COMP 210, FALL 2000
Lecture 10: Lists with Mixed Data

Reminders:

- Homework assignment 1 due today, assignment 2 due next Friday
- Exam will be 2/16/2000, in class–closed-notes, closed-book
- For the score keepers, we've caught up with the fall 99 syllabus

The First Exam

The first exam will cover Sections 1-11, excluding Intermezzo I (Section 8). You are responsible for the contents of the regular lectures, the lab lectures, and the homework problems on the first two homeworks—-numbers zero and one.

Review

1. Did a lot of review.
2. Built a program that traversed a list and returned a new list (by using **cons** as program's final act)

Working with Mixed Data (You will see more of this in lab today & tomorrow)
Lists can contain more than one kind of data. Consider a simple example – a list containing both symbols and numbers. The data definition looks like:

```
;; a list-of-sym-and-num is one of
;;        – empty, or
;;        – (cons S losn)
;;          where S is a symbol and losn is a list-of-sym-and-num, or
;;        – (cons N losn)
;;          where N is a number and losn is a list-of-sym-and-num
```

We haven't written the **define-struct** because we know we can use **cons**, **first**, and **rest** to implement it. (We will use the built-in functions to do this one.)

We can use such a list to represent a recipe. Each symbol names an ingredient. A number indicates how long to cook before adding the next ingredient. (This is not a general representation that accommodates all recipes. For example, it implicitly assumes that all the ingredients get mixed together and cooked. It is, however, good enough for some dishes, such as my spaghetti sauce.

```
;; Example data
 (cons 'garlic (cons 'cumin (cons 5 (cons 'beans (cons 10 empty ) ) ) ) )
```

What does the template for a program over list-of-sym-and-num look like?

```
(define (… a-losn …)
   (cond
         [(empty? a-losn)                    … ]
         [(symbol? (first a-losn))    … (first a-losn) … (rest a-losn) …]
         [(number? (first a-losn))    … (first a-losn) … (rest a-losn) …]
   ) )
```

If we want to write a program, such as **cooking-time**, we can use the template.

```
;; cooking-time: list-of-sym-and-num -> number
;; Purpose: sum up all the numbers in the list to determine total cooking time
(define (cooking-time a-losn)
  (cond
        [(empty?  a-losn)              0 ]
        [(symbol? (first a-losn))      (cook-time (rest a-losn)) ]
        [(number? (first a-losn))      (+ (first a-losn)  (cook-time (rest a-losn))) ]
   ) )

;; ingredient-count: list-of-sym-and-num -> number
;; Purpose: count the number of symbols in the list
(define (ingredient-count a-losn)
   (cond
        [(empty?  a-losn)              0 ]
        [(symbol? (first a-losn))      (+1 (ingredient-count (rest a-losn)) ]
        [(number? (first a-losn))      (ingredient-count (rest a-losn)) ]
   ) )

;; no-cook-recipe? : list-of-sym-and-num -> boolean
;; Purpose: return true if there are no numbers in the list
(define (no-cook-recipe?  a-losn)
   (cond
        [(empty?  a-losn)              true ]
        [(symbol? (first a-losn))      (no-cook-recipe (rest a-losn)) ]
        [(number? (first a-losn))      false ]
   ) )
```

Notice how similar these programs are. The structure of the data determines a major portion of the program's text. By using the design methodology, a large portion of the program is pre-determined–-it almost writes itself.

```
;; get-ingredients : list-of-sym-and-num -> list-of-symbols
;; Purpose: extract the ingredients from a recipe in our list format
(define (get-ingredients a-losn)
   (cond
        [(empty?  a-losn)              empty ]
        [(symbol? (first a-losn))      (cons (first a-losn) (get-ingredients (rest a-losn))) ]
        [(number? (first a-losn))      (get-ingredients (rest a-losn)) ]
   ) )
```

An Issue of Taste
Programming is part science (the COMP 210 part) and part art (the part built on taste, experience, and all those other "soft" terms). Consider our list-of-sym-and-num.
We wrote it as

```
;; a list-of-sym-and-num is one of
;;         – empty, or
;;         – (cons S losn)
;;           where S is a symbol and losn is a list-of-sym-and-num, or
;;         – (cons N losn)
;;           where N is a number and losn is a list-of-sym-and-num
```

We could also have written it as

```
;; a NumSym is either
;;         - a number, or
;;         - a symbol

;; a list-of-NumSyms is either
;;         - empty, or
;;         - (cons f r)
;; where f is a NumSym and r is a list-of-NumSyms
```

Another example:

```
;; a list-of-toppings is one of
;;         - empty, or
;;         - (cons 'cheese a-lot), where a-lot is a list-of-toppings, or
;;         - (cons 'pepperoni a-lot), where a-lot is a list-of-toppings, or
;;         - (cons 'spinach a-lot), where a-lot is a list-of-toppings.
```

Versus

```
;; a topping is one of
;;         – 'cheese, or
;;         – 'pepperoni, or
;;         – 'spinach

;; a list-of-toppings is one of
;;         – empty, or
;;         – (cons f r)
;;  where f is a topping and r is a list-of-toppings
```

Which of these data definitions is preferred? This is a matter of taste, experience–-in short, what Knuth called "The Art of Computer Programming." As you write more programs, larger programs, programs that are used by other people, and, finally, programs that are modified by other people, you will develop insight into this issue. [In fact, programmers with good taste can disagree over such fundamental issues.]

For COMP 210, here is a general rule to follow:

> If there is a meaningful relationship between the two cases, create a new kind of data to represent them–-for example, with pizza toppings and a list of pizza toppings. If there is no inherent relationship, as with numbers and symbols, make the alternatives be explicit cases in the list. NumSym is artificial; pizza toppings is not.

Design Methodology
Let's review the design methodology for programs that use lists (and other recurseive data definitions). We'll use **Bubba-serve?** as an example, and finish writing the function.

1. Data analysis – determine how many pieces of data describe interesting aspects of a typical object mentioned in the problem statement; add a data definitions for each kind ("class") of object in the problem

   For **Bubba-serve?** we need a structure that can hold zero or more names

   ```
   ;; a list-of-symbols is either
   ;;         – empty, or
   ;;         – (cons  first  rest)
   ;;  where first is a symbol and rest is a list-of-symbols
   ;;
   ;; Using the built-in list construct, we don't need the define-struct


   ;; examples
   empty
   (cons  'Bess  (cons  'Mike  (cons  'Susan  (cons  'Bubba   empty) ) ) )
   ```

2. Contract, purpose, header

   ```
   ;; Bubba-serve? : list-of-symbols -> boolean
   ;; Purpose:  determine whether  'Bubba is on the "mechanic" list
   ;; (define (Bubba-serve? a-los) …)
   ```

3. Test Cases

   ```
   ;; (Bubba-serve?  empty) = false
   ;; (Bubba-serve?  (cons  'Bess
                                 (cons  'Mike
                                        (cons  'Susan
                                               (cons  'Bubba   empty) ) ) )  ) = true
   ;; (Bubba-serve? (cons  'Fred (cons  'Jane  (cons  'Felix  empty) ) ) ) = false
   ```

4. Template – for any parameter that is a compound object, write down the selector expressions (access functions?). Template is problem-independent outline for the code body.

   ```
   ;; (define ( … a-los …)
   ;;     (cond
   ;;        [(empty? a-los) … ]
   ;;        [(list?     a-los)          …  (first a-los)
   ;;                                    …  (rest  a-los) … ]
   ```

5. Write the body (using the template)

```
;; Bubba-served? :  list-of-symbol -> bool
;; Purpose: return true if Bubba is in the list
(define (Bubba-served? a-los)
   (cond
     [(empty? a-los)   false ]
     [(list?      a-los)
         (cond
            [(symbol=? (first a-los) 'Bubba)  true ]
            [ else (Bubba-served? (rest  a-los))    ] )
       ]))
```

As you write the body, consider each clause in the **cond** separately. You don't need to think about the **list?** clause when your are writing the **empty?** clause.

6. Test the program (using the examples from step 3)