Accumulators

CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 5.1

Goals of this lesson

- Add a new strategy: "structural decomposition with accumulator"
- Accumulators are extra arguments that accumulate the context in which a function is called.
- We use an "accumulator invariant" to describe the information represented by the accumulator argument.
- Introduce the template for using an accumulator

Example 1: number-list

```
A NumberedListOf<X> is a ListOf<(list Num X)>

number-list : ListOf<X> -> NumberedListOf<X>
produce a list like the original, but with the
  elements numbered.

(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))
```

Let's try structural decomposition

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

What must number-list-helper do? Let's look at our example.

What must number-list-helper do?

```
(number-list (list 22 44 33))
= (number-list-helper
    22
    (number-list (list 44 33)))
= (number-list-helper
    22
    (list (list 1 44) (list 2 33)))
=     magic!

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

Bleahh!

- What we really want is for the recursive call to return

```
(list (2 44) (3 33))
```

- Then we could just write

```
(cons (list 1 (first lst))
      ...result of recursive call...)
```

- We want the recursive call to return (rest lst) numbered starting at 2.

So our code is something like

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

So let's generalize!

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

```
;; number-list-from : ListOf<X> Number ->
                        NumberedListOf<X>
;; Produces 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))
```

So let's generalize!

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

Whoa! Something new here: we've changed the extra parameter.

A parameter that changes on a recursive call, like **n**, is called an *accumulator.*

So let's generalize!

```
;; STRATEGY: structural decomp w/ accumulator
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
      (cons
        (list n (first lst))
        (number-list-from
          (rest lst)
          (+ n 1)))]))
```

Whoa! Something new here: we've changed the extra parameter.

A parameter that changes on a recursive call, like **n**, is called an *accumulator.*

But remember to recover the original function!

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

Could we do this without accumulators?

- Let's recall our original code

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

What must number-list-helper do? Let's look at our example again.

What must number-list-helper do?

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


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

What must number-list-helper do?

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

I see a map here

And a cons here

So now we can write the code

```
;; number-list-helper :
;;    X NumberedListOf<X> -> NumberedListOf<X>
;; Given x1 and ((1 x2) (2 x3) ...), produce the list
;; ((1 x1) (2 x2) (3 x3) ...)
;; strategy: HOFC + SD on (list Number X)
(define (number-list-helper first-val numbered-list)
  (cons
    (list 1 first-val)
    (map
      ;; (list Number X) -> (list Number X)
      ;; Returns a list like the original, but with the first element
      ;; incremented
      (lambda (elt)
        (list
          (+ 1 (first elt))
          (second elt)))
      numbered-list)))
```

Let's test it...

- and stress-test it:

```
;; tester : (Number -> X) Number -> Number
;; runs fn on the list [0..n-1], prints the time it took,
;; and returns n
(define (tester fn n)
  (local
    ((define ignored
       (time
        (fn (build-list n (lambda (n) n))))))
    n))
```

- 05-1-number-list-with-stress-tests.rkt

Now let's test the accumulator version

- and stress-test it:

```
;; tester : (Number -> X) Number -> Number
;; runs fn on the list [0..n-1], prints the time it took,
;; and returns n
(define (tester fn n)
  (local
    ((define ignored
       (time
        (fn (build-list n (lambda (n) n))))))
    n))
```

- 05-1-number-list-with-stress-tests.rkt

What happened here?

- If **`lst`** has length *N*, then without an accumulator:
  - **`(number-list-helper n lst)`** takes time proportional to *N* (we say it is *O(N)*)
  - **`(number-list lst)`** calls **`number-list-helper`** *O(N)* times.
  - So the whole thing takes *O(N^2)*.
- The version with accumulator runs in time *O(N)*.
  - much, much faster!

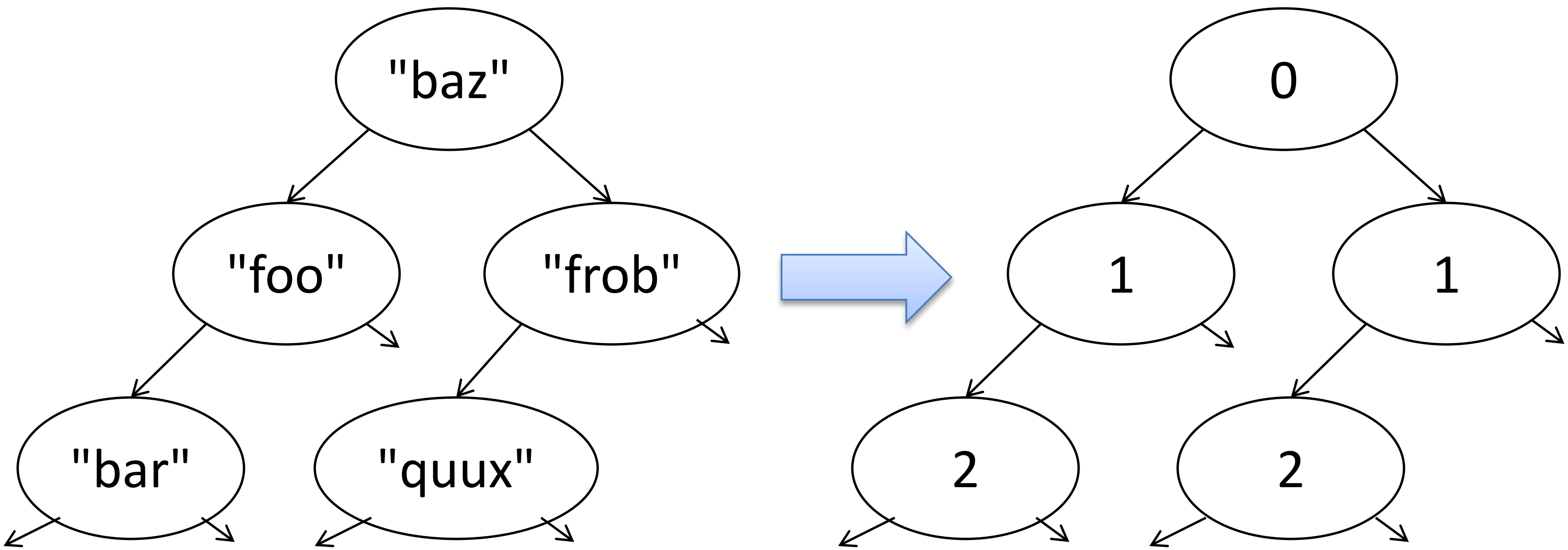
Example 2: mark-depth

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

;; A BinTree<X> is either
;; -- empty
;; -- (make-bintree BinTree<X> X BinTree<X>)

;; mark-depth : BinTree<X> -> Bintree<Number>
;; return a bintree like the original, but with
  each node labelled by its depth
```

Example



Template for BinTree<X>

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

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 again, let's generalize by adding an extra argument

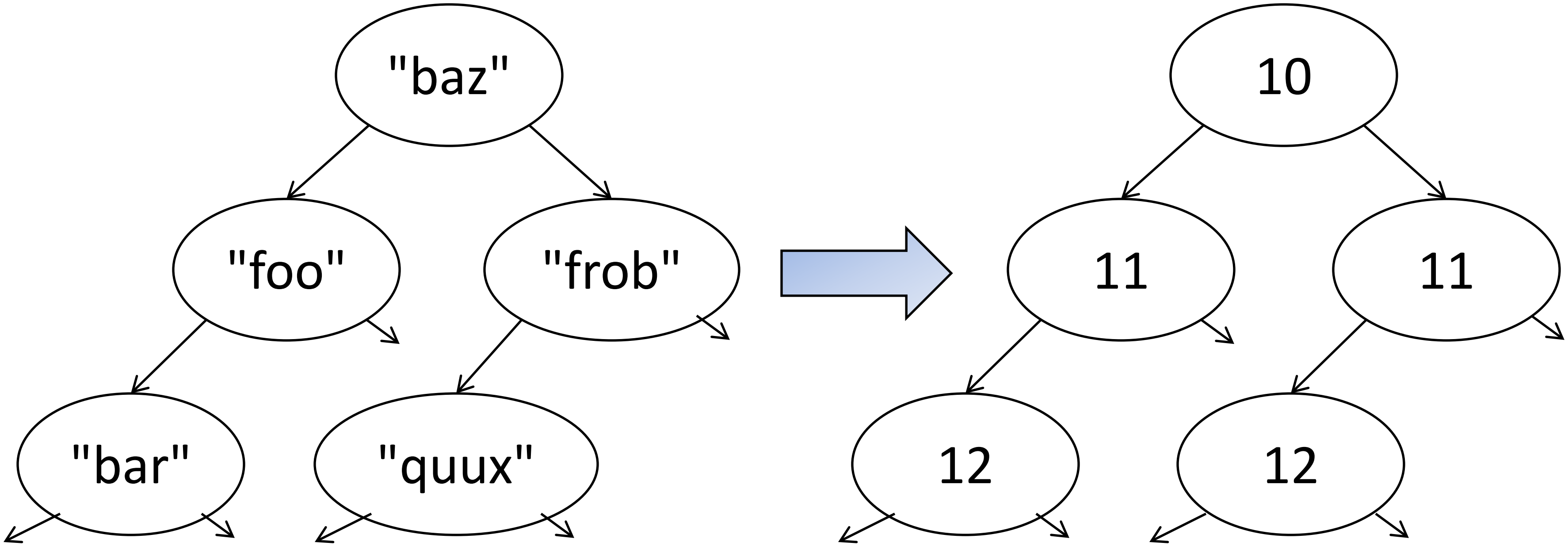
```
mark-depth-from : Bintree<X> Number ->
                                 Bintree<Number>
```

Return a bintree like the given one, except that each node is replaced by its depth ***starting from n***

examples: see below

strategy: structural decomposition on
Bintree<X> [tree] + accumulator [n]

Example (n = 10)



Livecoding: mark-depth.rkt

Recipe for accumulators (version 1)

Recipe for accumulators
Is information being lost when you do a structural recursion? If so, what?
Formulate a generalized version of the problem that includes the extra information as an accumulator. Document the purpose of the extra argument 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 accumulator.

Summary

- Sometimes you need more information than what structural decomposition gives you
- So generalize the problem to include the extra information as a parameter
- This parameter will probably be an accumulator: an extra parameter that changes when you recur
- Design the generalized function
- Then define your original function in terms of the generalized one.

Accumulators Accumulate Context

CS 5010 Program Design Paradigms "Bootcamp"

Lesson 5.2

Goals of this lesson

- Understand how accumulators represent contexts
- Learn how to document this as an *accumulator invariant*.
- Develop a template for structural decomposition + accumulator
- See another example using accumulators

Goals of this lesson

- Understand how accumulators represent contexts
- Learn how to document this as an *accumulator invariant* in the purpose statement.
- Learn how to do this for mutually-recursive data definitions

Let’s look again at number-elements

```
;; STRATEGY: structural decomp w/ accumulator
(define (number-list-from lst n)
  (cond
    [(empty? lst) empty]
    [else
      (cons
        (list n (first lst))
        (number-list-from
          (rest lst)
          (+ n 1)))]))

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

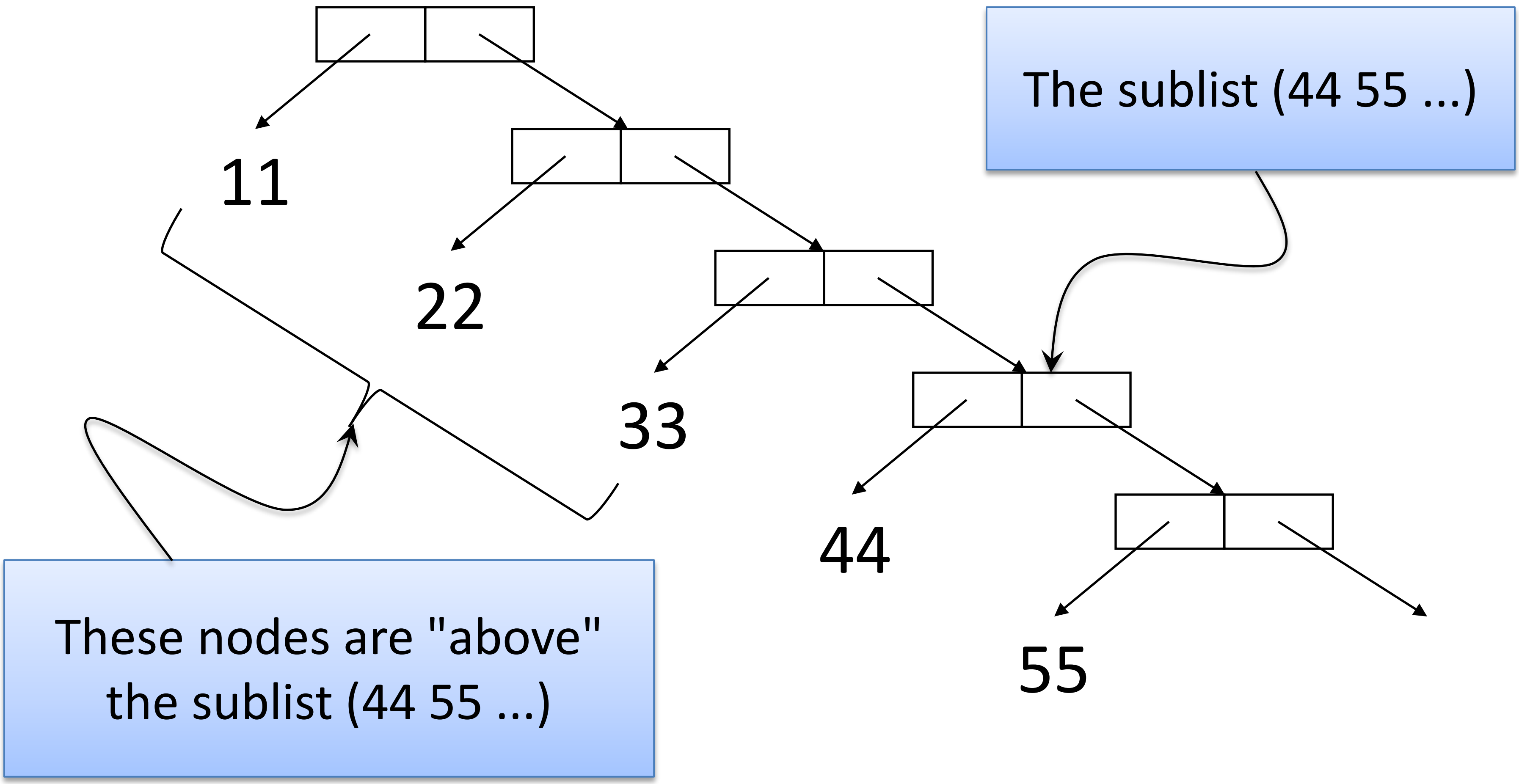
Let’s watch this work

```
(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)))
```

What does n represent?

- **`(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`**

What do we mean by "above"?



How could we document this?

- The helper has lost track of the original list; it only knows its position in the original.
- Need to document the connection in the purpose statement.

Example

Document that we are looking at a sublist of some list

```
;; number-list-from : ListOf<X> Number -> NumberedListOf<X>
;; GIVEN a sublist slst
;; WHERE slst is the n-th sublist of some list lst0
;; PRODUCES a copy of slst numbered according to its
   position in lst0.
;; strategy: struct decomp [slst : ListOf<X>]
;;  + accumulator [n]
(define (number-sublist slst n)
  (cond
    [(empty? slst) empty]
    [else
      (cons
        (list n (first slst))
        (number-sublist (rest slst) (+ n 1)))]))
```

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

This is called the *accumulator invariant*

Bintree<X> example

```
;; mark-subtree : Bintree<X> Nat -> Bintree<Nat>
;; GIVEN: a subtree stree of some tree
;; WHERE: the subtree occurs at depth n in the tree
;; PRODUCES: a tree the same shape as stree, but in which each node is
;; marked with its distance from the top of the tree
;; STRATEGY: struct decomp [stree : Bintree<X>]
;;           + accumulator [n]
(define (mark-subtree stree n)
  (cond
    [(empty? tree) empty]
    [else (make-bintree
            (mark-subtree (bintree-left stree) (+ n 1))
            n
            (mark-subtree (bintree-right stree) (+ n 1)))]))
```

What about mutually recursive data defs?

- You’ll have two mutually recursive fcns to handle the sub-Sos and sub-Loss– nothing else changes.

Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many cond clauses as subclasses that the data definition distinguishes.
How do the subclasses differ from each other?	Use the differences to formulate a condition per clause.
Do any of the clauses deal with structured values?	If so, add appropriate selector expressions to the clause.
Does the data definition use self-references?	Formulate ``natural recursions'' for the template to represent the self-references of the data definition.
Does the data definition refer to another data definition?	Formulate ``natural recursions'' for the template to represent the mutually-recursive references of the data definition.
Do you want an accumulator?	Add argument for accumulator and document it with ***INVARIANT*** ("where" clause)

Recipe for accumulators (version 2)

Recipe for accumulators
Is information being lost when you do a structural recursion? If so, what?
Formulate a generalized version of the problem that that works on a substructure of your original. Add an accumulator that represents the information "above" the substructure. Document the purpose of the accumulator 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 accumulator.

A Tiny Programming Language: Foombles

- The Information:

```
Foomble = Variable | (lambda (Variable) Foomble)
                   | (Foomble Foomble)
Variable = x | y | z | ... | xx | yy | zz | ...
```

Free-Vars

A variable is free if it occurs in a place that is not inside a lambda with the same name.

```
free-vars: Foomble -> SetOf<Variable>
examples (in terms of information, not data):
(free-vars x) => (list x)
(free-vars (lambda (x) x)) => empty
(free-vars (lambda (x) (x y)))
     => (list y)
(free-vars (z (lambda (x) (x y))))
     => (list z y)  {(list y z) would be ok}
(free-vars (x (lambda (x) (x y))))
     => (list x y)  {(list y x) would be ok}
```

Data Design

```
(define-struct var-foomble (name))
(define-struct lambda-foomble (var body))
(define-struct app-foomble (fn arg))

;; A Foomble is one of
;; (make-var-foomble Symbol)
;; (make-lambda-foomble Symbol Foomble)
;; (make-app-foomble Foomble Foomble)

;; interpretation: the cases represent
;; variables, lambdas, and applications,
;; repectively.
```

Template

```
foomble-fn : Foomble -> ?
(define (foomble-fn f)
  (cond
    [(var-foomble? f)
     (... (var-foomble-name f))]
    [(lambda-foomble? f)
     (...
      (lambda-foomble-var f)
      (foomble-fn (lambda-foomble-body f)))]
    [(app-foomble? f)
     (...
      (foomble-fn (app-foomble-fn f))
      (foomble-fn (app-foomble-arg f)))]))
```

Contract & purpose statement

```
;; free-vars : Foomble -> SetOf<Symbol>
;; Produces the set of names that occur free in
  the given Foomble
;; EXAMPLE:
;; (free-vars (z (lambda (x) (x y)))) = {y, z}
;; strategy: structural decomposition
```

We will represent sets as lists without duplication, as in sets.rkt.

Livecoding: foombles.rkt

`Using structural decomposition`

Function Definition

```
;; strategy: structural decomposition
(define (free-vars f)
  (cond
    [(var-foomble? f) (list (var-foomble-name f))]
    [(lambda-foomble? f)
      (set-minus
       (free-vars (lambda-foomble-body f))
       (lambda-foomble-var f))]
    [(app-foomble? f)
     (set-union
      (free-vars (app-foomble-fn f))
      (free-vars (app-foomble-arg f)))]))
```

expensive operation!

What do we lose as we descend into the structure?

- We lose information about which lambda-variables are above us.
- Fixed this up "on the way back up" with set-minus.
- Alternative: use an accumulator to keep track of the lambda-variables above us
  - like the counter in mark-depth
  - taking care of it "on the way down"

Contract and Purpose Statement

```
free-vars-acc : Foomble ListOf<Symbol>
                -> SetOf<Symbol>
GIVEN a sub-foomble sf
WHERE sf is part of some larger foomble f0
AND   los is the list of symbols that occur in
      lambdas above sf in f0,
PRODUCES the set of symbols from sf that are free in f0.

EXAMPLE: [in terms of information, not data]
(free-vars-acc
  (z (lambda (x) (x y)))
  (list z))
= (list y)
```

Livecoding: foombles.rkt

`...using accumulators`

Function Definition

```
;; STRATEGY: Struct Decomp on sf : Foomble + Accumulator [los]
(define (free-vars-acc sf los)
  (cond
    [(var-foomble? sf)
     (if (member (var-foomble-name sf) los)
         empty
         (list (var-foomble-name sf)))]
    [(lambda-foomble? sf)
     (free-vars-acc
      (lambda-foomble-body sf)
      (set-cons
       (lambda-foomble-var sf)
       los))]
    [(app-foomble? sf)
     (set-union
      (free-vars-acc (app-foomble-fn sf) los)
      (free-vars-acc (app-foomble-arg sf) los))]))
```

Is the variable already bound?

Add the lambda-variable to the list of bound variables in the body

Function Definition (part 2)

```
;; free-vars : Foomble -> SetOf<Symbol>
;; Produces the set of names that occur free in
  the given Foomble
;; EXAMPLE:
;; (free-vars (z (lambda (x) (x y))))
;;  = {y, z}

;; Strategy: function composition
(define (free-vars f)
   (free-vars-acc f empty))
```

How to choose?

- Both definitions are clear
- Which performs better?

Performance

Run time, in msec, vs. number of nodes

Size	no accumulator	accumulator
2559	0	0
81,919	328	47
655,358	2528	390
2,621,439	10732	1591

Summary

- Accumulators represent the context between the original argument and the current one.
- Must document what information from the context is being represented in the accumulator.
- We do this with an *accumulator invariant* expressed as a WHERE clause in the purpose statement.

Foldr and Foldl

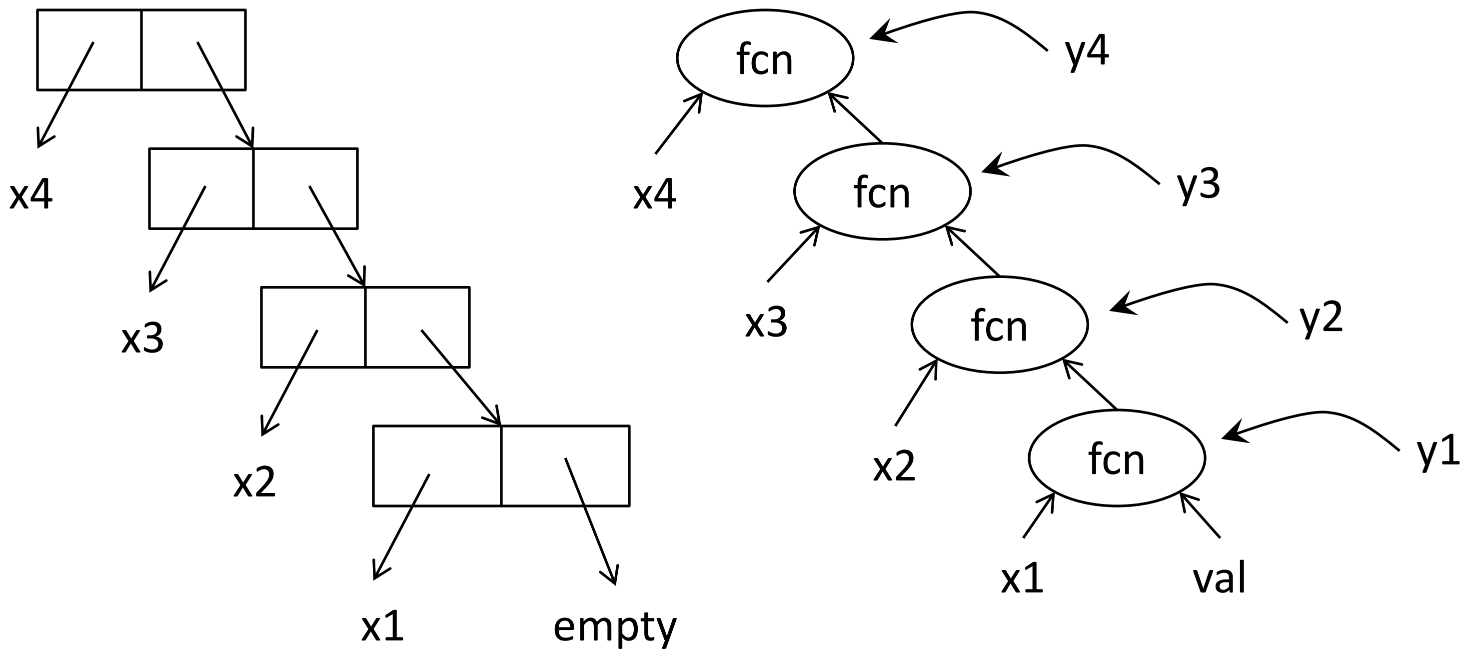
CS 5010 Program Design Paradigms
“Bootcamp”

Lesson 5.3

Goals of this lesson

- Look more closely at foldr
- Introduce foldl: like foldr but "in the other direction"
- Implement using accumulators
- Look at an application

foldr: the general picture




```
fcn : X Y -> Y
val : Y
```

```
foldr : (X Y -> Y) Y ListOf<X> -> Y
```

Another picture of foldr

Page 302 (2nd ed) or 313 (1st ed) says:

```
;; foldr : (X Y -> Y) Y ListOf<X> -> Y
;; (foldr f base (list x_1 ... x_n))
;;   = (f x_1 ... (f x_n base))
```

This may be clearer if we write the combiner in infix:
eg (x - y) instead of (f x y) :

```
(foldr – a (list x1 ... xn)) =
  x1 – (x2 – (... – (xn – a)))
```

What if we wanted to associate the other way?

Instead of

```
x1 – (x2 – (... – (xn – a)))
```

suppose we wanted

```
(((a – x1) – x2) ... – xn)
```

Where would we be in the middle of the computation?

```
(((a - x1) - x2) x3 ... - xn)
```

- At this point, we've processed x1 and x2, and we are looking at the sublist **`(x3 ... xn)`**

So introduce an accumulator

- to keep track of the result of processing the values we've seen so far

Contract and Purpose Statement

```
diff-helper : Number ListOf<Number>
                  -> Number
GIVEN a number sofar and a sublist slon
WHERE slon is a sublist of some LON lon
AND   sofar is the difference of the
      values above slon
PRODUCES the difference for the whole lon.

Example:
(diff-helper 30 (list 10 1)) = 19;
```

Function Definition

```
;; Strategy:
;; Struct Decomp [lon : ListOf<Number>]
;; + Accumulator [sofar]

(define (diff-helper sofar slon)
  (cond
    [(empty? slon) sofar]
    [else (diff-helper
            (- sofar (first slon))
            (rest slon))]))
```

Make sure that the invariant is true for the next call of diff-helper

Function Definition (part 2)

```
;; Strategy: Function Composition
(define (diff lon)
  (diff-helper 0 lon))
```

Writing this with a local

```
;; Struct Decomp [lon : ListOf<Number>] + Accumulator [sofar]
(define (diff lon)
  (local
    ((define (diff-helper sofar slon)
       ;; INVARIANT
       ;; slon is a sublist of lon
       ;; sofar is the result of processing the elements
       ;; of lon that precede slon.
       (cond
         [(empty? slon) sofar]
         [else (diff-helper
                 (- sofar (first slon))
                 (rest slon))])))
    (diff-helper 0 lon)))
```

We rewrite the WHERE clause as an invariant.

What might change?

```
(define (diff lon)
  (local
    ((define (diff-helper sofar slon)
       ;; INVARIANT
       ;; slon is a sublist of lon
       ;; sofar is the result of processing the elements
       ;; of lon that precede slon.
       (cond
         [(empty? slon) sofar]
         [else (diff-helper
                (- sofar (first lon))
                (rest slon))])))
    (diff-helper 0 lon)))
```

These are the only things that are specific to subtraction

So Generalize

```
;; (Y X -> Y) Y ListOf<X> -> Y
(define (associate-left fcn base lst)
  (local
    ((define (helper sofar slst)
       ;; INVARIANT:
       ;; slst is a sublist of lst
       ;; sofar is the result of processing the elements
       ;; of lst that precede slst.
       (cond
         [(empty? slon) sofar]
         [else (helper
                 (fcn sofar (first lon))
                 (rest slon))])))
    (diff-helper base slon0)))
```

Then we get

```
(associate-left – a (list x1 ... xn))
 =
(((a – x1) – x2) ... – xn)
```

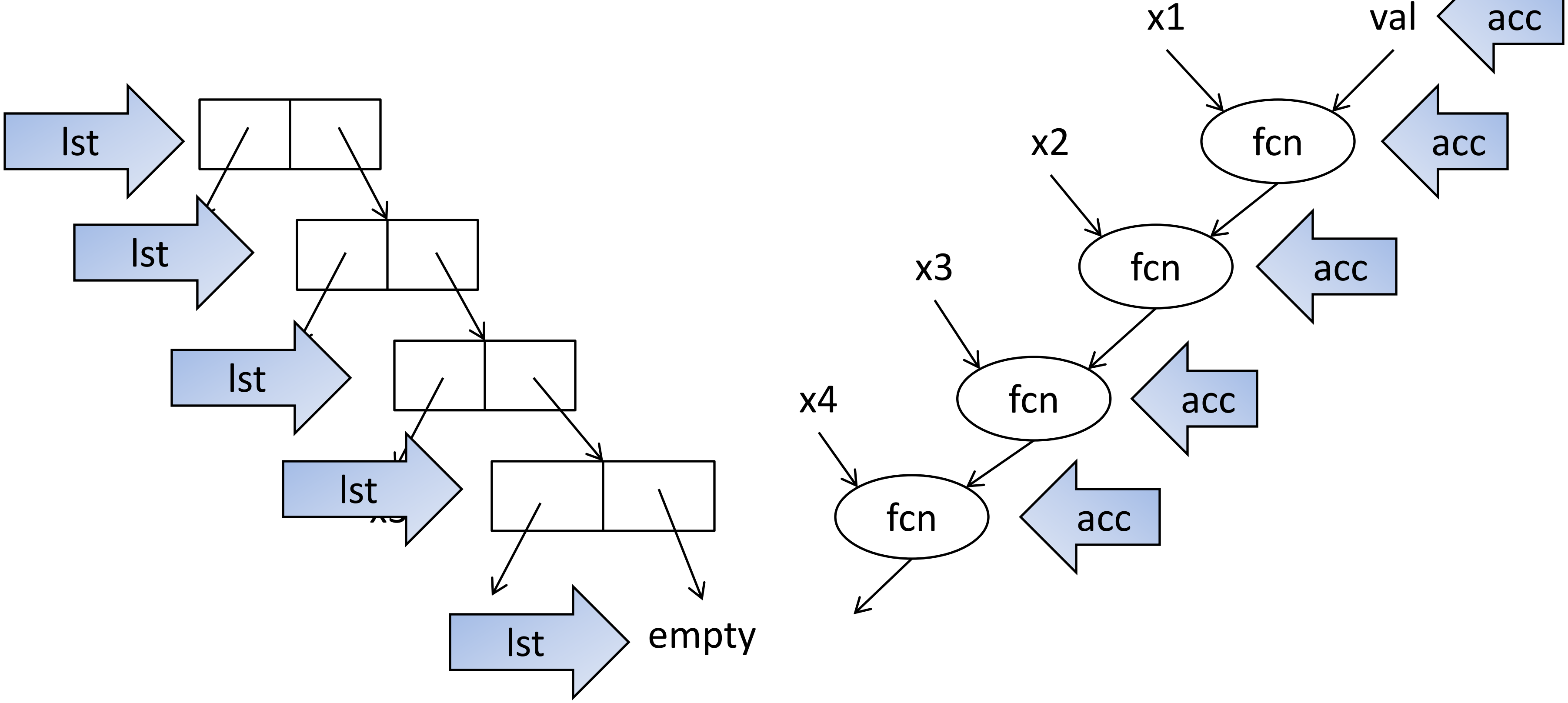
There's an app for that (almost)

```
;; foldl : (X Y -> Y) Y ListOf<X> -> Y
;; (foldl f base (list x1 ... xn))
;; = (f xn (f x_(n-1) ... (f x1 base)))

(define (foldl fcn base lst)
  (local
    ((define (foldl-helper sofar slst)
        ;; INVARIANT
        ;; slst is a sublist of lst
        ;; sofar is the result of processing the elements
        ;; of lst that precede slst.
        (cond
          [(empty? slon) sofar]
          [else (foldl-helper
                  (fcn (first lon) sofar)
                  (rest slon))])))
    (foldl-helper base slon0)))
```

Different order of arguments to the combiner (sorry about that)

Let's watch this work



fcn : X Y -> Y   val : Y

```
foldl : (X Y -> Y) Y ListOf<X> -> Y
```

Defining associate-left in terms of foldl

```
;; (Y X -> X) Y ListOf<X> -> Y
(define (associate-left fcn base lst)
  (foldl
   (lambda (x y) (fcn y x))
   base
   lst))
```

An Application: Simulation

```
;; simulating a process

;; Wishlist:
;; next-state : State Move -> State

;; simulate : State ListOf<Move> -> State
;; given a starting state and a list of
;; moves, find the final state
```

An Application: Simulation

```
;; strategy: structural decomposition on moves
  + accumulator (st)

(define (simulate st moves)
  (cond
    [(empty? moves) st]
    [else
      (simulate
        (next-state st (first moves))
        (rest moves)))]))
```

Or using LOCAL

```
;; simulate : ListOf<Moves> -> State
;; strategy: structural decomposition on moves + accumulator
   (st)
(define (simulate initial-state moves)
  (local
    (;; simulate-helper : State ListOf<Move> -> State
     ;; INVARIANT: st is the state reached by the moves so far
     (define (simulate-helper st moves)
       (cond
         [(empty? moves) st]
         [else
           (simulate-helper
             (next-state st (first moves))
             (rest moves))]))))
    (simulate initial-state moves)))
```

Or using foldl

```
(define (simulate initial-state moves)
  (foldl
   (lambda (move st)
     ;; INVARIANT: st is the state
     ;; reached so far
      (next-state st move))
   initial-state
   moves))
```

Summary

- Accumulators are a useful model for simulations
- Template is limited, but still useful
- Template => foldl abstraction
- comparison of foldr and foldl