

# Lists vs. Structures

CS 5010 Program Design Paradigms  
“Bootcamp”  
Lesson 6.1



© Mitchell Wand, 2012-2015

This work is licensed under a [Creative Commons Attribution-NonCommercial 4.0 International License](https://creativecommons.org/licenses/by-nc/4.0/).

# Module Introduction

- In this module we will learn about two related topics:
  - branching structures, such as trees
  - mutually recursive data definitions, such as lists of alternating strings and numbers.

# Module Outline

- Lesson 6.1 begins by considering alternative representations for sequence information
  - This is a warm-up for Lessons 6.2-6.3
- Lessons 6.2 and 6.3 show how to represent information that has a naturally branching structure, such as trees
- Lesson 6.4 introduces mutually-recursive data definitions
- Lesson 6.5 applies these ideas to S-expressions
  - S-expressions are nested lists
  - These are the basis for XML and JSON
- Lesson 6.6 combines all these ideas into a case study

# Module 06

## 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

- We've already studied how to represent sequences of data using lists.
- In this lesson, we will explore how to represent sequences of data using structures, like those we studied in Week 1, instead of lists.
- This is useful because many widely-used languages do not have built-in lists that we can use.

# Learning Objectives for this Lesson

- At the end of this lesson the student should be able to:
  - convert a data definition using the **ListOfX** pattern to a recursive data definition using structures
  - write a template for a recursive data definition using structures

# Recall our pizzas

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

```
;; A Pizza is a ListOfTopping
```

```
;; 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"))
```

In Module 4, we represented a pizza as a list of toppings. This week, we will use this example to study the structure representation.

# What if Racket didn't have cons?

- If Racket didn't have **cons**, we could still represent pizzas as mixed data, using a structure to represent a non-empty pizza.
- On the next slide, we'll see what the data definition would look like.
- We haven't written the template yet; we'll get to that soon.

# What if Racket didn't have cons?

We could still write a data definition:

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

A Topping is a String.

A Pizza is either

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

Interp:

```
(make-plain-pizza) represents a pizza with no toppings  
(make-topped-pizza t p) represents a pizza like p,  
                        but with topping t added on top.
```

This representation, using a set of alternatives each of which is a struct, is a standard strategy, sometimes called the "sum of products" representation. HINT: You won't go wrong if you use this as your default representation for data in Racket.

This data definition is *self-referential*

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

A Topping is a String.

A **Pizza** is either

```
-- (make-plain-pizza)
```

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

This data definition  
is self-referential,  
just like  
ListOfToppings was.

compare:

A ListOfToppings is either

```
-- empty
```

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

# Examples

Here are some examples of pizzas according to our new data definition.

`(make-plain-pizza)`

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

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

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

A Pizza is either  
-- `(make-plain-pizza)`  
-- `(make-topped-pizza Topping Pizza)`

Can you see why each of these is a Pizza, according to our new definition?

# Template for pizza functions

`pizza-fn : Pizza -> ??`

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

```
  (cond
```

```
    [(plain-pizza? p) ...]
```

```
    [else (... (topped-pizza-topping p)
```

```
              (pizza-fn
```

```
                (topped-pizza-base p)))]))
```

This template is *self-referential*

```
pizza-fn : Pizza -> ??  
(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)

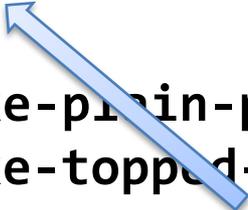


Interp:

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

A Pizza is either

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



Interp:

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

Observe that both data definitions are self-referential in the same way.  
You could represent pizzas either by lists or structures.

# 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))))]))
```



And here are the templates. Observe that they are also both self-referential in the same way.

# 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

# Summary

- You should now be able to
  - convert a data definition using the **ListOfX** pattern to a recursive data definition using structures
  - write a template for a recursive data definition using structures

# Next Steps

- Study the file 06-1-recursive-structures.rkt in the Examples folder.
- If you have questions about this lesson, ask them on the Discussion Board
- Do Guided Practice 6.1
- Go on to the next lesson