

Lists, Recursive Types, and Functions that Process them





Today's Goals

- Constructing and processing lists
 - **empty, cons**
 - **first, rest**
- Recursive type definitions
- Template for processing recursive types
- Remark on Scheme's built-in lists



Problem Analysis

- Problem descriptions often involve collections of data:
 - “...DNA is a sequence of nucleotides...”
 - “...a play consists of several acts...”
 - “...The index is a list of phrases and page numbers...”
- Lists are the simplest form of collections
- How do we develop programs with lists?



Constructing Lists

- Examples (each line is one list value)
 - **empty** // empty list. Let's us return a correct value with zero elements
 - **(cons 'BlackJack empty)**
 - **(cons 'Baccarat (cons 'BlackJack empty))**
 - **(cons 1 (cons 2 (cons 3 (cons 4 empty))))**
 - **(cons 'Rick (cons 3 (cons true empty)))**
- Important things to remember
 - **empty** carries no elements (meaning zero, *not* one element)
 - **cons** carries exactly one element
 - **cons** takes two arguments
 - The first is an element (the first element) in the list
 - The second is the rest of the list (“another” list)
 - **cons** is a checked constructor
 - It always checks that the second argument is itself a list



Processing Lists

- Examples:
 - `(first (cons 1 (cons 2 empty)))`
= 1
 - `(rest (cons 1 (cons 2 empty)))`
= `(cons 2 empty)`
 - `(first (rest (cons 1 (cons 2 empty))))`
= `(first (cons 2 empty))`
= 2
 - `(first empty)`
= *first: Can't take first of empty list*



Recursive Type Definitions

- How can we define a type for lists?

```
;; A list-of-symbols is either  
;;     empty  
;;     (make-my-cons s los)  
;; where s is a symbol  
;; and los is a list-of-symbols  
(define-struct my-cons (element tail))
```
- What is **first**? **rest**? **cons**?



Recursive Type Definitions

- How can we define a type for lists?

```
;; A list-of-symbols is either  
;;     empty  
;;     (make-my-cons s los)  
;; where s is a symbol  
;; and los is a list-of-symbols  
(define-struct my-cons (element tail))
```
- Note: Here we've made our own list type



Self-reference

- We already saw it in BNF
- Works the same way in type definitions
- We've seen it before in math

$$x = x / 2$$

$$y = (2-y)$$

$$f(z) = d f(z) / d z$$

- Requires care, but is very powerful



Template for Recursive Types

```
;; f : list-of-symbols -> ...  
;; (define (f x)  
;;   (cond  
;;     [(empty? x) ...]  
;;     [else ... (my-cons-element x) ...  
;;               ... (f (my-cons-tail x)) ...]))
```

- Recursive data types implies recursive functions



Template for Recursive Types

```
;; f : list-of-symbols -> ...
;; (define (f x)
;;   (cond
;;     [(empty? x) ...]
;;     [else ... (my-cons-element x) ...
;;               ... (f (my-cons-tail x)) ...]))
```

- Recursion allows us to process rest of list
- Place of recursive call essential for termination
- Non-termination (infinite loop) is hard to debug



Problem: Count elements in list

- Purpose: Compute the number of elements in a list
- Examples
 - `(count empty)`
 - returns 0
 - `(count (make-my-cons 'Lime empty))`
 - returns 1
 - `(count (make-my-cons 'Coconut
 (make-my-cons 'Lime empty)))`
 - returns 2
- How do we go about solving such a problem?



Problem: Count elements in list

- Write the contract, and follow template

```
;; count : list-of-symbols -> number
(define (count x)
  (cond
    [(empty? x) ...]
    [else ... (my-cons-element x) ...
              ... (count (my-cons-tail x)) ...]))
```



Problem: Count elements in list

- First case is easy (and often is)

```
;; count : list-of-symbols -> number
(define (count x)
  (cond
    [(empty? x) 0]
    [else ... (my-cons-element x) ...
              ... (count (my-cons-tail x)) ...]))
```



Problem: Count elements in list

- First case is easy (and often is)

```
;; count : list-of-symbols -> number
(define (count x)
  (cond
    [(empty? x) 0 ]
    [else (+ 1
             (count (my-cons-tail x)))]))
```



Scheme's Built-in Lists

- We defined our own list type from scratch to introduce recursive types from scratch
- Unless instructed otherwise, use Scheme's
- Built-in **cons** is a checked make-my-cons
 - Makes sure second argument is a list
- **first** is just my-cons-element
- **rest** is just my-cons-tail



For Next Class

- Homework due Monday
- Reading:
 - Chapter 10 and companion notes
 - Taking one list and creating another
- Quiz:
 - Chapter 10