# More examples of invariants

CS 5010 Program Design Paradigms

Lesson 7.2

# Lesson Introduction

- In Lesson 7.1, we introduced context arguments and invariants to solve problems involving lists
- In this lesson, we'll use these ideas to solve problems involving trees and mutually-recursive data definitions.

# Example 2: mark-depth

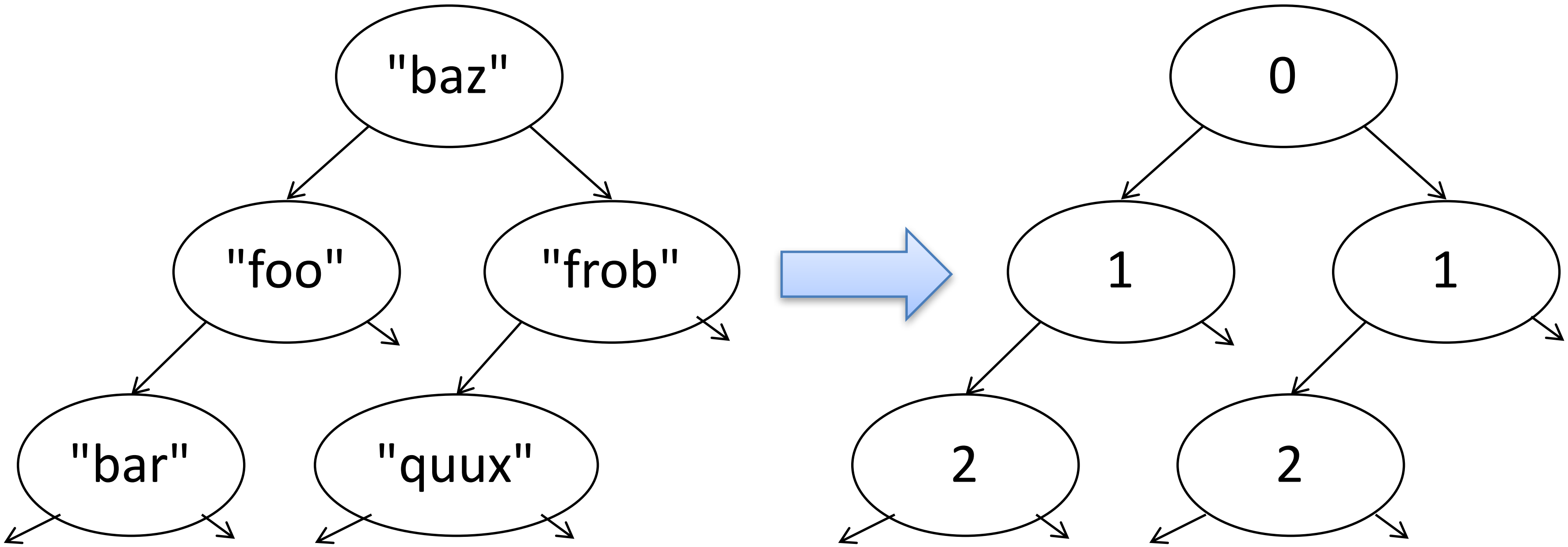
```
(define-struct bintree (left data right))

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

A **BintreeOfX** is a binary tree with a value of type **X** in each of its nodes. For example, you might have **BintreeOfSardines**. This is, of course, a different notion of binary tree than we saw last week.

# Example 2: mark-depth (2)

```
;; mark-depth : BintreeOfX -> BintreeOfNumber
;; 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 **BintreeOfString** and the result is a **BintreeOfNumber**, just like the contract says.

# Template for BinTreeOfX

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

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

# Filling in the template

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

But how do we know the depth?

# So let's add a context argument

```
;; mark-subtree : BinTreeOfX NonNegInt-> BinTreeOfNumber
;; GIVEN: a subtree stree of some tree, and a non-neg i
;; WHERE: the subtree occurs at depth n in the tree
;; RETURNS: a tree the same shape as stree, but in which
;; each node is marked with its distance from the top of the tree
;; STRATEGY: Use template for BinTreeOfX on stree
(define (mark-subtree stree n)
  (cond
    [(empty? stree) empty]
    [else (make-bintree
            (mark-subtree (bintree-left stree) (+ n 1))
            n
            (mark-subtree (bintree-right stree) (+ n 1)))]))
```

The invariant tells us where we are in the whole tree

The RETURNS clause tells us how our answer fits into the original problem.

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

# And we need to reconstruct the original function, as usual

```
;; mark-tree : BinTreeOfX -> BinTreeOfNumber
;; GIVEN: a binary tree
;; RETURNS: a tree the same shape as tree, 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 tree)
  (mark-subtree tree 0))
```

The whole tree is a subtree, and its top node is at depth 0, so the invariant of mark-subtree is satisfied.

# What about mutually recursive data definitions?

- You’ll have two mutually recursive functions to handle the sub-Sos and sub-Loss– nothing else changes.
- Let's write this out by writing down the Sos and Loss templates and adding a context argument.

# Template for SoS and LoSS, with context argument (part 1)

```
;; GIVEN: a SoS sos that is a subpart of some
;;  larger SoS sos0, and <describe ctxt>
;; WHERE: <describe how ctxt represents the
;;  portion of sos0 that lies above sos>
;; RETURNS: <something in terms of sos and sos0>
;; STRATEGY: Use the template for SoS on subsos

(define (sub-sos-fn subsos ctxt)
  (cond
    [(string? subsos) ...]
    [else (... (sub-loss-fn subsos (... ctxt)))]))
```

The invariant documents the meaning of ctxt

This still fits the SoS template

When we have a recursive call, we use a new value of the context argument, so that **sub-loss-fn**'s invariant will be true.

# Template for SoS and LoSS, with context argument (part 2)

```
;; GIVEN a LoSS loss that is a subpart of some
;;  larger SoS sos0, and a <describe ctxt>
;; WHERE: <describe how ctxt represents the
;;  portion of sos0 that lies above loss>
;; RETURNS: <something in terms of loss and sos0>
;; STRATEGY: Use template for Loss on sublos
(define (sub-loss-fn subloss ctxt)
  (cond
    [(empty? subloss) ...]
    [else (...
            (sub-sos-fn (first subloss) (... ctxt))
            (sub-loss-fn (rest subloss) (... ctxt))])))
```

The invariant again documents the meaning of **ctxt**

This still fits the LoSS template

Each recursive call uses a new value for the context argument, so that each called function's invariant will be true.

# Template for SoS and LoSS, with context argument (part 3)

```
;; GIVEN a SoSS sos0
;; RETURNS: <something>
;; Strategy: call a more general function
(define (sos-fn sos0)
  (sub-sos-fn sos ...))
```

Of course we need a function for the whole SoS!

Pass sub-sos-fn a value for its context argument that describes the empty context– that is, one that will make its invariant true.

# Summary

- You should now be able to:
  - explain the difference between structural arguments and context arguments
  - understand how context arguments represent contexts
  - document this representation as an invariant in the purpose statement
  - use these ideas to solve problems for lists, trees, and mutually-recursive data definitions.

# Next Steps

- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 7.1
- Go on to the next lesson