COMP 210, FALL 2000
Lecture 9: Lists with Mixed Data

Reminders:
- Homework due Wednesday
- Exam will be ***Wednesday, 9/27/2000 in McMurty Auditorium, Room 1055 Duncan Hall***. The exam will be a closed-notes, closed-book, fifty minute exam.

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 three homework assignments.

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-nums-and-syms is one of
;;       – empty, or
;;       – (cons S lons)
;;          where S is a symbol and lons is a list-of-nums-and-syms, or
;;       – (cons N lons)
;;          where N is a number and lons is a list-of-nums-and-syms
;; [We will use Scheme's built in list constructs for list-of-nums-and-syms]
```

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-nums-and-syms look like?

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

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

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

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

```
;; no-cook-recipe? : list-of-nums-and-syms -> boolean
;; Purpose: return true if there are no numbers in the list
(define (no-cook?  a-lons)
   (cond
       [(empty?  a-lons)                 true ]
       [(symbol? (first a-lons))         (no-cook? (rest a-lons)) ]
       [(number? (first a-lons))
          (cond
              {(= 0 (first a-lons))      false}
              {else                      (no-cook? (rest a-lons))}
              )]
    ) )
```

As several people pointed out, we can simplify this further by replacing the inner **cond** with a simple boolean expression

```
(and (= 0 (first a-lons)) (no-cook? (rest a-lons)) )
```

This results in the following version.

```
;; no-cook-recipe? : list-of-nums-and-syms -> boolean
;; Purpose: return true if there are no numbers in the list
(define (no-cook?  a-lons)
   (cond
       [(empty?  a-lons)                 true ]
       [(symbol? (first a-lons))         (no-cook? (rest a-lons)) ]
       [(number? (first a-lons))
                (and (= 0 (first a-lons)) (no-cook? (rest a-lons)) )]
   ) )
```

Some members of class preferred this version. Others asked whether or not DrScheme evaluated the recursive call on **no-cook?** when the expression

```
(= 0 (first a-lons))
```

evaluates to **false**. In fact it does not, but the reasoning is somewhat subtle. In effect, the rewriting engine replaces the **and** construct with the **cond** that we wrote originally. That is, it rewrites

```
(and (= 0 (first a-lons)) (no-cook? (rest a-lons)) )
```

with

```
(cond  {(= 0 (first a-lons))      false}
       {else                      (no-cook? (rest a-lons))} )
```

This expression evaluates the first clause. If its condition is **true**, the expression evaluates to **false**. Otherwise, it goes on to evaluate the second condition.

[Yes, if you wanted to always evaluate both sides, you would have to do something more complex than use an **and**. It does not work to simply reverse the order of the arguments to **and** because the recursive call to **no-cook?** might return **true**, which would terminate evaluation of the **cond**.]

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.

With the same template, we can write a program that returns a list (as at the end of last class).

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

More on Wednesday.