            (mark-depth-2 (bintree-node-right tree)
                          (+ d 1)))]))
```

Different arguments, different contract. We'll change the name so we won't get confused.

Function Definition, with Explanation

```
(define (mark-depth-2 tree d)
  (cond
    [(empty? tree) empty]
    [else (make-bintree-node
            (mark-depth-2 (bintree-node-left tree)
                          (+ d 1))
            d
            (mark-depth-2 (bintree-node-right tree)
                          (+ d 1)))]))
```

We start with a tree at depth **d**.

depth **d+1**, so we recur on the subtrees with **(+ 1 d)**

We are at depth **d**, so we put a **d** in this node.

How do we document this?

- We change the name of the function to **mark-subtree**. To emphasize the fact that we are dealing with a subtree somewhere inside a tree.
- We'll reserve the original name for the original function that works on the whole tree.
- We'll also change the name of the argument from **tree** to **st** (abbreviation for "subtree") to keep us focused on the fact that we're dealing with
- Then we'll add an invariant to say that **d** is the depth of our node in the whole tree.

Function Definition, with Invariant

```
;; mark-subtree : XBintree NonNegInt-> NumberBintree
;; GIVEN: a subtree st of some tree t, and a non-neg int d
;; WHERE: the subtree occurs at depth d in the tree t
;; RETURNS: a subtree the same shape as st, but in which
;; each node is marked with its distance from the top of the tree t
;; STRATEGY: Use template for XBintree on stree

(define (mark-subtree st d)
  (cond
    [(empty? st) empty]
    [else (make-bintree
           (mark-subtree (bintree-left st)
                         (+ d 1))
           d
           (mark-subtree (bintree-right st)
                         (+ d 1)))]))
```

Function Definition, with Invariant

```
;; mark-subtree : XBintree NonNegInt-> NumberBintree
;; GIVEN: a subtree st of some tree t, and a non-neg int d
;; WHERE: the subtree occurs at depth d in the tree t
;; RETURNS: a subtree the same shape as st, but in which
;; each node is marked with its distance from the top of the tree t
;; STRATEGY: Use template for XBintree on stree

(define (mark-subtree st d)
  (cond
    [(empty? st) empty]
    [else (make-bintree
           (mark-subtree (bintree-left st)
                         (+ d 1))
           d
           (mark-subtree (bintree-right st)
                         (+ d 1)))]))
```

The invariant tells us where we are in the whole tree

If **st** is at depth **d**, then its sons are depth **d+1**. So the WHERE clause is satisfied at each recursive call.

And we need to reconstruct the original function, as usual

```
;; mark-tree : XBintree -> NumberBintree
;; GIVEN: a binary tree t
;; RETURNS: a tree the same shape as t, but in which
;; each node is marked with its distance from the top of
;; the tree
;; STRATEGY: call a more general function
(define (mark-tree t)
  (mark-subtree t 0))
```

The whole tree is a subtree of itself, with its top node is at depth 0, so the invariant of mark-subtree is satisfied the first time it is called.

Structural Arguments and Context Arguments

- In this example, we call **st** a *structural argument*: we are recurring on the structure of this argument.
- We call **d** 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.

Let’s do another example

- Finding the sum of a list of numbers
- We’ve done this by a simple recursion, but let’s do it a different way.
- In the simple recursion, we did the addition from right to left.
- In the new solution, we’ll do it left to right.

The old solution: nl-sum (Lesson 4.1)

```
;; nl-sum : NumberList -> Number
(define (nl-sum lst)
  (cond
    [(empty? lst) 0]
    [else (+   (first lst)
               (nl-sum (rest lst)))]))
```

nl-sum sums from right to left

```
(nl-sum (cons 11 (cons 22 (cons 33 empty))))
= (+ 11  (nl-sum (cons 22 (cons 33 empty))))
= (+ 11  (+ 22    (nl-sum (cons 33 empty))))
= (+ 11  (+ 22    (+ 33    (nl-sum empty))))
= (+ 11  (+ 22    (+ 33    0)))
= (+ 11  (+ 22    33))
= (+ 11  55)
= 66
```

A different solution

```
(define (sublist-sum so-far unsummed)
  (cond
    [(empty? unsummed) so-far]
    [else (sublist-sum (+ so-far (first unsummed))
                       (rest unsummed))]))

(define (list-sum l)
  (sublist-sum 0 l))
```

Think about this definition for a minute. Can you figure out how it works?

Let’s watch this one work

```
  (list-sum      (cons 11 (cons 22 (cons 33 empty))))
= (sublist-sum 0 (cons 11 (cons 22 (cons 33 empty))))
= (sublist-sum 11         (cons 22 (cons 33 empty)))
= (sublist-sum 33                  (cons 33 empty))
= (sublist-sum 66                           empty)
= 66
```

This function works from left to right

- The first argument to sublist-sum is the sum of all the elements we've looked at "so far".
- This is a context argument: at each recursive call, represents the context in which sublist-sum is called.
- We say that it *abstracts* the context: it keeps only as much information about the context as the function needs.
- Let's write down a proper invariant to document this:

Invariant for **sublist-sum**

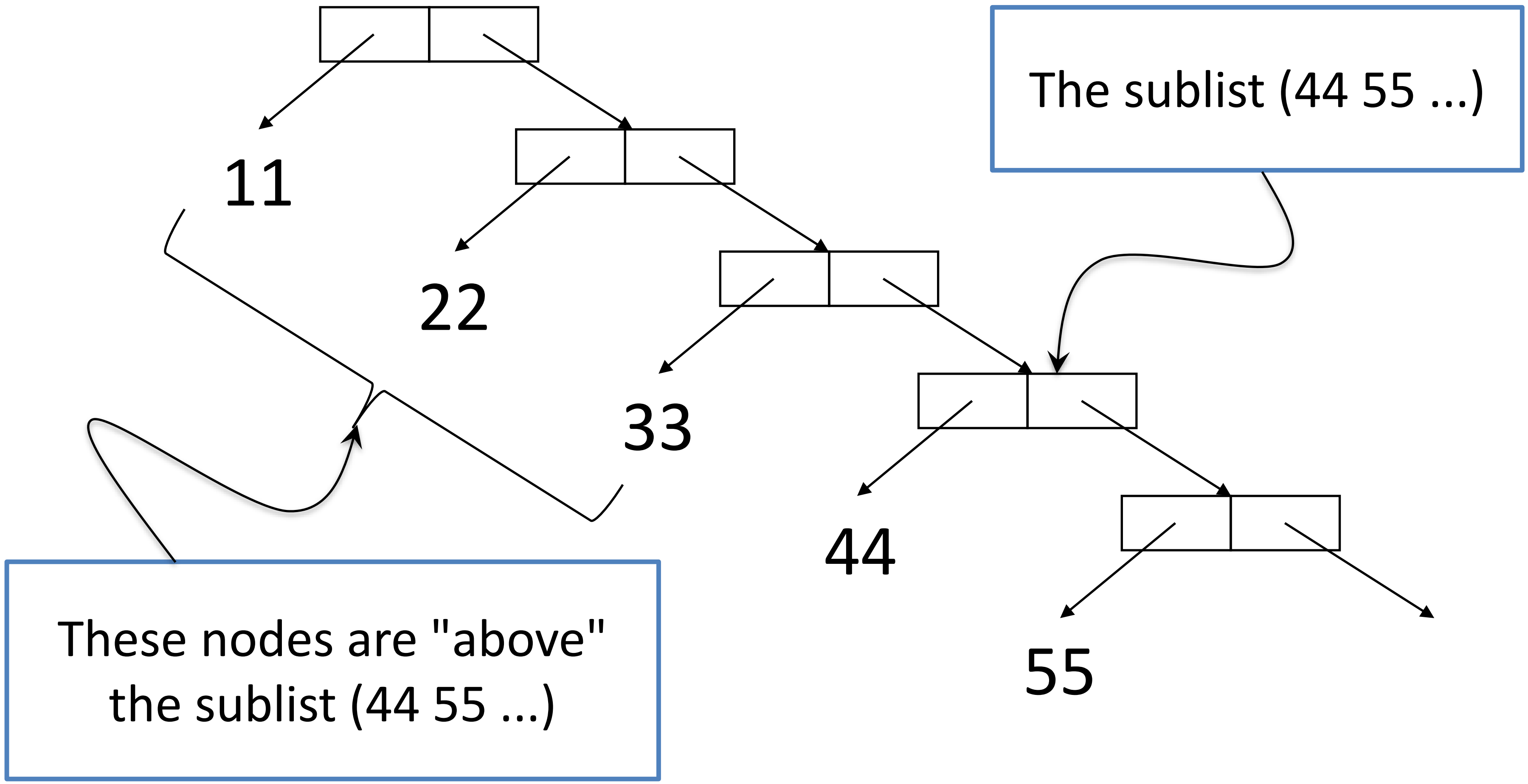
```
;; sublist-sum : Number NumberList -> Number
;; GIVEN: a number 'so-far' and a list of numbers 'unsummed'
;; WHERE: 'unsummed' is a sublist of some list 'whole-list'
;; AND:   so-far is the sum of all the elements to the left of
;;        unsummed in whole-list
;; RETURNS: the sum of all the elements in whole-list.
;; EXAMPLE:
;; (sublist-sum 5 (list 2 3 4)) = 14  [whole-list was (3 2 2 3 4)]
;; (sublist-sum 5 (list 2 3 4)) = 14  [whole-list was (3 1 1 2 3 4)]
;; note that a given set of arguments might correspond to different
;; values of 'whole-list'.  All we care about whole-list is that the
;; sum of its elements before the (list 2 3 4) is exactly 5
;; STRATEGY:
;; observer pattern for NumberList on 'unsummed'
(define (sublist-sum so-far unsummed)
  (cond
    [(empty? unsummed) so-far]
    [else (sublist-sum (+ so-far (first unsummed))
                       (rest unsummed))]))
```

so-far is the sum of the elements on **whole-list** that we've looked at so far; **unsummed** is the portion of the list that we haven't summed yet.

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"?



Review: Key Points for Lesson 7.2

- Sometimes our function needs more information than simply its place in a decision tree.
- We often capture this information in a *context variable.*
- A context variable is an abstraction of the information that we “pass over” when we recur on a structure.
- The invariant serves as a kind of interpretation for the data in the context variable.

Next Steps

- Study 07-2-1-mark-depth.rkt and 07-2-2-sum-list.rt
- If you have questions about this lesson, ask them on Piazza.
- Do Guided Practices 7.1 and 7.2
  - Be sure to do GP7.2, since it introduces material not covered in the slides!
- Go on to the next lesson