

COMP 210, Spring 2002

Lecture 9: Lists of Mixed Type & Introduction to Family Trees

Reminders:

- Exam will be 2/13/2002 in class
- Review session will be 7:30pm Monday 2/11/2002

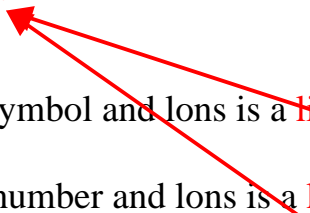
.

An Issue of Taste

In lab, you talked about lists of symbols and numbers. I won't belabor the details on how to declare, document, and use these mixed-element lists. However, there are some matters of taste that we should work through.

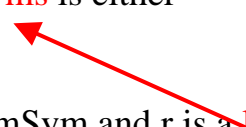
Programming is part science (the COMP 210 part) and part art (the part built on taste, experience, and all those other "soft" terms). Consider our list-of-nums-and-syms.

```
;; 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 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
```



The choice between these two representations depends on the underlying problem. If the numbers and symbols are just grouped together for convenience, then the mixed-list (list-of-nums-and-syms) is probably preferred. If the numbers and symbols are variations on some particular object, then defining a NumSym and having a list of NumSyms probably makes more sense. Some other examples may help convey a sense of this tradeoff ...

Another example

Back to pizza, with an emphasis on supply-side pizzanomics. We might represent a pizza as a list of toppings.

```
:: 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.
;;
;; we will use the built-in list constructor
```

Here, we can easily see how to add new toppings. If we were developing a program that took a list-of-toppings and produced a price (for example), this structure might make sense because it leads to a template that breaks each distinct topping out with its own clause in a **cond** expression.

The alternative is to define a new kind of information, a topping, and use a list of these toppings.

```
:: 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; toppings is not.

One Final Example

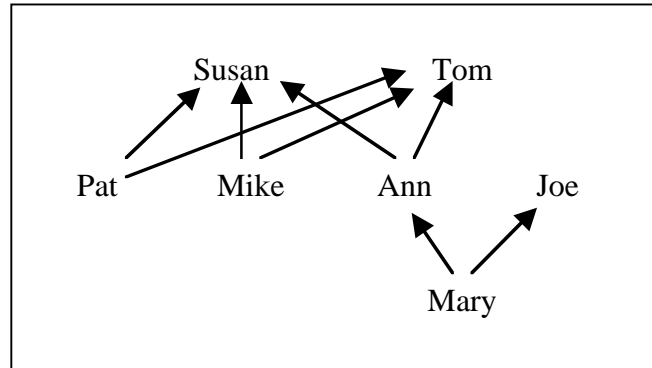
Consider building a personnel system for Rice. It might contain data items for faculty, for staff, for graduate students, and for undergraduates. In each category, the set of information that might be needed will be different. For example:

- Undergraduates have a college, a matriculation date, and a graduation date.
- Graduate students have a department, a degree program (MA, MS, MBA, Ph.D.) and an undergraduate school.
- Staff have a starting date, an evaluator's name, a date for the end of their probationary period, and an office.
- Faculty have a department, a rank (like Assistant, Associate, or Full), a school, and a tenure date.

The list of all personnel would include all four categories--faculty, staff, graduate student, and undergraduate. This would argue for a list with four kinds of structure plus empty, rather than defining a "person" as one of the four categories and keeping a list-of-person.

Working with Mixed Data

By now you should be comfortable working with lists and with recursion. This gives us the foundation we need to start designing programs that operate over more complex data structures. Today, we'll start by working with family trees.



This family tree depicts three generations of a family. Arrows run from child to parent, so Mary's parents are Ann and Joe, Ann's parents are Susan and Tom, and Pat and Mike are Ann's siblings.

How might we write a data definition that allows us to represent these family trees in Scheme? (Recall that we used a list to represent recipes.) This is where I think Computer Science gets fun—devising new and effective ways to represent complex kinds of information.

```

;; a ftn (for family-tree node) is either
;;   - a symbol, or
;;   - (make-ftn name father mother)
;; where name is a symbol and father & mother are both ftns
(define-struct ftn (name mother father))

;; Examples
'Mary
(make-ftn 'Ann 'Susan 'Tom)
(make-ftn 'Mary (make-ftn 'Ann 'Susan 'Tom) 'Joe)
(make-ftn 'Pat 'Susan 'Tom)
(make-ftn 'Mike 'Susan 'Tom)

```

Designing Programs for FTNs

What would the template for this **ftn** contain?

```

(define (f ... a-ftn ...)
  (cond
    [(symbol=? a-ftn) ... ]
    [(ftn?      a-ftn) ...
     (ftn-name a-ftn) ...
     (f (ftn-mother a-ftn)) ...
     (f (ftn-father a-ftn)) ... ]
  ))

```

Let's write a program **in-family?** that consumes an **ftn** and a **symbol** and produces a boolean that indicates whether or not a person with that name is in the family tree.

```

;; in-family?: ftn symbol → boolean
;; Purpose: determine if the symbol is in the ftn
;;          return true if found and false otherwise
(define (in-family? a-ftn kin) ...)

```

Next, we can copy the template over and fill it in.

```
(define (in-family? a-ftn name)
  (cond
    [(symbol=? a-ftn) (symbol=? a-ftn name)]
    [(ftn? a-ftn)
     (or
      (symbol=? (ftn-name a-ftn) name)
      (in-family? (ftn-mother a-ftn) name)
      (in-family? (ftn-father a-ftn) name)
     )]
  )])
```

We can use **or** to check all three possibilities in a single function call, producing the boolean **or** of the answers.

More in the next lecture.