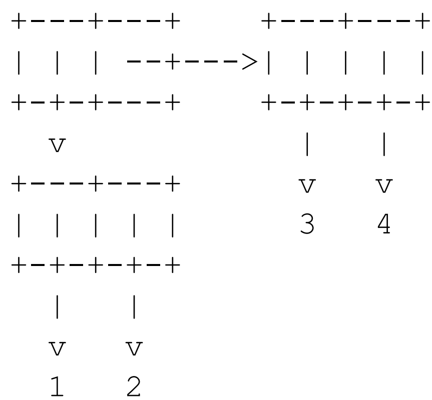
CS61A Notes 02b – Fake Plastic Trees

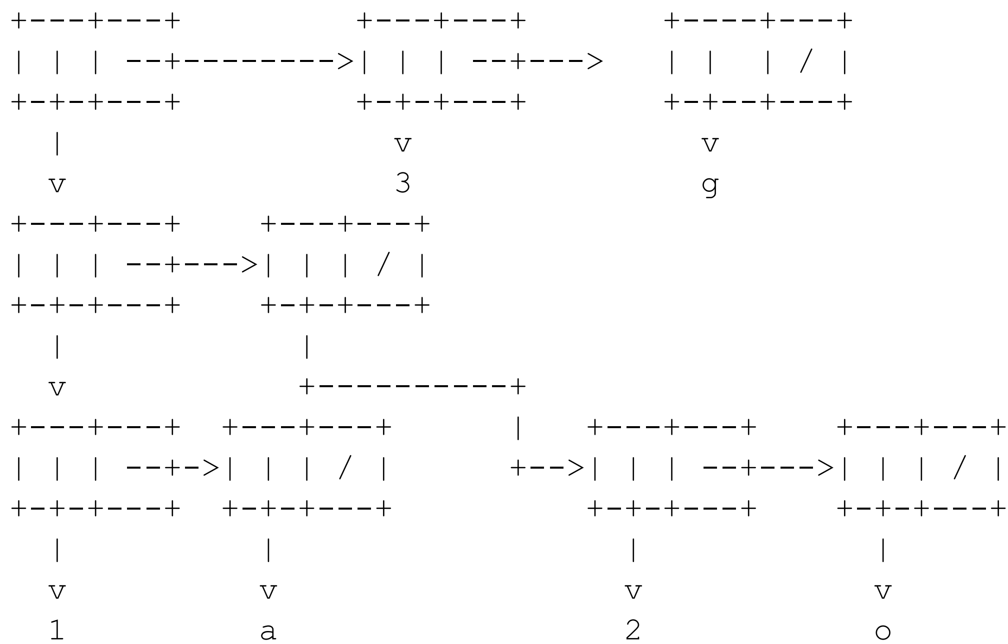
Box and Pointer Diagrams

QUESTIONS: Evaluate the following, and draw a box-and-pointer diagram for each. (Hint: It may be easier to draw the box-and-pointer diagram first.)

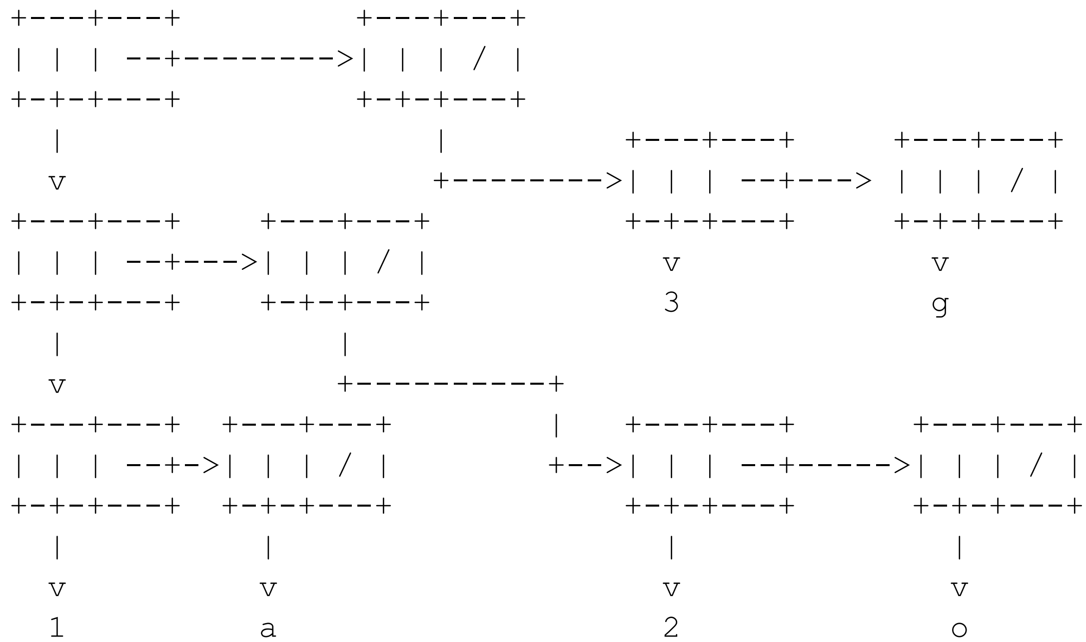
1. `(cons (cons 1 2) (cons 3 4))`



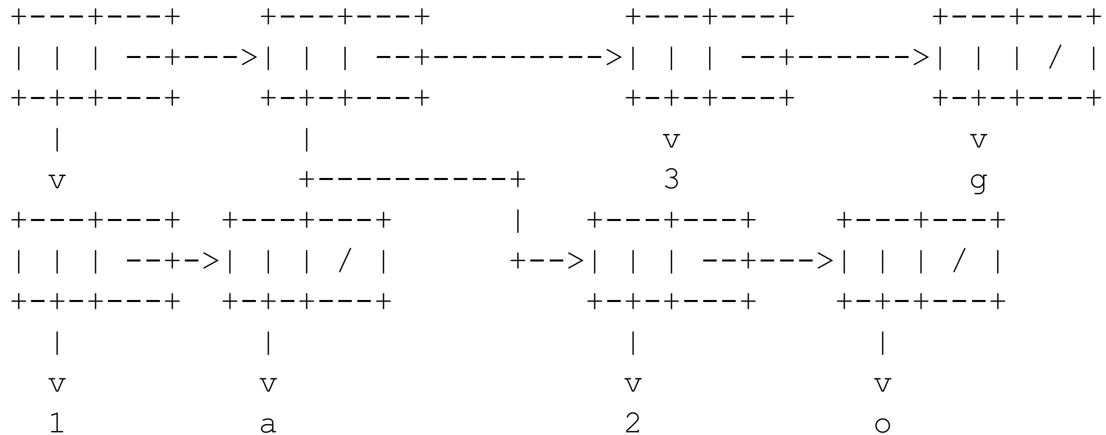
2. `(cons '((1 a) (2 o)) '(3 g))`



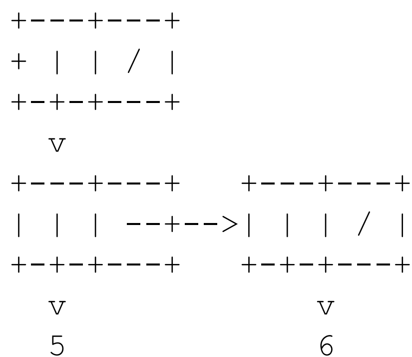
3. `(list '((1 a) (2 o)) '(3 g))`



4. `(append '((1 a) (2 o)) '(3 g))`



5. `(cdr (car (cdr '(((1) 3) (4 (5 6))) )))`



6. `(map (lambda (fn) (cons fn (fn 6))) (list square 1+ even?))`

```
+---+---+           +---+---+             +---+---+
| | | --+----------->| | | --+-------------->| | | / |
+-+-+---+           +-+-+---+             +-+-+---+
  v                   v                     v
+---+---+  +---+-+   +---+---+  +---+-+  +---+---+   +---+-+
| | | --+->| | |/|   | | | --+->| | |/|  | | | --+-->| | |/|
+-+-+---+  +-+-+-+   +-+-+---+  +-+-+-+  +-+-+---+   +-+-+-+
  v          v         v          v        v           v
  square     36        1+         7        even?       #t
```

*** Note: square, 1+ and even? are procedures here

(Slightly) Harder Lists

1. Define a procedure `(depth ls)` that calculates the maximum depth of sublists in `ls`. For example,

```
(depth '(1 2 3 4)) => 1
(depth '(1 2 (3 4) 5)) => 2
(depth '(1 2 (3 4 5 (6 7) 8) 9 (10 11) 12)) => 3
```

Remember that there's a procedure called `max` that takes in two numbers and returns the greater of the two.

An ATOM is anything that is not a pair.
Note that the empty list also falls under this category.

```
(define (depth ls)
  (if (atom? ls)
      0
      (max (1+ (depth (car ls))) (depth (cdr ls)))))
```

2. Define a procedure `(remove item ls)` that takes in a list and returns a new list with `item` removed from `ls`.

If the given list is null, return the empty list.
Otherwise, check if the first item in the list is what we want to remove.
Removing an element is accomplished by returning the "processed" rest of the list; in this case the removed version of the CDR of the list.
If we want to keep the first element instead, cons it onto the

```
removed rest of the list.
(define (remove item ls)
  (cond ((null? ls) nil)
        ((equal? item (car ls))
         (remove item (cdr ls)))
        (else (cons (car ls)
                    (remove item (cdr ls))))))
```

3. Define a procedure `(unique-elements ls)` that takes in a list and returns a new list without duplicates. You've already done this with `remove-dups`, and it used to do this:

   `(remove-dups '(3 5 6 3 3 5 9 8)) ==> (6 3 5 9 8)`
   where the *last* occurrence of an element is kept. We'd like to keep the *first* occurrences:
   `(unique-elements '(3 5 6 3 3 5 9 8)) => (3 5 6 9 8)`
   Try doing it without using `member?`. You might want to use `remove` above.

```
(define (unique-elements ls)
  (if (null? ls)
      '()
      (cons (car ls) (unique-elements
                         (remove (car ls) (cdr ls))))))
```

4. Define a procedure `(count-of item ls)` that returns how many times a given item occurs in a given list; it could also be in a sublist. So,

   `(count-of 'a '(a b c a a (b d a c (a e) a) b (a))) => 7`

```
(define (count-of item ls)
  (cond ((null? ls) 0)
        ((pair? (car ls))
         (+ (count-of item (car ls))
            (count-of item (cdr ls))))
        ((equal? (car ls) item)
         (1+ (count-of item (cdr ls))))
        (else (count-of item (cdr ls)))))
```

5. Define a procedure `(count-unique ls)` which, given a list of elements, returns a list of pairs whose `car` is an element and whose `cdr` is its number of occurrences in the list. For example,

   `(count-unique '(a b b b c d d a e e f a a))`
   `=> ((a . 4) (b . 3) (c . 1) (d . 2) (e . 2) (f . 1))`
   You might want to use `unique-elements` and `count-of` defined above.

```
(define (count-unique ls)
  (map (lambda (x) (cons x (count-of x ls))) (unique-elements
ls)))
```

6. Define a procedure `(interleave ls1 ls2)` that takes in two lists and returns one list with elements from both lists interleaved. So,

```
(interleave '(a b c d) (1 2 3 4 5 6 7)) => (a 1 b 2 c 3 d 4 5 6 7)
```

```
(define (interleave ls1 ls2)
  (cond ((null? ls1) ls2)
        ((null? ls2) ls1)
        (else (cons (car ls1) (interleave ls2 (cdr ls1))))))
```

7. Write a procedure `(apply-procs procs args)` that takes in a list of single-argument procedures and a list of arguments. It then applies each procedure in `procs` to each element in `args` in order. It returns a list of results. For example,

```
(apply-procs (list square double +1) '(1 2 3 4))
=> (3 9 19 33)
```

```
(define (apply-procs procs args)
  (if (null? procs)
      args
      (apply-procs (cdr procs) (map (car procs) args))))
```

Fake Plastic Trees

A tree is, abstractly, an acyclic, connected set of nodes (of course, that's not a very friendly definition). Usually, it is a node that contains two kinds of things – data and children. Data is whatever information may be associated with a tree, and children is a set of subtrees with a node as the parent. Concretely, it is often just a list of lists of lists of lists in Scheme, but it's best NOT to think of trees as lists at all. Trees are trees, lists are lists. They are completely different things, and if you, say, call `(car tree)` or something like that, that violates the data abstraction. `car`, `cdr`, `list` and `append` are for lists, not trees! And don't bother with box-and-pointer diagrams – they get way too complicated for trees. Just let the data abstraction hide the details from you, and trust that the procedures like `make-tree` work as intended.

Of course, that means we need our own procedures for working with trees analogous to `car`, `cdr`, etc. Different representations of trees use different procedures. You have already seen the ones for a general tree, which is one that can have any number of children (not just two) in any order (not grouped into smaller-than and larger-than). Its operators are:

```
;; takes in a datum and a LIST of trees that will be the children of this
```

```
;; tree, and returns a tree.
(define (make-tree label children) ...)

;; returns the datum at this node.
(define (datum tree) ...)

;; returns a LIST of trees that are the children of this tree.
;; NOTE: we call a list of trees a FOREST
(define (children tree) ...)
```

With general trees, you'll often be working with mutual recursion. This is a common structure:

```
(define (foo-tree tree)
     ...
     (foo-forest (children tree)))

(define (foo-forest forest)
     ...
     (foo-tree (car forest))
     ...
     (foo-forest (cdr forest)))
```

Note that `foo-tree` calls `foo-forest`, and `foo-forest` calls `foo-tree`! Mutual recursion is absolutely mind-boggling if you think about it too hard. The key thing to do here is – of course – **TRUST THE RECURSION**! If when you're writing `foo-tree`, you BELIEVE that `foo-forest` is already written, then `foo-tree` should be easy to write. Same thing applies the other way around.

QUESTIONS

1. Write `(square-tree tree)`, which returns the same tree structure, but with every element squared. Don't use "`map`"!

```
(define (square-tree tree)
  (make-tree (* (datum tree) (datum tree))
             (square-forest (children tree))))
(define (square-forest forest)
  (if (null? forest)
      '()
      (cons (square-tree (car forest)) (square-forest (cdr forest)))))
```

2. Write `(max-of-tree tree)` that does the obvious thing. The tree has at least one element.

```
(define (max-of-tree tree)
  (if (null? (children tree))
      (datum tree)
      (max (datum tree) (max-of-forest (children tree)))))
```

```
(define (max-of-forest forest)
  (if (null? (cdr forest))
      (max-of-tree (car forest))
      (max (max-of-tree (car forest)) (max-of-forest (cdr forest)))))
```

3. Write `(listify-tree tree)` that turns the tree into a list in any order. (This one you can't use map even if you tried... Muwahahaha.)

```
(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)))))
```

Binary Search Trees

A binary search tree is a special kind of tree with an interesting restriction – each node only has two children (called the "left subtree" and the "right subtree", and every node in the left subtree has datum smaller than the node of the root, and every node in the right subtree has datum larger than the node of the root. Here are some operators:

```
;; takes a datum, a left subtree and a right subtree and make a bst
(define (make-tree datum left-branch right-branch) ...)

;; returns the datum at this node
(define (datum bst) ...)

;; returns the left-subtree of this bst
(define (left-subtree bst) ...)

;; returns the right-subtree of this bst
(define (right-subtree bst) ...)
```

So then, let's get to it!

QUESTIONS

1. Jimmy the Smartass was told to write `(valid-bst? bst)` that checks whether a tree satisfies the binary-search-tree property – elements in left subtree are smaller than datum, and elements in right subtree are larger than datum. He came up with this:

```
(define (valid-bst? bst)
    (cond ((null? bst) #t)
          (else (and (or (null? (left-branch bst))
```

```
                    (and (< (datum (left-branch bst)) (datum bst))
                         (valid-bst? (left-branch bst))))
                (or (null? (right-branch bst))
                    (and (> (datum (right-branch bst)) (datumbst))
                            (valid-bst? (right-branch bst))))))))
```

Why will Jimmy never succeed in life? Give an example that would fool his pitiful procedure.

Checking if the bst property is true for your immediate children's labels does not guarantee that the property holds for the whole subtree. For example, this tree would fool valid-bst? :

```
       10
      /  \
     5     18
    /  \
   1   30
```

The 1 violates the bst property (1 is not larger than 10), but Jimmy's algorithm will merely check that 1 is smaller than 18, and move on.

Can you do better?

2. Write `(sum-of bst)` that takes in a binary search tree, and returns the sum of all the data in the tree.

```
(define (sum-of bst)
  (cond ((null? bst) 0)
        (else (+ (bdatum bst) (sum-of (left-branch bst))
                              (sum-of (right-branch bst))))))
```

Note that the NULL? check does not violate our assertion about the lack of an "empty (binary) tree" in Scheme. (Yes, there is no such thing as an empty (binary) tree.)
It is merely in place to avoid writing more code for the cases in which the right-branch and/or left-branch do not exist.

3. Write `(max-of bst)` that takes in a binary search tree, and returns the maximum datum in the tree. The tree has at least one element.

```
(define (max-of-bst bst)
  (if (null? (right-branch bst))
      (bdatum bst)
      (max-of-bst (right-branch bst))))
```

A binary SEARCH tree ensures that every element in the left-branch subtree is less than the bdatum of the tree, and that every element in the right-branch subtree is greater than the bdatum of the tree.

4. Write `(listify bst)` that converts elements of the given `bst` into a list. The list should be in NON-DECREASING ORDER!

```
(define (listify bst)
  (if (null? bst)
      nil
      (else (append (append (listify (left-branch bst))
                            (cons (datum bst)
                                  (listify (right-branch bst))))) )))
```

Note that the NULL? check does not violate our assertion about the lack of an "empty (binary) tree" in Scheme. (Yes, there is no such thing as an empty (binary) tree.)
It is merely in place to avoid writing more code for the cases in which the right-branch and/or left-branch do not exist.

The following questions use LEAF? which is defined as follows:

```
(define (leaf? bst)
  (and (null? (left-branch bst))
       (null? (right-branch bst))))
```

5. Write `(remove-leaves bst)` that takes in a `bst` and returns the `bst` with all the leaves removed.

```
(define (remove-leaves bst)
  (cond ((null? bst) nil)
        ((leaf? bst) nil)
        (else (make-binary-tree (datum bst)
                                (remove-leaves (left-branch
bst))
                                (remove-leaves (right-branch
bst)))
         )))
```

6. Write `(height-of tree)` that takes in a tree and returns the height – the length of the longest path from the root to a leaf.

```
(define (height-of tree)
  (cond ((leaf? tree) 0)
        (else (+ 1 (max (height-of (left-branch tree))
                        (height-of (right-branch tree)))) )))
```