

# Using svn

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 4.0

# Basic Workflow

- **svn update**
- **<work>**
- **svn add** <new files>
- **svn commit**
  - or **svn commit -m "Log message"**
- When in doubt:
  - **svn ls**
  - repo browser (or use your web browser)

# Lists vs. Structures

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 4.1

# Recall our pizzas

```
;; A Topping is a String.
```

```
;; A Pizza is a ListOf<Topping>
```

```
;; interp: a pizza is a list of toppings, listed from top to bottom
```

```
;; pizza-fn : Pizza -> ??
```

```
; Given a Pizza, produce ....
```

```
;; (define (pizza-fn p)
```

```
;;   (cond
```

```
;;     [(empty? p) ...]
```

```
;;     [else (... (first p)
```

```
;;               (pizza-fn (rest p)))]))
```

```
;; Examples:
```

```
(define plain-pizza empty)
```

```
(define cheese-pizza (list "cheese"))
```

```
(define anchovies-cheese-pizza (list "anchovies" "cheese"))
```

# What if Racket didn't have cons?

We could still write a data definition:

```
(define-struct topped-pizza (topping base))
```

A Topping is a String.

A Pizza is either

- the string "plain crust"

- (make-topped-pizza Topping Pizza)

Interp:

"plain crust" means a pizza with no toppings

(make-topped-pizza t p) represents the pizza p  
with topping t added on top.

This data definition is *self-referential*

```
(define-struct topped-pizza (topping base))
```

A Topping is a String.

A **Pizza** is either



```
-- the string "plain crust"
```

```
-- (make-topped-pizza Topping Pizza)
```

compare:

A ListOfToppings is either



```
-- empty
```

```
-- (cons Topping ListOfToppings)
```

# Examples

`"plain crust"`

`(make-topped-pizza "cheese" "plain-crust")`

`(make-topped-pizza "anchovies"  
 (make-topped-pizza "cheese" "plain-crust")))`

`(make-topped-pizza "onions"  
 (make-topped-pizza "anchovies"  
 (make-topped-pizza "cheese" "plain crust")))))`

# Template for pizza functions

`pizza-fn : Pizza -> ??`

Given a `Pizza`, produce ....

```
(define (pizza-fn p)
  (cond
    [(plain-pizza? p) ...]
    [else (... (topped-pizza-topping p)
                (pizza-fn
                  (topped-pizza-base p))))]))
```

Add to wishlist:

`plain-pizza? : Pizza -> Boolean`

returns true iff the given pizza is a plain pizza

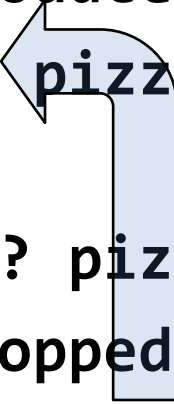


This template is *self-referential*

`pizza-fn : Pizza -> ??`

Given a Pizza, produce ....

```
(define (pizza-fn pizza)
  (cond
    [(plain-pizza? pizza) ...]
    [else (... (topped-pizza-topping pizza)
                (pizza-fn
                  (topped-pizza-base pizza))))])
```



We also call this a  
*recursive* template

# Lists vs Structures: Data Definitions

A ListOfToppings (LoT) is  
either

-- empty  
-- (cons Topping LoT)




Interp:

-- empty means a pizza with  
no toppings  
-- (cons t p)  
represents the pizza p with  
topping t added on top.

A Pizza is either

-- the string "plain crust"  
-- (make-topped-pizza  
Topping Pizza)




Interp:


"plain crust" means a pizza  
with no toppings  
(make-topped-pizza t p)  
represents the pizza p with  
topping t added on top.

# Lists vs. Structures: Templates

```
pizza-fn : Pizza -> ??  
(define (pizza-fn p)  
  (cond  
    [(empty? p)  
     ...]  
    [else  
     (...  
      (first p)  
      (pizza-fn  
        (rest p))))]))
```



```
pizza-fn : Pizza -> ??  
(define (pizza-fn p)  
  (cond  
    [(plain pizza? p)  
     ...]  
    [else  
     (...  
      (topped-pizza-  
        topping p)  
      (pizza-fn  
        (topped-pizza-base  
          p))))]))
```



# Lists vs. Structures: The Choice

- Use structures for compound information with a fixed size or fixed number of components.
- Use lists for homogeneous sequences of data items.
  - so we'll use mostly lists
  - DON'T use lists for data of fixed size or a fixed number of components
- Each language has its own idioms
  - some don't have lists at all
  - some have other ways of representing sequences– use them when possible

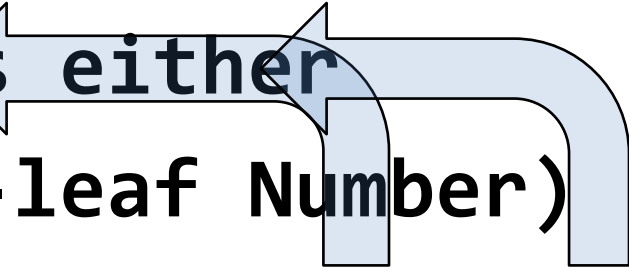
# Trees

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 4.2

# Binary Trees

```
(define-struct leaf (datum))  
(define-struct node (lson rson))
```

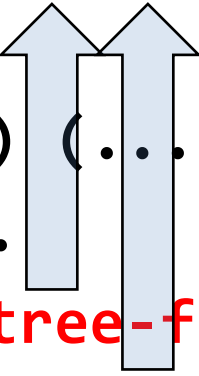
```
;; A Tree is either  
;; -- (make-leaf Number)  
;; -- (make-node Tree Tree)
```



# Template

tree-fn : Tree -> ???

```
(define (tree-fn t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
                  (tree-fn (node-lson t))
                  (tree-fn (node-rson t)))]))
```



*Self-reference in the data definition  
leads to self-reference in the template;  
Self-reference in the template leads to  
self-reference in the code.*

# The template questions

`tree-fn : Tree -> ???`

`(define (tree-fn t)`

`(cond`

`[(leaf? t) (... (leaf-datum t))]`

`[else (...`

`(tree-fn (node-lson t))`

`(tree-fn (node-rson t) ))]))`

What's the answer  
for a leaf?

If you knew the answers for the 2  
sons, how could you find the answer  
for the whole tree?



# Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs one <a href="#"><u>cond</u></a> clause for each subclass 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.

# leaf-sum

**leaf-sum** : Tree -> Number

```
(define (leaf-sum t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (+
            (leaf-sum (node-lson t))
            (leaf-sum (node-rson t)))])])
```

What's the answer  
for a leaf?

If you knew the answers for the 2  
sons, how could you find the answer  
for the whole tree?

# leaf-max

**leaf-sum** : Tree -> Number

```
(define (leaf-sum t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (max
              (leaf-sum (node-lson t))
              (leaf-sum (node-rson t))))]))
```

What's the answer  
for a leaf?

If you knew the answers for the 2  
sons, how could you find the answer  
for the whole tree?

# leaf-min

**leaf-sum** : Tree -> Number

```
(define (leaf-sum t)
  (cond
    [(leaf? t) (leaf-datum t)]
    [else (min
              (leaf-sum (node-lson t))
              (leaf-sum (node-rson t))))]))
```

What's the answer  
for a leaf?

If you knew the answers for the 2  
sons, how could you find the answer  
for the whole tree?

# From templates to folds

- Can always build a fold function
- Just fill in the blanks in the template

# Template

`tree-fn : A B C Tree -> ???`

```
(define (tree-fn a b c t)
  (cond
    [(leaf? t) (... (leaf-datum t))]
    [else (...
      (tree-fn a b c (node-lson t))
      (tree-fn a b c (node-rson t)))]))
```

# Template → tree-fold

**tree-fold** : ... Tree -> ???

```
(define (tree-fold combiner base t)
  (cond
    [(leaf? t) (base (leaf-datum t))]
    [else (combiner
      (tree-fold combiner base
        (node-lson t))
      (tree-fold combiner base
        (node-rson t))))]))
```

# What's the contract for tree-fold?

## tree-fold

: contract for combiner contract for base Tree  $\rightarrow$  X

```
(define (tree-fold combiner base t)
  (cond
    [(leaf? t) (base (leaf-datum t))]
    [else (combiner
                  (tree-fold combiner base
                             (node-lson t))
                  (tree-fold combiner base
                             (node-rson t)))])])
```

The diagram illustrates the recursive calls of `tree-fold`. It shows a tree structure with nodes containing 'X'. Arrows point from the code to the corresponding parts of the tree. The 'base' function is shown as `(Number  $\rightarrow$  X)` and the 'combiner' function as `(X X  $\rightarrow$  X)`.



# Be sure to reconstruct the original functions!

```
(define (tree-sum t)
  (tree-fold + (lambda (n) n) t))
```

```
(define (tree-min t)
  (tree-fold min (lambda (n) n) t))
```

```
(define (tree-max t)
  (tree-fold max (lambda (n) n) t))
```

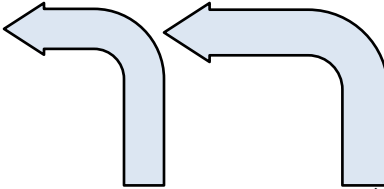
# Demo

- 04-1-tree-fold.rkt
- Using tree-fold, write number-of-nodes, increment-all.

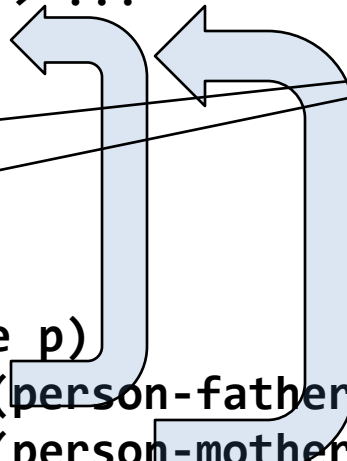
# Another example of trees: Ancestor Trees

```
(define-struct person (name father mother))
```

```
;; A Person is either  
;; -- "Adam"  
;; -- "Eve"  
;; -- (make-person String Person Person)
```



```
;; person-fn : Person -> ???  
(define (person-fn p)  
  (cond  
    [(adam? p) ...]  
    [(eve? p) ...]  
    [else (...  
      (person-name p)  
      (person-fn (person-father p))  
      (person-fn (person-mother p)))])])
```



Good idea to use  
wishlist functions  
here.

*Self-reference in the data  
definition leads to self-  
reference in the template;  
Self-reference in the  
template leads to self-  
reference in the code.*

# From template to fold:

```
;; person-fn : A B C Person -> ???  
(define (person-fn a b c p)  
  (cond  
    [(adam? p) ...]  
    [(eve? p) ...]  
    [else (  
      (person-name p)  
      (person-fn a b c (person-father p))  
      (person-fn a b c (person-mother p)))]))
```

# From template to fold:

```
;; person-fold : ... Person -> ???  
(define (person-fold adam-val eve-val combiner p)  
  (cond  
    [(adam? p) adam-val]  
    [(eve? p) eve-val]  
    [else (combiner  
              (person-name p)  
              (person-fold adam-val eve-val combiner  
                            (person-father p))  
              (person-fold adam-val eve-val combiner  
                            (person-mother p))))]))
```

# What's the contract for person-fold?

```
;; person-fold
```

```
;; : X X (String X X -> X) Person -> X
```

```
(define (person-fold adam-val eve-val combiner p)
```

```
  (cond
```

```
    [(adam? p) adam-val]
```

```
    [(eve? p) eve-val]
```

```
    [else (combiner
```

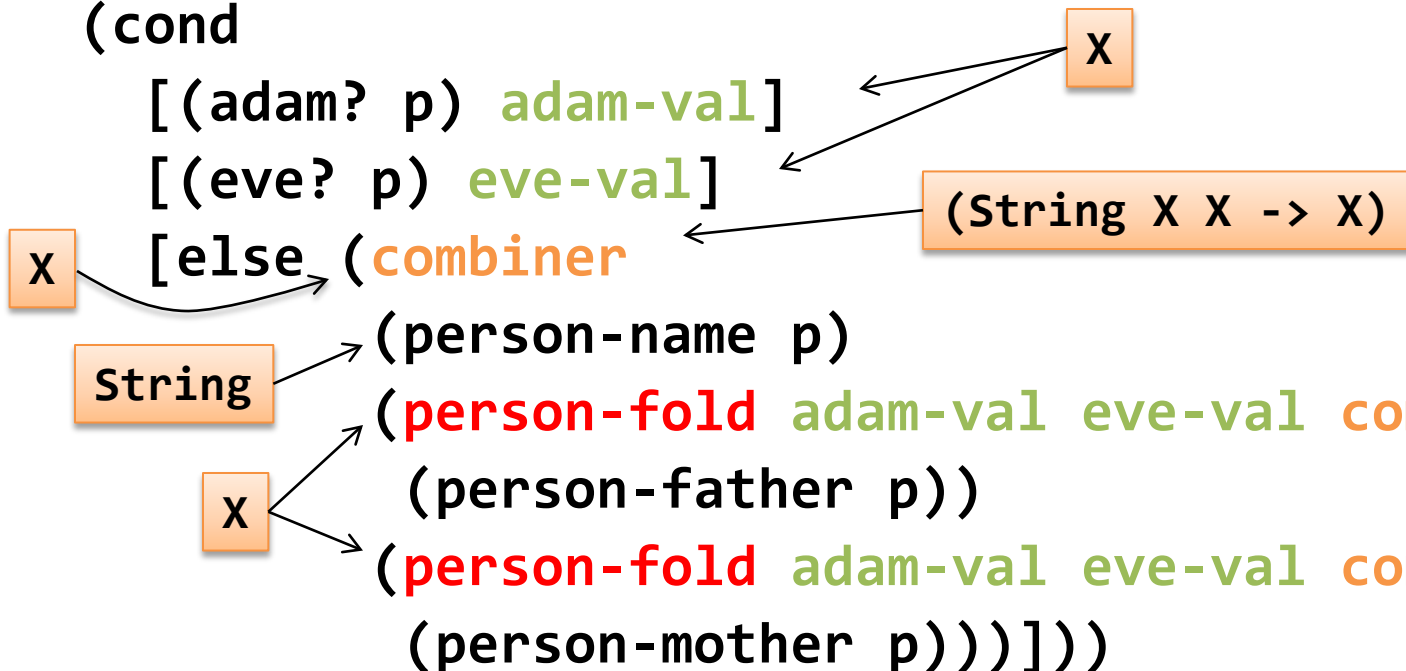
```
      (person-name p)
```

```
      (person-fold adam-val eve-val combiner
```

```
        (person-father p))
```

```
      (person-fold adam-val eve-val combiner
```

```
        (person-mother p))))]
```



# Mutually-Recursive Data Definitions

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 4.3

# Mutually Recursive Data Definitions

- Sometimes two kinds of data are intertwined
- In this lesson, we'll consider an easy example: alternating lists
- An alternating list is a list whose elements alternate between numbers and strings



# Data Definitions

**;; A ListOfAlternatingNumbersAndStrings  
(LANS) is one of:**

**;; -- empty**

**;; -- (cons Number LASN)**

**;; A ListOfAlternatingStringsAndNumbers  
(LASN) is one of:**

**;; -- empty**

**;; -- (cons String LANS)**

# Examples

```
empty is a LANS
  (cons 11 empty) is a LANS
    (cons "foo" (cons 11 empty)) is a LASN
      (cons 23 (cons "foo" (cons 11 empty))) is a LANS
        (cons "bar" (cons 23 (cons "foo" (cons 11 empty)))) is a LASN
```

These data definitions are *mutually recursive*

```
;; A ListOfAlternatingNumbersAndStrings  
  (LANS) is one of:
```

```
;; -- empty
```

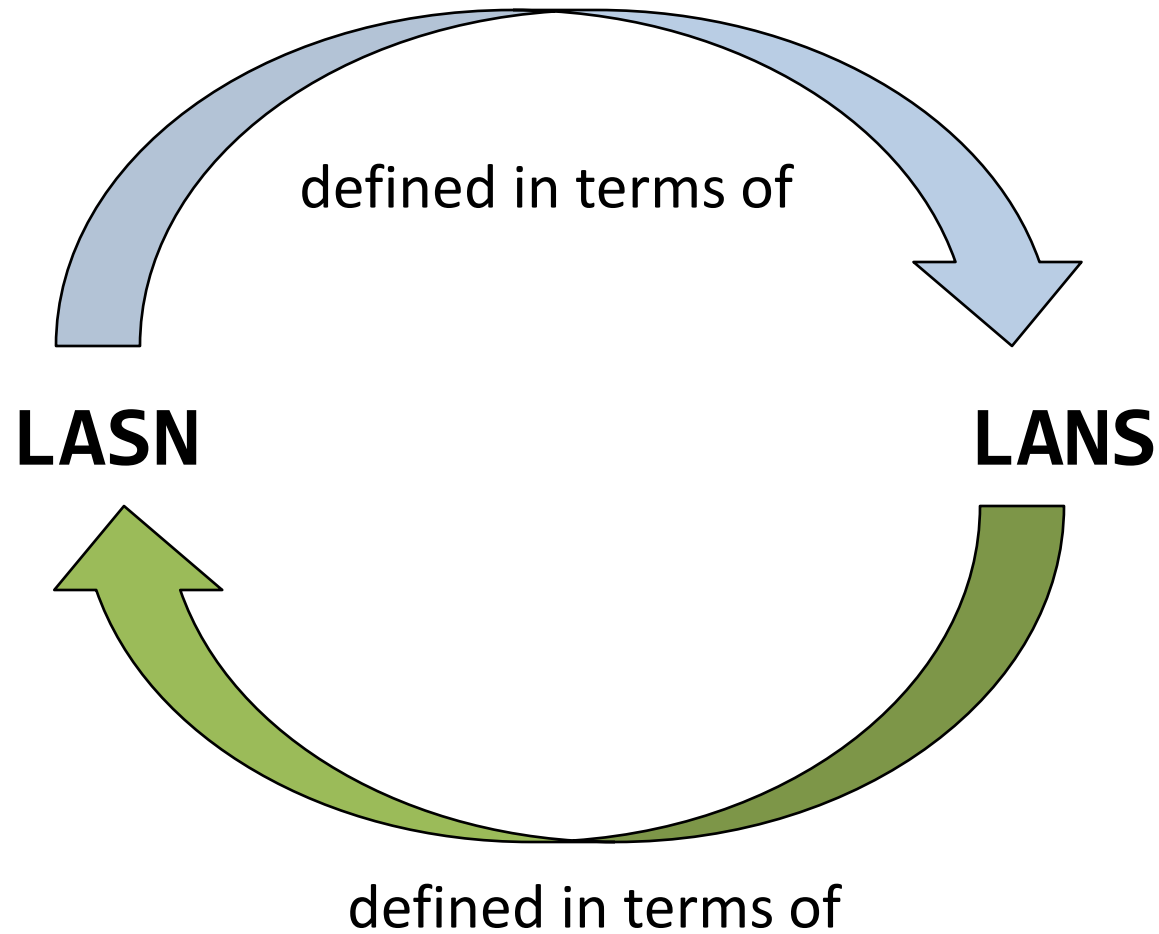
```
;; -- (cons Number LASN)
```

```
;; A ListOfAlternatingStringsAndNumbers  
  (LASN) is one of:
```

```
;; -- empty
```

```
;; -- (cons String LANS)
```

# This is mutual recursion



# Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many <a href="#"><u>cond</u></a> 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 use mutually-recursive references?	Formulate ``natural recursions'' for the template to represent the mutually-recursive references of the data definition.

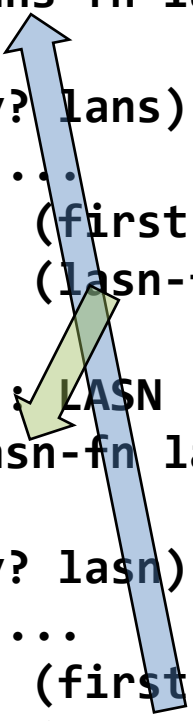
# Templates come in pairs

```
;; lans-fn : LANS -> ??  
;; (define (lans-fn lans)  
;;   (cond  
;;     [(empty? lans) ...]  
;;     [else (...  
;;       (first lans)  
;;       (lans-fn (rest lans)))]))
```

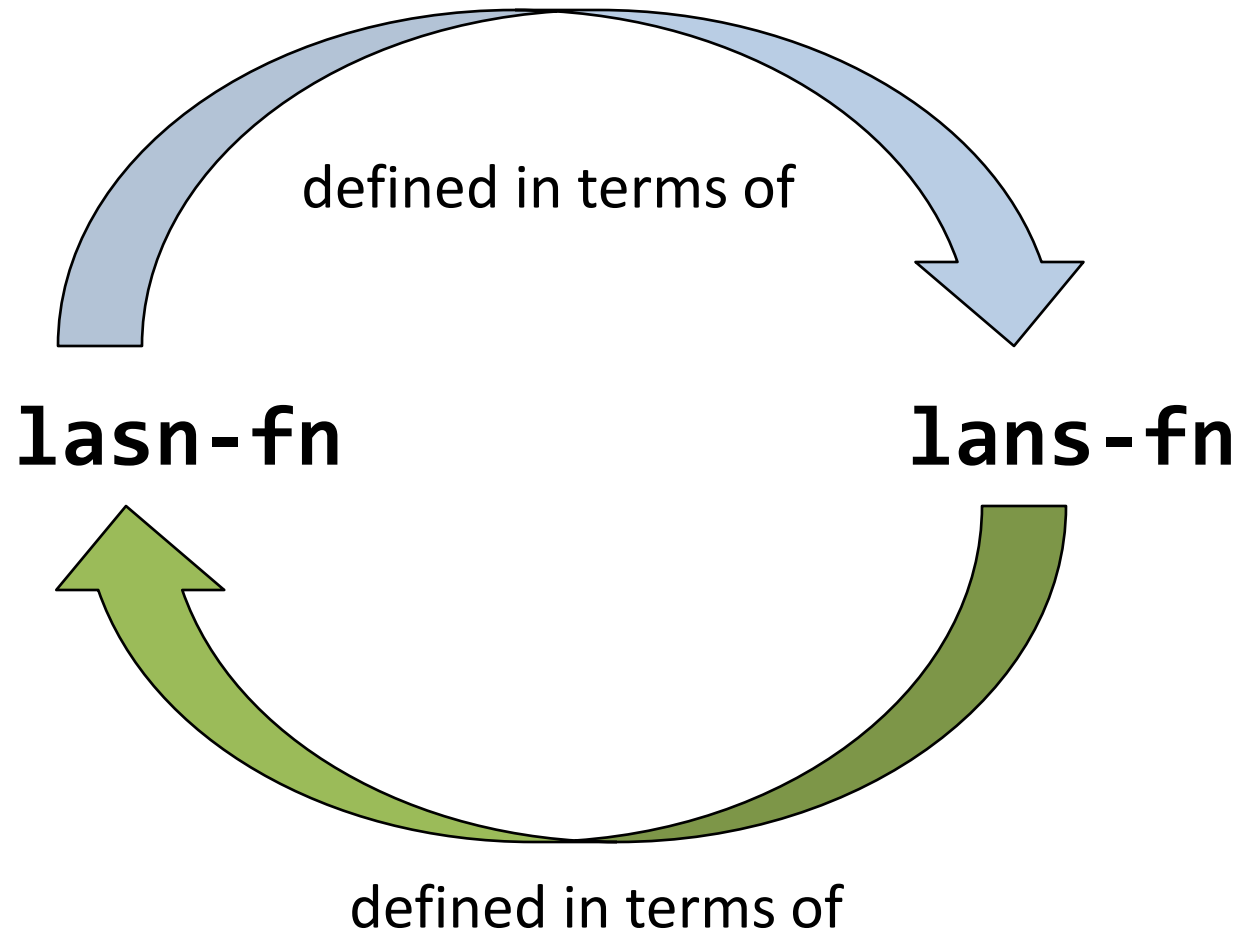
```
;; lasn-fn : LASN -> ??  
;; (define (lasn-fn lasn)  
;;   (cond  
;;     [(empty? lasn) ...]  
;;     [else (...  
;;       (first lasn)  
;;       (lans-fn (rest lasn)))]))
```

# Templates are mutually recursive

```
;; lans-fn : LANS -> ??  
;; (define (lans-fn lans)  
;;   (cond  
;;     [(empty? lans) ...]  
;;     [else (...  
;;       (first lans)  
;;       (lans-fn (rest lans)))]))  
  
;; ;; lasn-fn : LASN -> ??  
;; (define (lasn-fn lasn)  
;;   (cond  
;;     [(empty? lasn) ...]  
;;     [else (...  
;;       (first lasn)  
;;       (lans-fn (rest lasn)))]))
```



# This is mutual recursion





# The template questions

```
;; lans-fn : LANS -> ??  
;; (define (lans-fn lans)  
;;   (cond  
;;     [(empty? lans) ...]  
;;     [else (...  
;;              (first lans)  
;;              (lans-fn (rest lans)))]))
```

What is the answer for the empty LANS?

If you knew the answer for the LASN inside the LANS, what would the answer be for the whole LANS?

```
;; ;; lasn-fn : LASN -> ??  
;; (define (lasn-fn lasn)  
;;   (cond  
;;     [(empty? lasn) ...]  
;;     [else (...  
;;              (first lasn)  
;;              (lans-fn (rest lasn)))]))
```

What is the answer for the empty LASN?

If you knew the answer for the LANS inside the LASN, what would the answer be for the whole LASN?

# One function, one task

- Each function deals with exactly one data definition.
- So functions will come in pairs
- Write contracts and purpose statements together, **or**
- Write one, and the other one will appear as a wishlist function

# Example

**lans-sum : LANS -> Number**

**Returns the sum of all the numbers  
in the given Lans**

**lasn-sum : LASN -> Number**

**Returns the sum of all the numbers  
in the given Lasn**

# Example

```
(lans-sum  
  (cons 23  
    (cons "foo"  
      (cons 11 empty)))) = 34
```

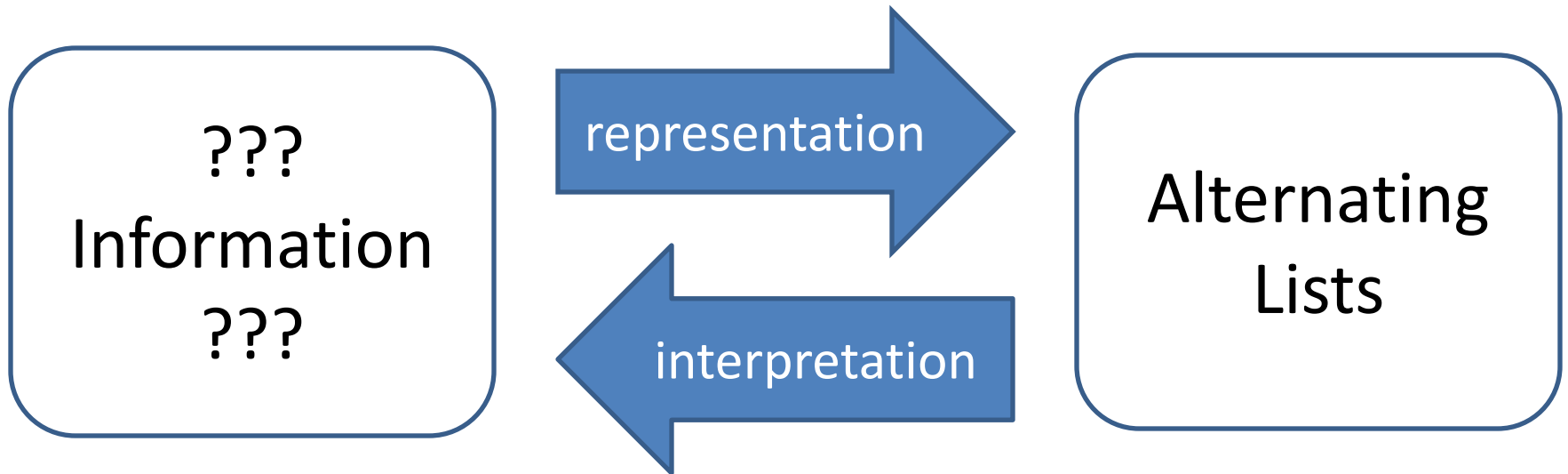
```
(lasn-sum  
  (cons "bar"  
    (cons 23  
      (cons "foo"  
        (cons 11 empty)))))) = 34
```

# Strategy and Function Definitions

```
;; strategy: structural decomposition on LANS and LASN
;; lans-sum : LANS -> Number
(define (lans-sum lans)
  (cond
    [(empty? lans) 0]
    [else (+
            (first lans)
            (lans-sum (rest lans)))]))

;; lasn-sum : LASN -> Number
(define (lasn-sum lasn)
  (cond
    [(empty? lasn) 0]
    [else (lans-sum (rest lasn))]))
```

# What are alternating lists good for?



Answer: Not much! Don't use them! Use a list of structs instead

But they make a good example of mutually-recursive data definitions

# Lists of Lists

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 4.4

# S-expressions

- An S-expression is either a string or a list of S-expressions.
- So if it's a list, it could contain strings, or lists of strings, or lists of lists of strings, etc.
- Think of it as a nested list, where there's no bound on how deep the nesting can get.



# Examples

**"alice"**

**"bob"**

**"carole"**

**("alice" "bob")**

**(( "alice" "bob" ) "carole")**

**("dave" ( "alice" "bob" ) "carole")**

**(( "alice" "bob" )**

**(( "ted" "carole" )))**

# Data Definition

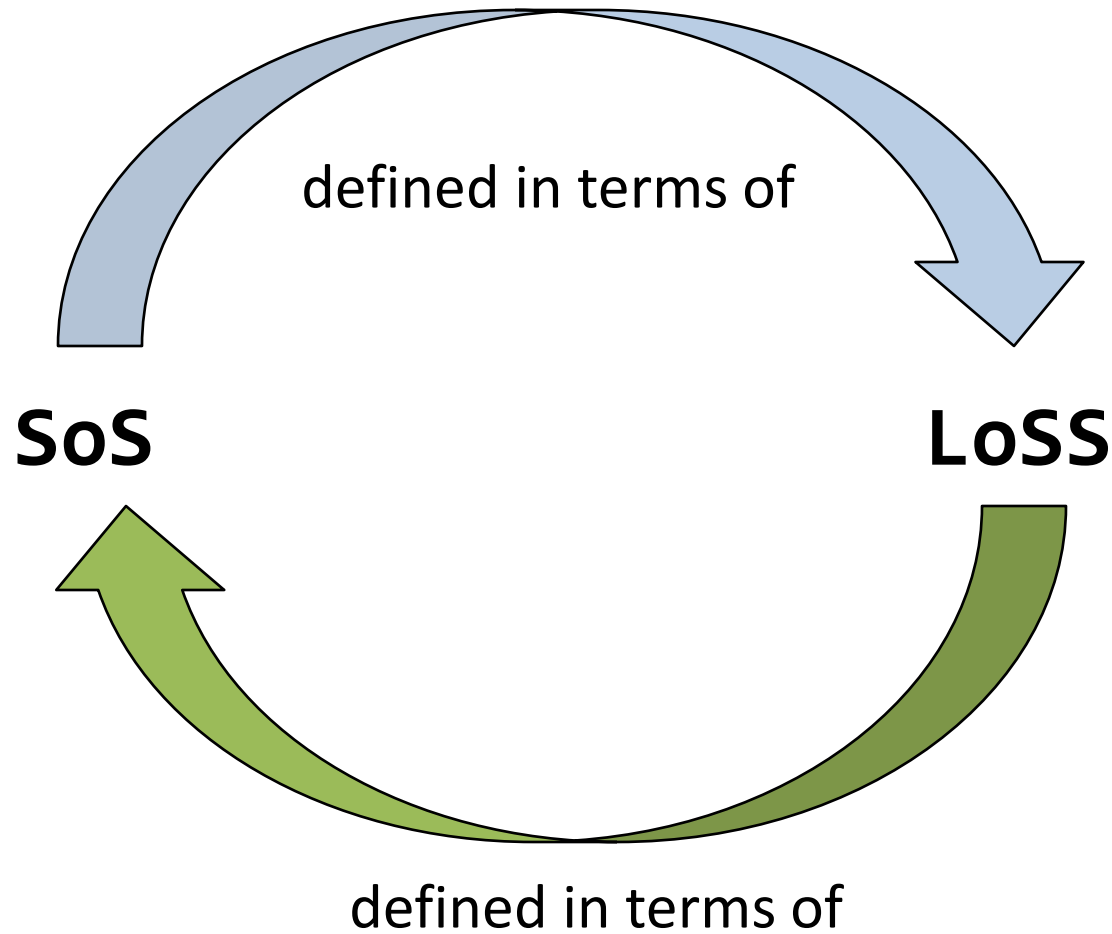
An S-expression of Strings (SoS) is either

- a String
- a List of SoS's (LoSS)

A List of SoS's (LoSS) is either

- empty
- (cons SoS LoSS)

# This is mutual recursion



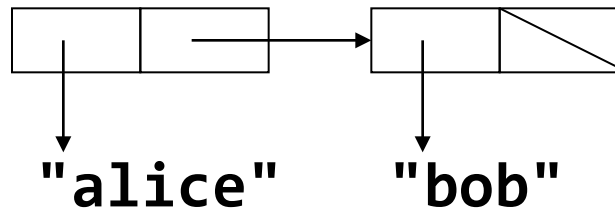
# Data Structures

**"alice"**

**"bob"**

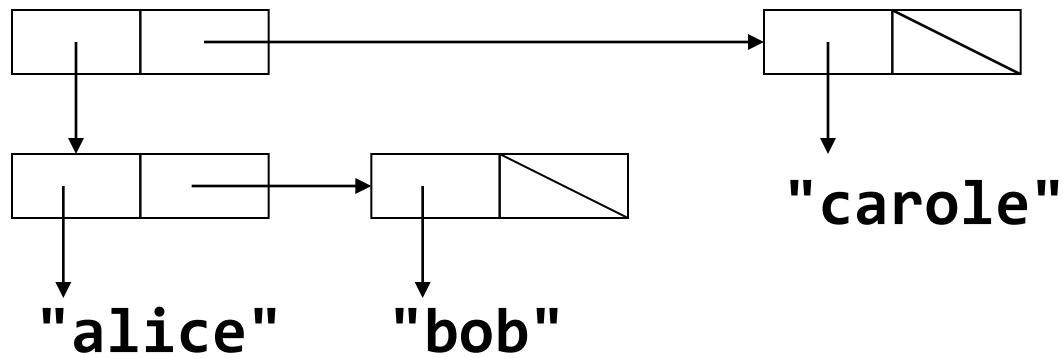
**"carole"**

**("alice" "bob")**



# Data Structures

**(( "alice" "bob") "carole")**

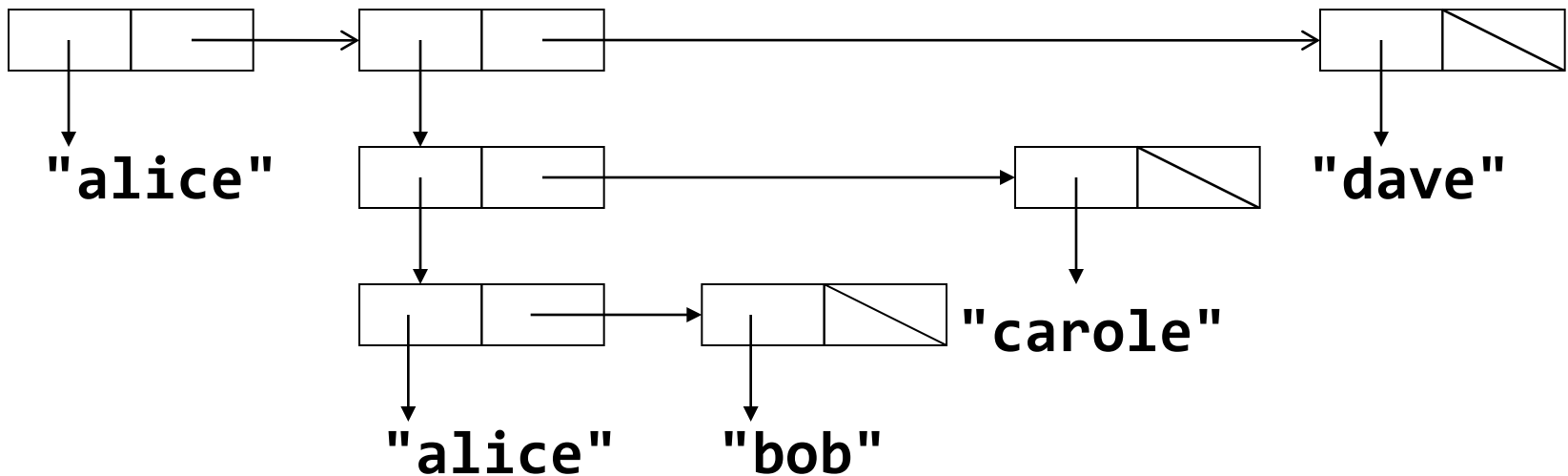


# Data Structures (cont'd)

( "alice"

```
(( "alice" "bob") "carole")
```

**"dave")**



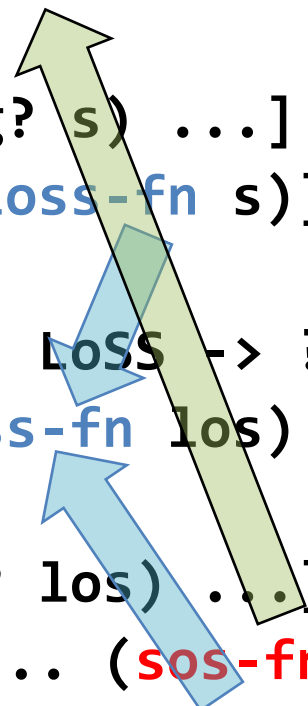
# Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many <a href="#"><u>cond</u></a> 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 use mutually-recursive references?	Formulate ``natural recursions'' for the template to represent the mutually-recursive references of the data definition.

# Template: functions come in pairs

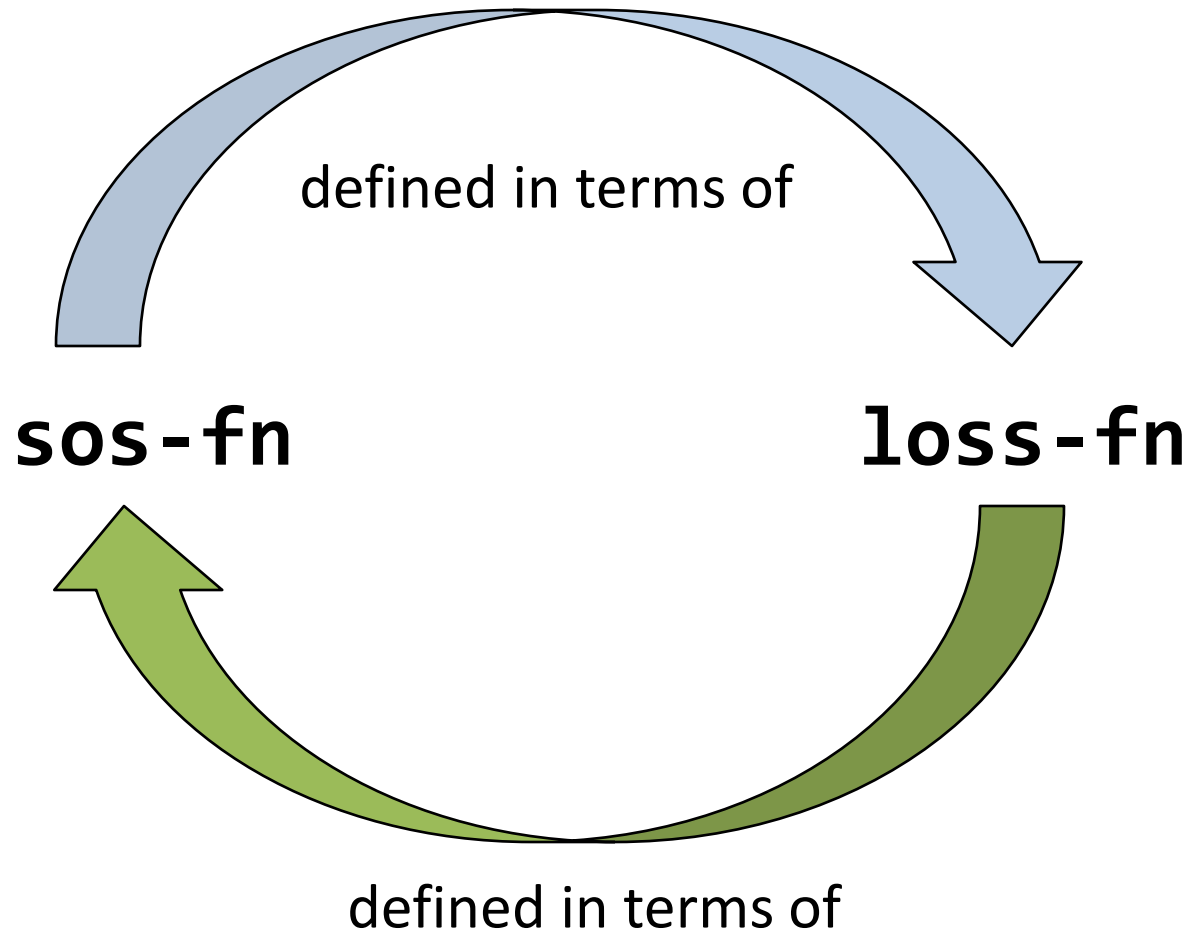
```
;; sos-fn : SoS -> ??  
(define (sos-fn s)  
  (cond  
    [(string? s) ...]  
    [else (loss-fn s)]))
```

```
;; loss-fn : LoSS -> ??  
(define (loss-fn los)  
  (cond  
    [(empty? los) ...]  
    [else (... (sos-fn (first los))  
                (loss-fn (rest los)))]))
```





# This is mutual recursion



# One function, one task

- Each function deals with exactly one data definition.
- So functions will come in pairs
- Write contracts and purpose statements together, **or**
- Write one, and the other one will appear as a wishlist function

# occurs-in?

```
:: occurs-in? : Sos String -> Boolean  
:: returns true iff the given string occurs somewhere in  
the given sos.  
  
:: occurs-in-loss? : Loss String -> Boolean  
:: returns true iff the given string occurs somewhere in  
the given loss.
```

# Examples/Tests

```
(check-equal?  
  (occurs-in? "alice" "alice")  
  true)
```

```
(check-equal?  
  (occurs-in? "bob" "alice")  
  false)
```

```
(check-equal?  
  (occurs-in?  
    (list "alice" "bob")  
    "cathy")  
  false)
```

```
(check-equal?  
  (occurs-in?  
    (list (list "alice" "bob")  
          "carole")  
    "bob")  
  true)
```

```
(check-equal?  
  (occurs-in?  
    (list "alice"  
          (list (list "alice" "bob")  
                "dave")  
          "eve")  
    "bob")  
  true)
```

# Livcoding: sos-and-loss.rkt

# The S-expression pattern

Can do this for things other than strings:

**An Sexp<X> is either**

- an X**
- a ListOf<Sexp<X>>**

**A ListOf<Sexp<X>> is either**

- empty**
- (cons Sexp<X> ListOf<Sexp<X>>)**

# The Template for Sexp<X>

```
;; sexp-fn : Sexp<X> -> ??  
(define (sexp-fn s)  
  (cond  
    [(X? s) ...]  
    [else (losexp-fn s)]))  
  
;; losexp-fn : ListOf<Sexp<X>> -> ??  
(define (losexp-fn los)  
  (cond  
    [(empty? los) ...]  
    [else (... (sexp-fn (first los))  
                (losexp-fn (rest los)))]))
```

# Sexp of Sardines

**An SoSardines is either**

- a Sardine**
- a LoSSardines**

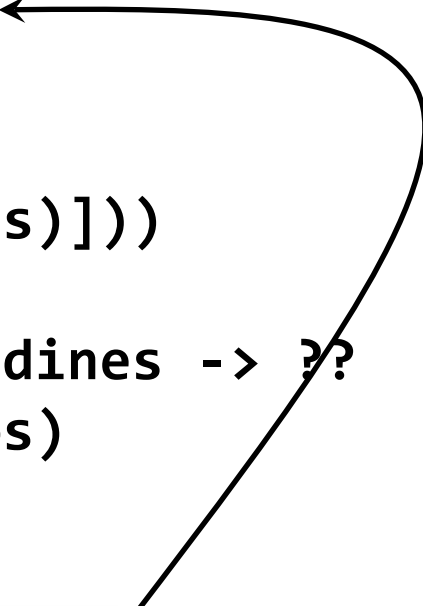
**A LoSSardines is either**

- empty**
- (cons SoSardines  
LoSSardines)**



# The Template for SoSardines

```
;; sosard-fn : SoSardines -> ??  
(define (sosard-fn s) ←  
  (cond  
    [(sardine? s) ...]  
    [else (lossard-fn s)]))  
  
;; lossard-fn : LoSSardines -> ??  
(define (lossard-fn los)  
  (cond  
    [(empty? los) ...]  
    [else (... (sosard-fn (first los))  
                (lossard-fn (rest los)))]))
```



# Summary

- Nested Lists occur all the time
- Mutually recursive data definitions
- Mutual recursion in the data definition leads to mutual recursion in the template
- Mutual recursion in the template leads to mutual recursion in the code

# More Examples

```
:: number-of-strings : Sos -> Number  
:: number-of-strings-in-loss : Loss -> Number  
:: returns the number of strings in the given sos or  
loss.
```

```
:: characters-in : Sos -> Number  
:: characters-in-loss : Loss -> Number  
:: returns the total number of characters in the strings  
in the given sos or loss.
```

```
:: number-of-sardines : SoSardines -> Number  
:: returns the total number of sardines in the given  
SoSardines.
```

# Descendant Trees

CS 5010 Program Design Paradigms

“Bootcamp”

Lesson 4.5

# Ancestor Trees

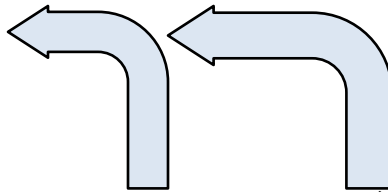
```
(define-struct person (name father mother))
```

```
;; A Person is either
```

```
;; -- "Adam"
```

```
;; -- "Eve"
```

```
;; -- (make-person String Person Person)
```



```
;; person-fn : Person -> ???
```

```
(define (person-fn p)
```

```
  (cond
```

```
    [(adam? p) ...]
```

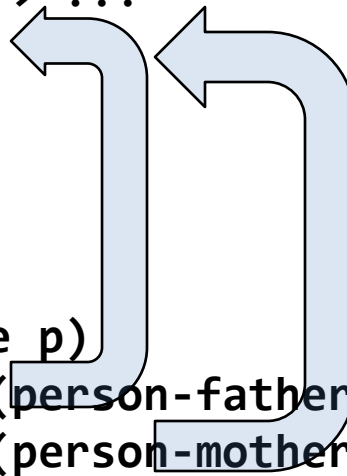
```
    [(eve? p) ...]
```

```
    [else (...
```

```
      (person-name p)
```

```
      (person-fn (person-father p))
```

```
      (person-fn (person-mother p))))]
```



# A Different Info Analysis: Descendant Trees

```
(define-struct person (name children))
```

```
;; A Person is a
```

```
;; (make-person String ListOfPersons)
```

```
;; A ListOfPersons (LoP) is one of
```

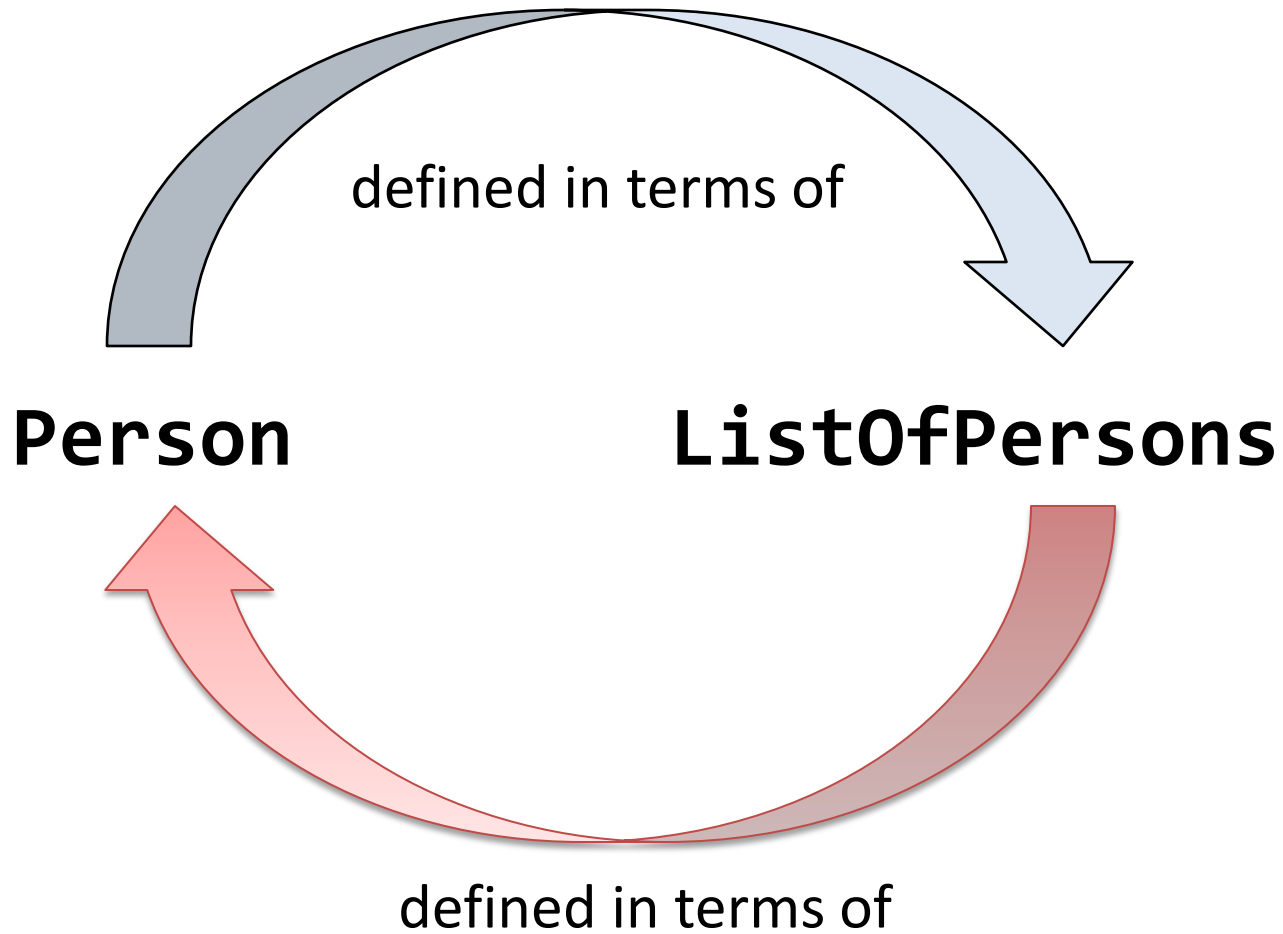
```
;; -- empty
```

```
;; -- (cons Person ListOfPersons)
```



Two *mutually recursive*  
data definitions

# This is mutual recursion



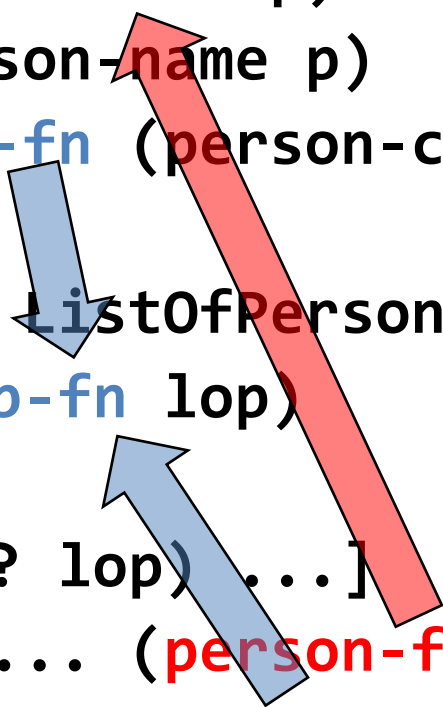
# Recipe for Templates

Question	Answer
Does the data definition distinguish among different subclasses of data?	Your template needs as many <a href="#"><u>cond</u></a> 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 with which it is mutually recursive?	Formulate ``natural recursions'' for the template to represent the mutually-recursive references of the data definition.



# Template: functions come in pairs

```
;; person-fn : Person -> ??  
(define (person-fn p)  
  (... (person-name p)  
        (lop-fn (person-children p))))  
  
;; lop-fn : ListOfPersons -> ??  
(define (lop-fn lop)  
  (cond  
    [(empty? lop) ...]  
    [else (... (person-fn (first lop))  
                (lop-fn (rest lop)))]))
```



# The template questions

```
;; person-fn : Person -> ??  
(define (person-fn p)  
  (... (person-name p)  
        (lop-fn (person-children p)))))
```

Given the answer for a person's children, how do we find the answer for the person?

```
;; lop-fn : ListOfPersons -> ??  
(define (lop-fn lop)  
  (cond  
    [(empty? lop) ...]  
    [else (... (person-fn (first lop))  
                (lop-fn (rest lop)))])])
```

What's the answer for the empty LoP?

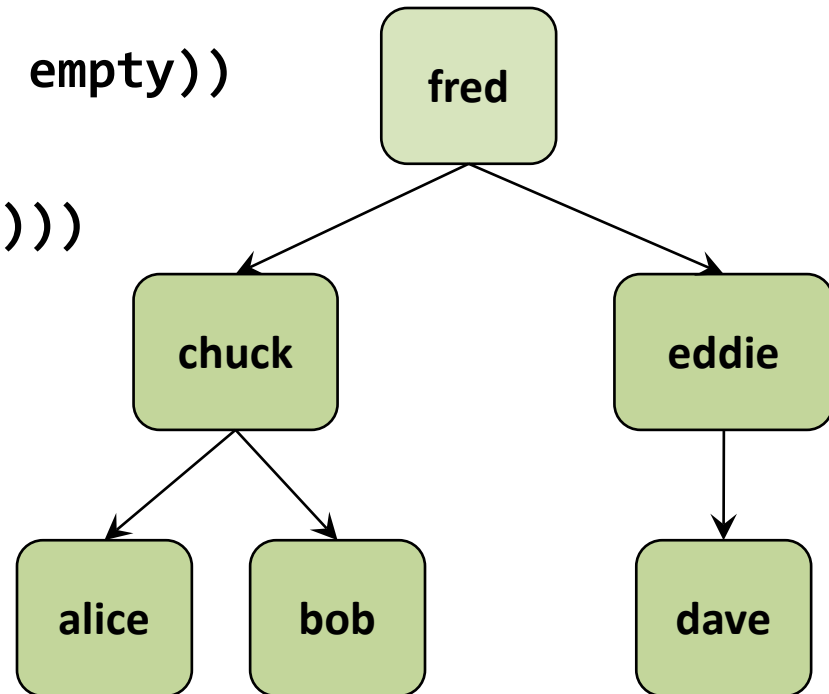
Given the answer for the first person in the list and the answer for the rest of the people in the list, how do we find the answer for the whole list?

# Examples

```
(define alice (make-person "alice" empty))  
(define bob (make-person "bob" empty))  
(define chuck (make-person "chuck" (list alice bob)))
```

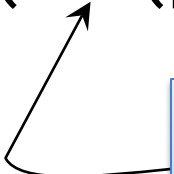
```
(define dave (make-person "dave" empty))  
(define eddie  
  (make-person "eddie" (list dave)))
```

```
(define fred  
  (make-person  
    "fred"  
    (list chuck eddie)))
```



# Grandchildren

```
;; grandchildren : Person -> LoP
;; produce a list of the grandchildren of the given
   person.
;; (grandchildren fred) = (list alice bob dave)
(define (grandchildren p)
  (... (person-children p)))
```



Q: Given p's children, how do we find p's grandchildren?

A: We need a function which, given a list of persons, produces a list of all their children

# all-children

```
;; all-children : LoP -> LoP
;; given a list of persons, produces the list of all
   their children.
;; (all-children (list fred eddie))
;; = (list chuck eddie dave)
(define (all-children lop)
  (cond
    [(empty? lop) empty]
    [else (append
              (person-children (first lop))
              (all-children (rest lop))))]))
```

# descendants

- Given a person, find all his/her descendants.
- What's a descendant?
  - a person's children are his/her descendants.
  - any descendant of any of a person's children is also that person's descendant.
- Hey: this definition is recursive!

# Contracts and Purpose Statements

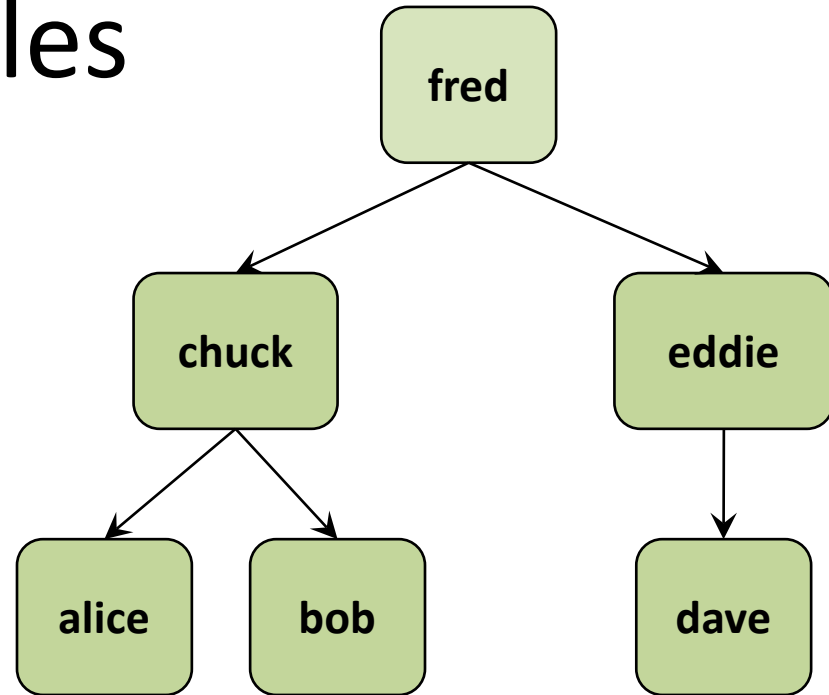
**:: descendants : Person -> LoP**

**:: given a Person, produce the list of his/her  
descendants**

**:: all-descendants : LoP -> LoP**

**:: given a ListOfPersons, produce the list of all their  
descendants**

# Examples



`(descendants fred)`

`= (list chuck eddie alice bob dave)`

`(all-descendants (list chuck eddie))`

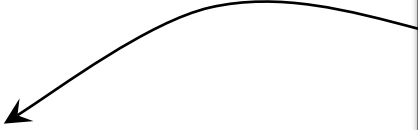
`= (list alice bob dave)`



# Function Definitions

```
(define (descendants p)
  (append
    (person-children p)
    (all-descendants (person-children p))))
```

This comes  
right from the  
definition!



```
(define (all-descendants lop)
  (cond
    [(empty? lop) empty]
    [else (append
      (descendants (first lop))
      (all-descendants (rest lop)))]))
```

# Tests

```
(check-equal?  
  (descendants fred)  
  (list chuck eddie alice bob dave))
```

```
(check-equal?  
  (all-descendants (list chuck eddie))  
  (list alice bob dave))
```

# Are these good tests?

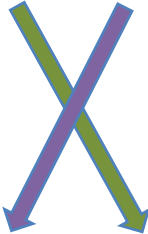
- How could a program fail these tests but still be correct?
- Answer: it could produce the list of descendants in a different order.

# Better Tests

```
(require "sets.rkt")    ;; or whatever...
```

```
(check set-equal?  
  (descendants fred)  
  (list chuck eddie alice bob dave))
```

```
(check set-equal?  
  (descendants fred)  
  (list chuck eddie alice dave bob))
```



```
(check set-equal?  
  (all-descendants (list chuck eddie))  
  (list alice bob dave))
```

# Examples

- Rewrite all-children using the list abstractions from last week.
- Challenge: can you rewrite all-descendants using the list abstractions from last week?