CS 135 Fall 2015

Tutorial 03: Lists and Recursion

Goals of this tutorial

You should be able to...

- understand and write the data definitions for lists
- understand and use the template for processing lists to write recursive functions consuming this type of data.
- do step-by-step traces on list functions.

Review: List data definition

```
;; A (listof Any) is one of:
;; ⋆ empty
;; ⋆ (cons Any (listof Any))
```

From the data definition, a list of values of any type is either empty or it consists of a **first** value followed by a list of values (the **rest** of the list).

This is a **recursive** definition. It contains a **base** case, and a **recursive** (self-referential) case.

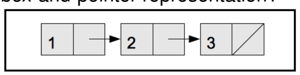
Recursive types should be operated with recursive functions.

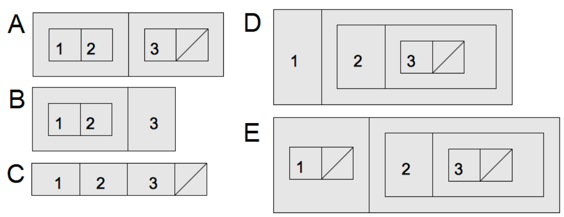
Review: Basic list constructs

- empty: A value representing a list with 0 items.
- cons: Consumes an item and a list and produces a new, longer list.
- first: Consumes a nonempty list and produces the first item.
- rest: Consumes a nonempty list and produces the same list without the first item.
- empty?: Consumes a value and produces true if it is empty and false otherwise.
- cons?: Consumes a value and produces true if it is a cons value and false otherwise.

Clicker Question - box-and-pointer

Which of the following nested box representations match this box-and-pointer representation?





Template for processing a list of symbols

Here we use list of symbols as an example to show a general method for processing a list.

```
;; my-los-fn: (listof Sym) → Any
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [(cons? los) . . . ]))
```

Since cons is a recursive structure type, we can use its selectors in the structure template to get the contents.

The second conditional question can now be replaced by else.

```
;; my-los-fn: (listof Sym) → Any
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else (. . . (first los) . . . (rest los) . . . )]))
```

Now we have the first item and the rest of the list.

Since (rest los) is a list of symbols, we should apply the same computation to it – that is, we apply my-los-fn onto the rest of the symbols until we have nothing left.

The resulting template is a recursive function that consumes a list, and applies the necessary steps to work towards the base case of empty:

```
;; my-los-fn: (listof Sym) → Any
(define (my-los-fn los)
  (cond [(empty? los) . . . ]
        [else (. . . (first los) . . .
                (my-los-fn (rest los)) . . . )]))
```

We can now fill in the dots for a specific example.

Group Problem - list of Strings - Data Definition

Write a data definition for a list of Strings. You can find a similar example on module 5, slide 15 (slide 3 of this tutorial as well). Keep this solution for the next problem.

Group Problem - list of Strings - Template

Based on the previous data definition, write an appropriate template.

Module 5, slides 16-18 also show an example of a template (slide 6-8 of this tutorial as well). Keep this for the next problem.

Group Problem - strings-equal?

Based on the previous template, write a predicate strings-equal? that consumes a list of strings and produces true if all of the strings are equal, otherwise false. Include the contract, one extra example, and one test.

Recall that the function string=? consumes two or more Str and produces a Bool. An example would be (string=? "apple" "apple") yields true.

```
;; (strings-equal? los) checks if every Str in los is equal.
;; Examples:
(check-expect (strings-equal? empty) true)
(check-expect (strings-equal? (cons "one" empty)) true)
```


Review: Defining Structures

The special form

(define-struct sname (fname1 . . . fnamen))

defines the structure type sname and automatically defines the following primitive functions:

- **Constructor:** make-sname
- **Selectors:** sname-fname1 . . . sname-fnamen
- **Predicate:** sname?

The sname? predicate tests if its argument is a sname.

Sname may be used in contracts if the respective data definition has been stated.

Review: Difference between Structures and Lists

When you have a fixed amount of data and you want to group data together, you may use a structure to represent it.

For example, suppose we want to represent information associated with downloaded MP3 files which contains: performer, title, length, genre(rap, country, etc.), we can define a structure as follows.

```
(define-struct mp3info (performer title length genre))
;; An Mp3Info is a (make-mp3info Str Str Num Sym)
```


Review: Difference between Structures and Lists

When the amount of data is unbounded, meaning it may grow or shrink – and you don't know how much, so you cannot use a structure for that kind of data.

For example, suppose you enjoy attending concerts of local musicians and want a list of the upcoming concerts you plan to attend. The number will change as time passes. We will also be concerned about order. So we may use a (listof Str) to represent the concerts data.

Group Problem - rich-accounts

Consider the following definition of a bank account structure account,

```
(define-struct account (owner balance))
;; An Account is a (make-account Str Num)
```

Write a function rich-accounts that consumes a list of Accounts and a Number, and produces a list of Owners whose Accounts have a Balance greater than or equal to the inputted Number. The purpose, contract, and examples are provided.

Base Case

- What is the simplest input for my function?
  - For a list this is typically the empty list.
- What do I do when I reach my base case?
  - Think about what your function produces. If you're producing a list, your base case will often produce the empty list.
- Questions sometimes specify how to deal with the base case.

Recursive Call

- Some argument will change.
- Your recursive call should bring you closer to your base case.
- With lists this means looking at the rest of your list.
- If you have multiple recursive calls, they may be different.

Dealing with the First Element(s)

- We know the list isn't empty.
- You may want to check some property of (first List).
- Your function may produce something using the (first List).
- Will combine with your recursive call.
- If you're dealing with structures, think about the template of that structure.

Group Problem - condensed trace

Recall our definition of strings-equal?. We will perform a condensed trace of:

```
(strings-equal?
   (cons "iPod"
      (cons "iPod"
         (cons "Playbook" empty))))
```


Group Problem - condensed trace

```
(define (strings-equal? los)
  (cond
    [(empty? los) true]
    [(empty? (rest los)) true]
    [else (and (string=? (first los) (first (rest los)))
               (strings-equal? (rest los)))]))

(strings-equal? (cons "iPod"
                      (cons "iPod"
                            (cons "Playbook" empty))))
```


Group Problem - condensed trace

```
(string-equal? (cons "iPod"
                     (cons "iPod"
                           (cons "Playbook" empty))))
```

Group Problem - condensed trace

```
(string-equal? (cons "iPod"
                     (cons "iPod"
                           (cons "Playbook" empty))))

=> (and (string=? "iPod" "iPod")
        (strings-equal? (rest (cons "iPod" (cons "iPod" (cons "Playbook" empty))))))
```


Group Problem - condensed trace

```
(string-equal? (cons "iPod"
                     (cons "iPod"
                           (cons "Playbook" empty))))

=> (and (string=? "iPod" "iPod")
        (strings-equal? (rest (cons "iPod" (cons "iPod" (cons "Playbook" empty))))))

=> (and true
        (strings-equal? (rest (cons "iPod" (cons "iPod" (cons "Playbook" empty))))))
```


Group Problem - condensed trace

```
(string-equal? (cons "iPod"
                     (cons "iPod"
                           (cons "Playbook" empty))))

=> (and (string=? "iPod" "iPod")
        (strings-equal? (rest (cons "iPod" (cons "iPod" (cons "Playbook" empty))))))

=> (and true
        (strings-equal? (rest (cons "iPod" (cons "iPod" (cons "Playbook" empty))))

=> (and (strings-equal? (cons "iPod" (cons "Playbook" empty))))
```

Group Problem - condensed trace

```
=> (and (strings-equal? (cons "iPod" (cons "Playbook" empty))))
```


Group Problem - condensed trace

```
=> (and (strings-equal? (cons "iPod" (cons "Playbook" empty))))

=> (and (string=? "iPod" "Playbook")
        (strings-equal? (rest (cons "iPod" (cons "Playbook" empty)))))
```


Group Problem - condensed trace

```
=> (and (strings-equal? (cons "iPod" (cons "Playbook" empty))))

=> (and (string=? "iPod" "Playbook")
        (strings-equal? (rest (cons "iPod" (cons "Playbook" empty)))))

=> (and false (strings-equal? (rest (cons "iPod" (cons "Playbook" empty)))))
```

Group Problem - condensed trace

```
=> (and (strings-equal? (cons "iPod" (cons "Playbook" empty))))

=> (and (string=? "iPod" "Playbook")
        (strings-equal? (rest (cons "iPod" (cons "Playbook" empty)))))

=> (and false (strings-equal? (rest (cons "iPod" (cons "Playbook" empty)))))

=> false
```


Group Problem - string-length-adder (Optional)

Based on the template on slide 10, write the contract and function definition for string-length-adder which consumes a list of Strings and produces the length of all the strings added together. If you are given empty, produce 0.

Recall that the function string-length consumes a Str and produces a Nat. An example would be (string-length "apple") yields 5.

```
;; (string-length-adder los) adds up the length of all strings in los, a given (listof Str).
;; Examples:
(check-expect (string-length-adder
               (cons "MA" (cons "MAT" (cons "MATH" (cons "MATH rocks" empty))))) 19)
(check-expect (string-length-adder (cons "Turkey" empty)) 6)
```