Computer Science 115 – SOS Final Exam Review

Tutor: Jumana Bahrainwala (jzbahrai@uwaterloo.ca)
Coordinator: Erin Vanderstoep (evanders@uwaterloo.ca)

STRUCTURES

- A structure bundles data in a package together
- Serves as a useful way to refer to data you define
- Defining a structure is the first step (they usually do this for you)
- Gives three useful functions

Note: The contract should have the name of the structure rather than "structure"

To define a structure:

```
(define-struct sname (field1 field2.....))
```

Doing this gives you three functions:

```
make-sname                          ;constructor
sname-field1, sname-field2....      ;selectors
sname?                              ;type predicate
```

○ Basically this is saying that you have created a structure called "sname" that has properties defined in f1....fn

Structure Template

```
(define (function-name  a-struct)
        ... (structure-f1 a-struct)...
        ... (structure-f2 a-struct)...
                    ...
        ... (structure-fn a-struct)...)
```

A Data Definition: is a comment specifying the types of data

```
;;a posn in a structure (make-posn   xcoord    ycoord), where
;;xcoord is a number
;;ycoord is a number
```

Structure Definitions: show what each item represents

(define-struct contact (name email_address))

;; A contact is a structure *(make-contact n e) where*

;; *n is a non-empty string representing a person's name and*

;; *e is a non-empty string representing a person's email address.*

Computer Science 115 – SOS Final Exam Review

LISTS

Formal Definition

A list is either:

- empty or
- (cons f r), where f is a value and r is a list

Examples of Lists in Scheme:

=> (cons ‘a (cons ‘b (cons ‘c (cons ‘d empty))))
=> empty
=> (list ‘a ‘b ‘c ‘c)

List Template

```
(define (mylist alist)
  (cond
    [(empty? alist)...]
    [(cons? alist) ...]))
```

Difference between Structures and Lists:

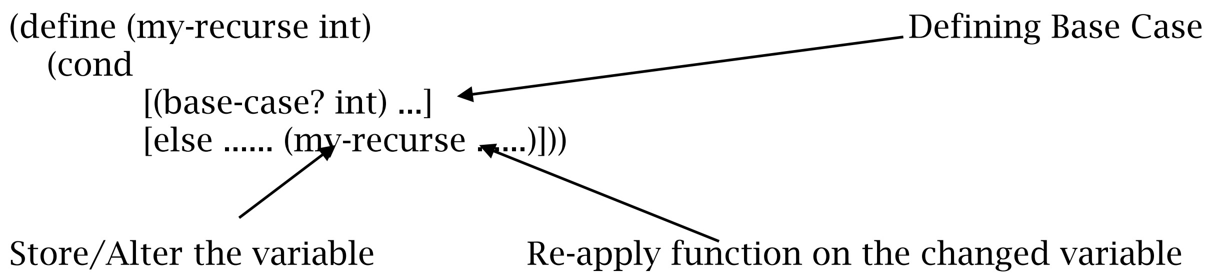
- Structures are easier to select data
- Lists are better for recursion

INTRODUCTION TO RECURSION

Base Case: The starting and ending point of the function. The first thing that the function checks in an “if” case. Function stops recursing when the base case is true.

Recursive Part: Only if your base case fails. Ask “What do you want to change if your base case fails?” Keep doing this until your base case passes.

A general recursive template



```
(define (my-recurse int)
  (cond
          [(base-case? int) ...]
          [else ...... (my-recurse ......)]))
```

Problem 1
Give the structure templates for the following structures. Then, define a function that consumes a list of CD structures and produces the total price, using the template for CD.

```
(define-struct movie (title producer))

(define-struct CD (artist title price))
```

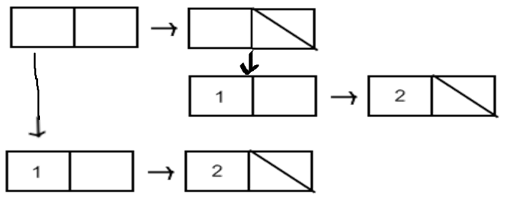
Solution

```
(define (my-movie-fn a-movie)
            ...(movie-title a-movie)...
            ...(movie-producer a-movie)...)

(define (my-CD-fn a-CD)
            ...(CD-artist a-CD)...
            ...(CD-title a-CD)...
            ...(CD-price a-CD)...)

(define (total-price aloCD)
  (cond
            [(empty? aloCD) 0]
            [else (+ (CD-price (first aloCD)) (total-price (rest aloCD)))]))
```

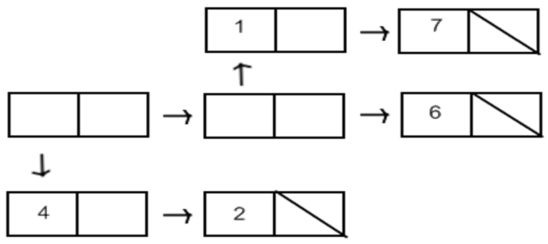
:: why did we choose this particular base case? We want the function to stop once we have added up ALL the prices. So we need to keep going until the list is empty. When it is empty, we aren't adding any price, so just add 0 to our price total.

:: Or you can think about it another way. If we had an empty list in there, there would be no prices to add, and our total price is zero.

LIST OF LISTS



(list (list 1 2) (list (list 1 2) empty)



(list (list 4 2) (list 1 7) 6)

List of List Template

```
(define (my-list-fn a-list)
   (cond ((empty? a-list) ...)
            ((cons? (first a-list)) ...)
            (else ...(first a-list)...(my-list-fn (rest a-list)))))
```

Dictionaries/Association Lists

- Each List contains smaller lists, with each smaller list having two elements
- One represents a key, and other represents the value corresponding to the key
- An al is either empty or (cons (list k v) alst) where k is a number (key), v is a string (value) and alst is a list
- (list (list 5 a) (list 6 jumana) (list 19 computers))

Problem 2

Consume an association list and a key and find the value that corresponds to that key, or return false if the key is not found.

Solution

```
(define (find-val al k)
      (cond
         [(empty? al) false]
         [(= (first (first al)) k) (second (first al))]
         [else (find-val (rest al) k)]))
```

PROCESSING TWO LISTS

Three Methods for Processing Two Lists

1. Have a list "go along for the ride"
2. Processing in Lockstep (only for lists of equal lengths)
3. Processing Lists at Different Rates

Each type of problem is described below:

Lockstep Template

```
(define (my-list-fn lst1 lst2)
   (cond
     [(empty? lst1) ...]
     [else  ... (first lst1) ... (first lst2)...
            ... (my-lst-fn (rest lst1) (rest lst2))]))
```

Different Rates Template

```
(define (my-lst-fn lst1 lst2)
   (cond
     [(and (empty? lst1) (empty? lst2)) ...]
     [(and (cons? lst1) (empty? lst2)) ...]
     [(and (empty? lst1) (cons? lst2)) ...]
     [(and (cons? lst1) (cons? lst2)) ...]))
```

Problem 3

Develop a function, cross, that consumes a list of symbols and a list of numbers and produces all possible pairs of symbols and numbers.

Solution

```
(cross (list 'a 'b 'c) (list 1 2))
        => (list (list 'a 1) (list 'a 2) (list 'b 1) (list 'b 2) (list 'c 1) (list 'c 2))

(define (cross-helper s l)
   (cond
      [(empty? l) empty]
      [else (cons (list s (first l)) (cross-helper s (rest l)))]))

(define (cross alos alon)
   (cond
      [(empty? alos) empty)]
      [else (append (cross-helper (first alos) alon) (cross (rest alos) alon)))]))
```

Problem 4

Develop the function zip, which combines a list of names and a list of phone numbers into a list of phone records. The lists are of the same size. Assume the following structure definition:

```
(define-struct phone-record (name number))
   ;;A phone-record is a structure (make-phone-record s n)
   ;; where s is a string representing a person's name
   ;; n is a number representing their number
```

Solution

```
(define (zip names numbers)
(cond
        [(empty? names) empty]
        [else (cons (make-phone-record (first names) (first numbers))(zip (rest
           names) (rest numbers)))]))
```

Problem 5

The function find-name consumes two lists of strings (l1 and l2) and returns a list of strings which are in l2 as well as l1.
Example: (find-name (list “harry” “hermoine” “ronald”) (list “sunny” “kumar” “harry” “ronald”)) -> (list “harry” “ronald”)

Solution

Helper Function
```
(define (is-in? word low)
  (cond [(empty? low) false]
        [(equal? word (first low)) true]
        [else (is-in? word (rest low))]))
```

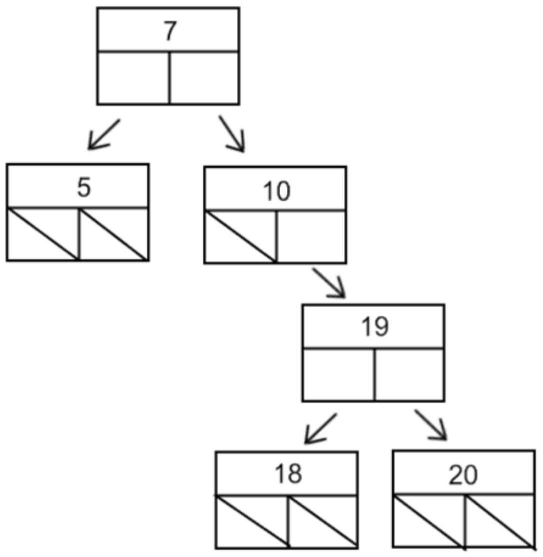
Main Function
```
(define (find-name l1 l2)
  (cond [(and (empty? l1) (empty? l2)) empty]
        [(and (cons? l1) (empty? l2))empty]
        [(and (empty? l1) (cons? l2)) empty]
        [(and (cons? l1) (cons? l2))
         (cond [(is-in? (first l1)l2)(cons (first l1) (find-name (rest l1) l2))]
               [else (find-name (rest l1) l2)])]))
```

BINARY TREE
(define-struct node (key val left right))

- is either EMPTY, OR
- (make-node k v l r)
  - k is a number representing a key (like association lists)
  - v is a value representing a value (like association lists)
  - l is a binary tree
  - r is a binary tree

BINARY SEARCH TREE
(define-struct node (key val left right))

- is either EMPTY, OR
- (make-node k v l r), where
  - k is a number representing a key (like association lists)
  - v is a value representing a value (like association lists)
  - l is a binary tree **with keys less than k**
  - r is a binary tree **with keys greater than k**



Problem 6
Develop the function, contains-bt. The function consumes a binary tree and a number n and determines whether the number occurs in the tree.

```
(define-struct node (key val left right))
```

Solution

```
(define (contains-bt t n)
   (cond
        [(equal? (node-key t) n) true]
        [else (or (contains-bt (node-left t) n)(contains-bt (node-right t) n))]))
```

Problem 7
Develop the function, contains-bt. The function consumes a binary search tree and a number n and determines whether the number occurs in the tree.

```
(define-struct node (key val left right))
```

Solution

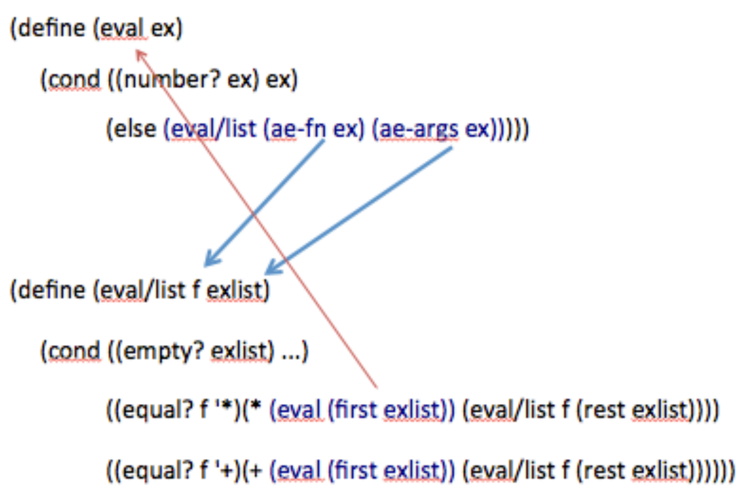
```
(define (contains-bst t n)
   (cond
        [(equal? (node-key t) n) true]
        [(< n (node-key t)) (contains-bst (node-left t) n)]
        [(> n (node-key t)) (contains-bst (node-right t) n]))
```

MUTUAL RECURSION

You are "referring" to another function when you make your recursive calls.

(define-struct ae (fn args))
- A number or
- (make-ae f alist), where f is a symbol, and alist is an aexplist

An aexplist is either
- Empty or
- (cons a alist), where a is an aexp, and alist is an aexplist



```
(define (eval ex)
   (cond ((number? ex) ex)
         (else (eval/list (ae-fn ex) (ae-args ex)))))

(define (eval/list f exlist)
   (cond ((empty? exlist) ...)
         ((equal? f '*)(* (eval (first exlist)) (eval/list f (rest exlist))))
         ((equal? f '+)(+ (eval (first exlist)) (eval/list f (rest exlist))))))
```

If you have two structures that refer to each other in their definitions, it's usually a sign that mutual recursion should be used.

LOCAL
Benefits of local definitions:

- Hide helper functions inside main function
- Improve efficiency
- The exact helper functions you were using before can now be contained in the code

```
(define (swap-parts mystring)
    (local [(define len (string-length mystring))
                (define mid (quotient len 2))
                 (define front (substring mystring 0 mid))
                 (define back (substring mystring mid len))]
    (string-append back front)))
```

FUNCTIONAL ABSTRACTION
<u>The map function</u>

- map takes a list of numbers and a function and applies the function to the list of numbers.

Contract: (X → Y) (list of X) → (list of Y)
Example: (map add1 (list 1 2 3)) → (list 2 3 4)

The filter function

- filter takes a question and a list; for every element in the list that evaluates the question to true, filter returns those elements in a new list.

Contract: (X → boolean) (list of X) → (list of X)
Example: (filter even? (list 1 2 3 4)) → (list 2 4)

The foldr function

- foldr takes a function, its base case, and a list and then reduces the list to a single value using the function.

Contract: (X → X) X (list of X) → X
Example: (foldr max 0 (list 5 6 3 4 3 2)) → 6

Problem 8

Create the function perfect-squares that consumes a list of non negative integers, lon, and produces a list of all those elements from lon that are perfect squares. Make sure you use local.

Solution

```
(define (perfect-squares lon)
   (local
         [define (is-perfect-square? num)(integer? (sqrt num)))]
   (filter is-perfect-square? lon)))
```

Problem 9
Make a function, add-positives that adds all the squares of positive numbers in a list of integers and strings.

Solution

```
(define (add-positives loa)
    (foldr + 0  (map sqr (filter integer? loa))))
```