COMP 210, FALL 2000
Lecture 7: Lists with Mixed Data

Reminders:

- Homework due Wednesday
- Exam will be ***Friday, 2/23/2001 in class***. The exam will be a closed-notes, closed-book, fifty minute exam.

Review

- Drill on natural recursion on lists.

Working with Mixed Data (You will see more of this form of data in lab next week.) 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-num-and-sym is one of
;;    - empty, or
;;    - (cons S lons)
;;       where S is a symbol and lons is a list-of-num-and-sym, or
;;    - (cons N lons)
;;       where N is a number and lons is a list-of-num-and-sym
;; [We will use Scheme list structures for list-of-num-and-sym]
```

We did not write any **`define-struct`** statements because we know we can use the built-in Scheme list structures to implement mixed lists.

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 a simple spaghetti sauce.)

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

Question: is this a good choice of data representation for simple recipes? It is debatable. An alternate representation consisting of a list of pairs where each pair is a list of ingredient names and a cooking time might be better because it explicitly groups ingredients that are added at the same time. But such a list representation is homogeneous (not mixed) so it would not illustrate mixed lists.

On the other hand, the alternate representation suggests that ingredients are always cooked; a cooking time of 0 presumably means no cooking is necessary, but it is artificial. What if two consecutive pairs have cooking times of 0? Shouldn't the ingredient lists in those pairs have been merged into a single ingredient list? Does this form of redundancy does not occur in the mixed list representation? It depends. If we allowing cooking times of 0 it does! But we could stipulate that all cooking times must be greater than 0. In the alternate representation, we do not have this option!

Question Are there any other constraints that we should impose on the definition of **`list-of-num-and-sym`**?

For the moment, we will not constrain the cooking times in our mixed list representation to be positive, but we will constrain them to be non-negative! What does this imply about our data definition? We need to qualify the form of numbers in **`list-of-num-and-sym.`**

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

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

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

```
;; cooking-time: list-of-num-and-sym -> 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-num-and-sym -> 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-num-and-sym -> boolean
```

```
;; Purpose: return true if there are no non-zero 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)) (no-cook? (rest a-lons))]
           [else false])))
```

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-num-and-sym -> 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)))]))
```

The semantics of this program are deceptively subtle. Does Scheme evaluate the recursive call on **no-cook?** when the expression

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

evaluates to **false**? It does not; otherwise, the program would be wrong! In effect, the rewriting engine replaces the **and** expression with the **cond** expression that we wrote originally! That is, it rewrites

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

as

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

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

Question: how could we express an “and” operation that always evaluates both of its arguments?

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

Question: how would the definition of **`no-cook?`** change if we stipulate in the definition of **`list-of-num-and-sym`** that cooking-times are always positive?

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

```
;; get-ingredients : list-of-num-and-sym -> 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))]))
```