Here are some questions that the TAs have written to help you review for the final. This review sheet covers only some of the topics after the second midterm – be sure to review other concepts as well!

**Exceptions**

What would Python print?

```python
def a():
    print("Steven says")
    int("what happens here")
    print("hello")

def b():
    try:
        print(1)
        a()
        print(2)
        return 3
    except ValueError:
        print(4)
        return 5

def c():
    b()
    a()
    return 6

>>> a()
Steven says
Traceback.... ValueError: invalid literal for int()...

>>> b()
1
Steven says
4
5

>>> c()
1
Steven says
4
Steven says
Traceback... ValueError: invalid literal for int()...
```
Dynamic Scope

```python
x = 10
def car(x):
    return van()

def van():
    if x == 10:
        return "honk"
    return "tumble"
```

What would `car(11)` return in lexical scope?
“honk”
What would `car(11)` return in dynamic scope? Draw an environment diagram.
“tumble”

Concurrency

```python
x = 10
Assume the following 2 lines are run in parallel.
>>> x = x + x
>>> x = x * x
```

Correct values of `x`:
200
400

Possible values of `x` when interleaving has occurred:
20
100
110
200 ← Note that you can get this both through correct execution and interleaved execution!
Parsing
Write the function `tokenize_html`. Given a string that includes HTML tags and text between the HTML tags, return a list which consists of HTML tags and the text between the HTML tags. See the doc-string for examples.

```python
def tokenize_html(line):
    """Given a well-formed HTML string, return a list where each element is a token. HTML tags are considered elements, and text between tags are elements.

    An HTML tag starts with a < and ends with a >. Any time a < appears, you can assume there is a corresponding >, and another < will not appear before the corresponding >.

    For example, you will NOT have to worry about a string like "<html <asdf>>".

    You may assume that the string starts and ends with an HTML tag.

    >>> tokenize_html("<html>hi</html>")
    ['<html>', 'hi', '</html>']
    >>> tokenize_html("<body>hello there    </body>"")
    ['<body>', 'hello there    ', '</body>']
    ""
    output = []
    current = ""
    for char in line:
        if char == "<":
            output.append(current)
            current = ""
            current += char
        if char == "">":
            output.append(current)
            current = ""
    return [x for x in output if x] #gets rid of empty strings
```
**Interpreters**  
Given a list returned by a call to `tokenize_html`, turn that list into a Tree structure. For example,

```python
['<a>', 'hello there', '<b>', 'lower', '</b>', 'rumble', '</a>']
```

should turn into the tree structure:

```
   '<a>'          
  /   |          
'hello there' '<b>'  'rumble'  
       'lower'
```

The tag `<a>` corresponds to the closing tag `</a>`, so the elements between those two tags become children of the Tree whose datum is `<a>`. Relevant Tree code has been included below.

```python
class Tree(object):
    def __init__(self, datum, children=[]):
        self.datum = datum
        self.children = children

    def print_tree(t, prefix=''):  
        print prefix + t.datum
        for child in t.children:
            new_prefix = '  |___';
            for _ in prefix:
                new_prefix = ' ' + new_prefix
            print_tree(child, new_prefix)

    def is_tag(token):
        return token.startswith('<')

    def is_end_tag_of(tag, token):
        return token == tag[0] + '/' + tag[1:

#continued on next page
def list_to_tree(lst):
    def eval_one(lst):
        first = lst.pop(0)
        if not is_tag(first):
            return Tree(first, []), lst
        contained = []
        nxt = lst.pop(0)
        while not is_end_tag_of(first, nxt):
            contained.append(nxt)
            nxt = lst.pop(0)
        children = []
        while contained:
            child, contained = eval_one(contained)
            children.append(child)
        return Tree(first, children), lst
    return eval_one(lst)[0]
**Iterators**

The basic idea behind the Sieve of Eratosthenes that you saw in class was that a number $n$ is prime if it is not divisible by any primes less than $n$. Write an iterator that produces prime numbers in order forever.

class Primes(object):
    def __iter__(self):
        past_primes = []
        guess = 2
        while True:
            is_prime = True
            for p in past_primes:
                if guess % p == 0:
                    is_prime = False
                    break
            if is_prime:
                yield guess
                past_primes.append(guess)
            guess += 1

**Streams**

Write `interleave`, a function that takes two streams and returns a stream where the elements of each stream are interleaved.

def interleave(s1, s2):
    def compute_rest():
        return interleave(s2, s1.rest)
    return Stream(s1.first, compute_rest)

Now create a stream of all positive and negative integers. *Hint*: The first element in the stream should be 0.

def all_integers_stream():
    pos_stream = make_positive_stream() #positive stream includes 0
    neg_stream = make_negative_stream()
    return interleave(pos_stream, neg_stream)

def make_positive_stream(first = 0):
    def compute_rest():
        return make_positive_stream(first + 1)
    return Stream(first, compute_rest)

def make_negative_stream(first = -1):
    def compute_rest():
        return make_negative_stream(first - 1)
    return Stream(first, compute_rest)