CS216 Notes Section 3 (6 February 2006)

Recursive Definitions

To solve a problem recursively:

1. Be optimistic.
   - Assume you can solve it.
   - If you could, how would you solve a bigger problem.
2. Think of the simplest version of the problem, something you can already solve. (This is the base case.)
3. Combine them to solve the problem.

Recursive Definitions on Lists

To solve a problem involving list data recursively:

1. Be very optimistic. Since lists themselves are recursive structures, we should be able to come up with a recursive definition for almost anything we can think of doing with a list.
   1. Assume we can solve a smaller version of the problem.
   2. Break the problem into the first element of the list and the rest of the list.
2. Think of the simplest version of the problem, something you can solve already. For lists, this is usually the empty list. (Sometimes, it might be the length 1 list.)
3. Combine them to solve the problem.

The PS2 LinkedList.py implementation is more complex than we'd like, because we want to deal with empty lists instead of just using None to represent them (this would break Python's type checking). So, we'll consider procedures for the ListNode class. The LinkedList procedure would just need to call these for the self.__node.

Many recursive list procedures can be defined with this template:

```python
def listproc (lst):
    if lst.__next == None:
        # what to do for a one element list
    else:
        f (g(lst[0]), lst.__next.listproc ()
```

For example:

```python
def length (self):
    if self.__next == None:
        return 1
    else:
        return 1 + self.__next.length ()
```

What are $f$ and $g$ to match the template?
Finish the `sumlist` procedure definition below. It should take a `ListNode` object as its parameter and evaluate the sum of the values of all elements in the list.

```python
def sumlist (self):
    if self.__next == None:
        return ____________________________
    else:
        return ____________________________ + self.__next.________________________
```

Define the `filterMatches` procedure that takes a `ListNode` and an element, and evaluates to the list containing all elements from the original list that do not match `el`.

```python
def filterMatches (self, el):
    if self.__next == None:
        if self.__info == el:
            ____________________________
        else:
            return self
    else:
        if self.__info == el:
            return filterMatches (self.__next, el)
        else:
            ____________________________
```

**Recursive Definitions on Trees**

To solve a problem involving trees recursively:

1. **Be extremely optimistic.** Since trees themselves are recursive structures, we should be able to come up with a recursive definition for almost anything we can think of doing with a tree.
   1. Assume we can solve a smaller version of the problem.
   2. Break the problem into the root node and the children of that node.
2. Think of the simplest version of the problem, something you can solve already. For trees, this is usually a leaf or the empty tree.
3. Combine them to solve the problem.

Finish this definition that counts the number of nodes in a tree:

```python
def size(self):
    count = 1  # count this node
    for child in children(self):
        count += child.________________________
    return count
```
Finish this definition that finds the maximal value in a tree:

```python
def largest(self):
    for child in children(self):
        if child.____________________ > max:
            max = child.largest()
    return max
```

What is the running time of `largest` defined above?

Define a procedure that implements `largest` whose running time is in $O(n)$:

```python
def largest (self):
```

**Finding All Possible Lists**

Finding all possible lists for a set of elements is much easier than finding all possible trees, but many of the same ideas apply.

What are the possible lists that can be created using the elements $\{1, 2, 3\}$?

Define a generator that generates all possible lists (here we use the Python list datatype, not the list types from PS2). (If you are stuck, see a partial solution on the next page.)
def allLists (elements):
    if len(elements) == 1:
        yield ___________________
    else:
        for index in range(len(elements)):
            # pick this element first
            first = elements[index]

            # rest is a copy of the original list with first removed
            rest = elements[:]
            del rest[index]

            assert (len(rest) == len(elements) - 1)

            for rlist in allLists (rest):
                if not rlist == None:
                    ________________________
                    yield rlist