HIERARCHICAL DATA 5

COMPUTER SCIENCE 61AS

Concepts and Definitions

1. What is a Tree? How is it different from a Deep List? When would you use one over the other?

> **Solution:** A Tree is an Abstract Data Type composed of a datum (some data value) and children (a List of Trees). Thus each Tree may connect to zero or more other Trees. Trees are used to structure data in a hierarchical fashion (compared with Lists, which structure data in a sequential fashion.) A Deep List is a list that may have other lists inside and is also used as a hiarchical data structure. When to use a Tree over a Deep List depends on the application. For example, you can model a filesystem (files and folders) as a Tree–the name of the file or folder would be the datum, and the children would be the list of Trees, each containing the name of another file or folder. Files have no children, while folders have a child for each file it holds. Deep lists are useful when you have a list of lists. For example, if you represented full names as lists, and you had a list of full names, i.e. '((john lennon) (paul mccartney) (george harrison) (ringo starr)).

2. What is mutual recursion?

> **Solution:** Mutual recursion is a way to do recursion on two different types of data that are nested within each other. Usually, we will use mutual recursion on trees. In a tree, we have two different types of "things" – we have the tree itself (and other trees in it), and we have forests, which are lists of trees.

3. When working with trees, what is (typically) the input to each mutually recursive procedure?

> **Solution:** Normally, we'll have one function that deals only with trees and another function that deals only with forests. Since trees contain forests, the procedure that

deals with trees neecs to call the one that deals with forests. Since forests contain trees, the procedure that deals with forests need to call the one that deals with trees. In other words, mutual recursion.

4. What is a binary tree? Which selectors do we use for it?

Solution: A binary tree is the same as a normal tree except it has at most two children. We use `left-branch` and `right-branch` to select the two child trees.

5. What is car/cdr recursion?

Solution: In car/cdr recursion, you treat the `car`s and `cdr`s symmetrically, just like you would treat `left-branch` and `right-branch` symmetrically for binary trees. This allows your procedure to work any structure created by cons, not just lists.

Practice with Capital-T Trees

1. Assume `forest` is a forest (list of Trees) and `tree` is a Tree where each element is a number. For each of the following lines of code, say whether it is a data abstraction violation, a domain/range mismatch, or valid.

   a. `(car forest)`

   Solution: Valid

   b. `(car tree)`

   Solution: DAV

   c. `(children tree)`

   Solution: Valid

   d. `(children forest)`

   Solution: Domain/range mismatch

   e. `(null? (children tree))`

Solution: Valid

f. `(cdr (children tree))`

Solution: Valid

2. Write a procedure `add-child-lengths` which takes as input a tree, and produces as output a new tree in which each datum is now a list whose first element is the original datum, and whose second element is the number of children it has. (You can find the length of a list using the `length` procedure.) Use mutual recursion here.

Solution:

```
(define (add-child-lengths tree)
      (make-tree (list (datum tree) (length (children tree)))
                 (forest-lengths (children tree))))

(define (forest-lengths forest)
    (if (null? forest)
        nil
        (cons (add-child-lengths (car forest))
              (forest-lengths (cdr forest)))))
```

3. Write `sum-tree`, which takes as input a tree of numbers and produces as output the sum of all the numbers in the tree. You should use sequence operations (`map, filter, accumulate`) to implement it. No helper procedures allowed! (This is a hard question to start with, so you may want to first do it with mutual recursion, which is easier.)

Solution:

```
Using sequence operations:

(define (sum-tree tree)
    (+ (datum tree)
       (accumulate + 0 (map sum-tree (children tree)))))

Using mutual recursion:

(define (sum-tree tree)
    (+ (datum tree)
         (sum-forest (children tree))))

(define (sum-forest forest)
    (if (null? forest)
        0
        (+ (sum-tree (car forest))
           (sum-forest (cdr forest)))))
```

4. `listify-tree` takes as input a tree and produces as output a flat (not deep) list of all of the datums in the tree, in any order.

   a. Write `listify-tree` using higher order functions.

Solution:

```
(define (listify-tree tree)
    (cons (datum tree)
          (accumulate append nil (map listify-tree (children tree)))))
```

   b. Write `listify-tree` using mutual recursion.

Solution:

```
(define (listify-tree tree)
    (cons (datum tree) (listify-forest (children tree))))

(define (listify-forest forest)
    (if (null? forest)
        nil
        (append (listify-tree (car forest))
                (listify-forest (cdr forest)))))
```

5. Write `count-leaves`, which returns the number of leaves in the input tree. A leaf is any tree which has no children.

Solution:

```
Using sequence operations:

(define (count-leaves tree)
    (if (null? (children tree))
        1
        (accumulate + 0 (map count-leaves (children-tree)))))


Using mutual recursion:

(define (count-leaves tree)
    (if (null? (children tree))
        1
        (count-leaves-forest (children tree))))

(define (count-leaves-forest forest)
    (if (null? forest)
        0
        (+ (count-leaves (car forest))
           (count-leaves-forest (cdr forest)))))
```

Practice with HOFs and Deep Lists

1. Louis Reasoner writes a procedure, `deep-squares`, that takes in a deep list and squares every number in it. He writes the following code.

```
(define (deep-squares lol)
    (cond ((null? lol) '())
          ((list? (car lol))
           (cons (map square (car lol))
                 (deep-squares (cdr lol)) ))
          (else (cons (square (car lol))
                      (deep-squares (cdr lol)) ))))
```

a. Say whether the code above will error, work, or is a data abstraction violation.

```
(deep-squares '())
(deep-squares '(1 (2 3) (4 5))
(deep-squares '(1 (2 (3) 4) (((5))))
```

Solution: This code will only work for lists up to a depth of two. This is because, when the `car` of the list is a list, we simply map `square` over the elements, which means we assume the elements are simple numbers (not lists themselves).

b. Now fix Louis's code. Hint: it can be fixed with an extremely small change. Do NOT use deep-map in your solution.

Solution: Replace `(map square (car lol))` with `(deep-squares (car lol))`

2. Express the following function without using recursion. (Hint: use `map`).

```
(define (jelly-words lst)
    (if (null? lst)
        '()
        (cons (word 'jelly (car lst))
              (jelly-words (cdr lst)))))

> (jelly-words '(fish bean donut car cdr) )
(jellyfish jellybean jellydonut jellycar jellycdr)
```

Solution:

```
(define (jelly-words lst)
    (map (lambda (x) (word 'jelly x)) lst))
```

3. Express the following function without using recursion.

```
(define (sqrt-positives nums)
    (cond ((null? nums) '())
          ((<= (car nums) 0) (sqrt-positives (cdr nums)))
          (else (cons (sqrt (car nums))
                      (sqrt-positives (cdr nums))))))

> (sqrt-positive '(1 -9 5 -4))
(1 25)
```

Solution:

```
(define (sqrt-positives nums)
    (map sqrt (filter (lambda (x) (> x 0)) nums)))
```

Practice with Car/cdr Recursion

1. Write the procedure `sum-binary-tree`, which sums the datum of a binary tree.

Solution:

```
(define (sum-binary-tree bt)
    (if (equal? bt the-empty-tree)
        0
        (+ (datum bt)
           (sum-binary-tree (left-branch bt))
           (sum-binary-tree (right-branch bt)))))
```

2. Now, write the procedure `sum-cons-structure`, which should work on any structure created by `cons`. Would `(sum-cons-structure bt)` evaluate to the same thing as `(sum-binary-tree bt)` (assuming that `bt` is a valid input to `sum-binary-tree`)? If yes, why don't we generally do this? If no, why not?

Solution:

```
(define (sum-cons-structure struct)
    (if (pair? struct)
        (+ (sum-cons-structure (car struct))
           (sum-cons-structure (cdr struct)))
        struct))
```

Yes, we could use `sum-cons-structure` in place of `sum-binary-tree`, but only with our current representation of binary trees. If we changed the representation, then it would no longer work, while `sum-binary-tree` would still work. That's the point of data abstraction!