1 Trees As Nested Lists

- Node - leaf or list of children nodes
- Leaf - anything that is not a pair

\[
(\text{define (leaf? obj)}\n  (\text{not (pair? obj)}))
\]

\[
(\text{define test-tree '(1 (2 (3 (4) 5) 6) 7))}
\]

Draw the tree:

```
  1
 /|
/  \\
2  6
 /|
/  \\
3  5
```

2 Beam Search

- Like breadth-first search, except it checks only the best n paths of length L before moving onto paths of length L+1
- Requires some measure of the distance to the goal
- \( n = \infty \) => breadth-first search
- \( n = 1 \) => depth-first search

3 Best-First Search

- Extends the best partial path so far
- Requires some measure of the distance to the goal
4 A* Search

- Start with a one-element queue consisting of a zero-length path that contains only the root node.
- Loop until the first path in the queue terminates at the goal node or the queue is empty:
  - Remove the first path from the queue. Create new paths by extending the first path to all the neighbors in the terminal node.
  - Reject all new paths with loops.
  - If two or more paths reach a common node, keep only the path that reaches the common node with the smallest cost and remove all others.
  - Sort the entire queue by the total path length and an estimate of the remaining cost to reach the goal, with the least-cost paths in front.
- If the goal node is found, announce success. Otherwise, announce failure.
- As long as estimates of the remaining cost are lower bounds, A* search produces optimal paths to the goal.

5 Tree Manipulation

(define (tree-manip tree init leaf first rest accum)
  (cond ((null? tree) init)
        ((leaf? tree) (leaf tree))
        (else (accum
               (tree-manip (first tree) init leaf first rest accum)
               (tree-manip (rest tree) init leaf first rest accum))))
)

Use tree-manip to do the following:

a. Take the product of the even-valued leaves of the tree.

(even-product test-tree) => 48

(define (even-product tree)
  (tree-manip tree 1 (lambda (a) (if (even? a) a 1))
              car cdr *))

b. Flatten the tree.

(flatten test-tree) => (1 2 3 4 5 6 7)

(define (flatten tree)
  (tree-manip tree nil list car cdr append))

c. Deep-reverse a tree.

(deep-reverse test-tree) => (7 (6 (5 (4) 3) 2) 1)

(define (deep-reverse tree)
  (tree-manip tree nil (lambda (a) a) car cdr
              (lambda (a b) (append b (list a))))))
d. Sum up the values of the leaves of the tree.

\[
\text{(sum test-tree) } \Rightarrow 28
\]

\[
\text{(define (sum tree)} \\
\text{ (tree-manip tree 0 (lambda (a) a) car cdr +))}
\]

e. Create a new tree that keeps the odd-valued leaves of the original tree within the same tree structure, but completely removes the even-valued leaves.

\[
\text{(remove-even test-tree) } \Rightarrow (1 ((3 5)) 7)
\]

\[
\text{(define (remove-even tree)} \\
\text{ (tree-manip tree nil \\
\text{ (lambda (a) (if (even? a) nil a)) \\
\text{ car cdr \\
\text{ (lambda (a b) \\
\text{ (if (null? a) b \\
\text{ (append (list a) b))))))})}
\]