Midterm Exam ICS-33, Spring 2014

Signature _____ Name Printed _____

Lab # (1- 8) _____

LEAVE THIS TEST CLOSED, FACE UP, UNTIL YOU ARE INSTRUCTED TO BEGIN

- This is a closed book test. Keep your desk clear (no handouts, notes, books, programs, or calculators).
- You should write all your answer clearly on these pages. Make sure that your final answers are easily recognizable. If you need extra space, use the **backs of these pages** for any scratch work (indicate on the main page if all/part of your answer is on a back page). Try to solve each problem simply and directly.
- In fairness to all students in all labs, when time is called, you are to stop writing immediately.
- The number of points for each problem is clearly marked. In total, this test is worth 195 points (although you can earn 205 points). During the 110 minute testing period, you should spend about 1/2 minute per point. Not all problems are equally easy. If you get stuck, move on to the next problem; return if you have time.
- It is generally better to answer all questions briefly than to answer some completely and leave some blank.
- Sometimes information useful for one problem on the test may be found in other problems on the test.
- Students must remain in their labs until the end of the exam and not use their/our computers.
- Do not speak to anyone about the exam until 8pm. Doing so is a serious academic integrity violation.
- I expect to return graded exams within two weeks. I will post my solution within a few days of the Exam.

Problem	Points	Score
1	50	50
2	15	65
3	35	100
4	15	115
5	15	130
6	20	150
7	20	170
8	10	180
9	15	195
10	10	205
Midterm	120+75+10=205	

Problem	Points	Score

1. (50 pts) We will store a point in 2-dimensional space as a 2-tuple: its x-coordinate followed by its y-coordinate, like (0.0, 0.0). **Clustering** associates each point (in a list) to its closest reference point (from another list).

Define the following four functions to help perform and analyze clustering.

(a) Define a function **closest** that has two arguments: a list of reference points (a list of one or more 2-tuples) and a single point (a 2-tuple); it returns the reference point (2-tuple) that is closest to the single point. Call the **dist** function, defined below, which returns the distance between any two points (2-tuples).

```
def dist(pt1,pt2):
    return math.sqrt( (pt1[0]-pt2[0])**2 + (pt1[1]-pt1[0])**2)

def closest(ref_pts, pt):
    answer   = ref_pts[0]
    min_dist = dist(pt,answer)
    for x in ref_pts[1:]:
        d = dist(pt,x)
        if d < min_dist:
            answer   = x
            min_dist = d
    return answer
```

```
def closest (ref_pts, pt):
    answer = ref_pts[0]
    for x in ref_pts[1:]:
        if dist(pt,x) < dist(pt,answer):
            answer = x
    return answer
```

(b) Define a function **cluster** that has two arguments: a list of reference points and a list of points; it returns a dictionary (use a **dict**) such that (a) each reference point is a key in the dictionary, and (b) each reference point is associated with a list (possibly empty) of all the points from the list of points that are closest to it. Call the **closest** function specified above, whether or not you wrote it correctly.

```
def cluster(ref_pts, pts):
    cluster_dict = {pt:[] for pt in ref_pts}
    for pt in pts:
        cluster_dict[closest(ref_pts,pt)].append(pt)
    return cluster_dict
```

(continued)

(c) Define a function **sort** that has one argument: a dictionary of the kind returned by the **cluster** function describe in part(b); **sort** returns a list of 2-tuples, each consisting of a reference point and its associated points, ordered by the reference point's distance from the origin (the tuple **(0,0)**): closest to the origin first, farthest away from the origin last. Do not include in the returned list any reference points associated with an empty list of points. Call the **dist** function defined in part (a).

```
def sort(cluster_dict):
   return [c for c in sorted(cluster_dict.items(), key = lambda x : dist((0,0),x[0]))\
              if len(c[1]) > 0]


   return sorted((c for c in cluster_dict.items() if len(c[1])>1),\
                     key = lambda x : dist((0,0),x[0]))
```

(d) Define a function **error** that has one argument: a dictionary of the kind returned by **cluster**; it returns the sum of the distances between every reference point and all the points in its associated list. You may call the **dist** function defined in part (a).

```
def error(cluster_dict):
    return sum(dist(ref_pt,pt) for ref_pt,pts in cluster_dict.items() for pt in pts)
```

2. (15 pts) (a) State whether the Regular Expression (RE) pattern **`"^(a+)[xyb](bb)?$"`** matches each of the following test strings (write **T**rue or **F**alse): for those it matches, show the groups produced by the match.

Test	Match?	If Match? is **T**rue, the groups (starting with group 0) are…
`"xbb"`	F	
`"aaxybb"`	F	
`"aaax"`	T	**`"aaax"`**, **`"aaa"`**, **`""`**
`"aaabbb"`	T	**`"aaabbb"`**,**`" aaa"`**, **`"bb"`**

(b) Rewrite the ***`list`*** EBNF descriptions as an equivalent Regular Expression (RE) pattern string. Do not use **`\d`**.

`digit` ⇐ **`0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`**
`integer` ⇐ **`[+|-]`*`digit`*`{`*`digit`*`}`**
`list` ⇐ *`integer`*`{,`*`integer`*`}`**

Regular Expression (RE) pattern string: **`"[+-]?[0-9]+(,[+-]?[0-9]+)*"`**

3. (35 pts) The following class starts to define a **Fraction** as two **int** values: **numerator** and **denominator**. For example, we can write **a = Fraction(3,2)** to represent the fraction **3/2** and **b = Fraction(5,3)** to represent the fraction **5/3**. In this problem, assume all fractions have positive numerators and denominators.

In the class below, overload the **repr** function, the multiplication operator (so that we can compute **a*b** and **a*2** and **2*a**: note that multiplication is commutative), and the less-than operator (so that we can compute **a<b** and **a<2**, but does not have to correctly compute **2<a**). Note the semantics for implementing * and < on fractions are

$$\frac{m}{n} * \frac{o}{p} = \frac{mo}{np} \qquad\qquad \frac{m}{n} < \frac{o}{p} \text{ exactly when } mp < no$$

For any other type of operands in * and < raise a **TypeError** with appropriate information.

```
class Fraction:
    def __init__(self,numerator,denominator):
        self.n = numerator
        self.d = denominator

    # overload repr
    def __repr__(self):
        return 'Fraction('+str(self.n)+','+str(self.d)+')'



    # overload * allowing only Fraction * Fraction, Fraction * int, and int * Fraction
    def __mul__(self,right):
        if type(right) is int:
            return Fraction(self.n*right , self.d)
        elif type(right) is Fraction:
            return Fraction(self.n*right.n , self.d*right.d)
        else:
            raise TypeError('Fraction.__mul__: right('+str(right)+') not Fraction/int')

    def __rmul__(self,left):
        return self*left # let __mul__ do the job, 2*Fraction(3,2) = Fraction(3,2)*2







    # overload < allowing only Fraction < Fraction and Fraction < int
    def __lt__(self,right):
        if type(right) is int:
            return self.n < self.d*right
        if type(right) is Fraction:
            return self.n*right.d < self.d*right.n
        else:
            raise TypeError('Fraction.__lt__: right('+str(right)+') not Fraction/int')
```

4. (15 pts) Define a class named **Poly** (for polynomial) whose constructor takes an argument that is a list of terms: 2-tuples whose first index specifies a **float coefficient** and whose second index specifies an **int power**. For example, we might write **p = poly ([(3.0, 2), (2.0, 1), (-1.0, 0)])** which represents a polynomial with the terms $3.0x^2 + 2.0x - 1.0$.

- Define the **__init__** method for **Poly** to store its argument as a dictionary whose keys are the **powers** of x and whose associated values are the **coefficients** of x; for the example **p** above, store its dictionary as **{2: 3.0, 1: 2.0, 0: -1.0}**.
- Define the **__getitem__** method for **Poly** so that (given the polynomial above) **p[2]** would return **3.0**: the coefficient associated with the 2nd power. Note that **p[3]** should return **0.0**, because there is no 3rd power in this polynomial, and any argument that is not a non-negative **int** should raise the **TypeError** exception with an appropriate message.
- Define the **__call__** method for **Poly** so that (given the example above) **p(1.5)** returns **8.75**: the value resulting from evaluating the polynomial with x = **1.5**. Use calls to **__getitem__** to simplify your code. You may assume (and not check) that the argument to **__call__** is a single **float** value.

```
class Poly:

    # overload __init__
    def __init__(self,terms):
        self.terms = {power:coef for coeff,power in terms}




    # overload __getitem__
    def __getitem__(self,index):
        if type(index) is not int or index < 0:
            raise TypeError('Poly.__getitem__: index('+str(index)+') not int or < 0')
        elif index in self.terms:
            return self.terms[index]
        else:
            return 0.0




    # overload __call__
    def __call__(self,arg):
        return sum(self[power] * arg**power for power in self.terms)
        #return sum(coef * arg**power for coeff,power in self.terms.items())
```

5. (15 pts) Recall that we defined a Finite Automaton (FA) by specifying a dictionary whose (a) keys are **starting states** and whose (b) associated values are dictionaries, whose (b1) keys are **inputs** and whose (b2) associated values are **result states**. For example, the **parity** FA is represented by the following dictionary.

```
{'even': {'1': 'odd', '0': 'even'}, 'odd': {'1': 'even', '0': 'odd'}}
```

Assume that we have defined an **`FA`** class whose **`self.fa`** attribute stores such a dictionary. Write an iterator for this class which produce 3-tuples of **starting states** and **inputs** and **result states**, completely describing the FA, where the resulting tuples appear as if sorted.

For the parity FA, the tuples would be produced in the following order: **`('even', '0', 'even')`**, **`('even', '1', 'odd')`**, **`('odd', '0', 'odd')`**, and **`('odd', '1', 'even')`** because **`'even' < 'odd'`** and **`'0' < '1'`**.

Write this code as simply as possible, using a generator.

```
class FA:
  # Assume correct definitions of methods: __init__, defining self.fa as the dictionary

  # overload __iter__ as a generator, using self.fa as the dictionary
   def __iter__(self):
       for ss,irsd in sorted(self.fa.items()):
           for i,rs in sorted(irsd.items()):
               yield (ss, i, rs)
```

6. (20 pts) Using the proof rules for recursive functions, prove that the following **separate** function (from Quiz 5) is correct. Recall **separate** is passed a predicate and a **list**; it returns a 2-**tuple** whose **0** index is a list of all the values in the argument list for which the predicate returns **True**, and whose **1** index is a list of all the values in the argument list for which the predicate returns **False**.

```
def separate(p,l):
    if l == []:
        return [],[]
    else:
       t_list,f_list = separate(p,l[1:])
       if p(l[0]):
           return [l[0]]+t_list, f_list
       else:
           return t_list       , [l[0]] + f_list
```

1. It recognizes the base case as an empty list and it returns the correct result: a 2-tuple of empty lists, because there are no values for either list.

2. Each recursive call in **separate** has an argument that is closer to the base case by passing an argument that is a list whose length is smaller by one.

3. Assuming that **t_list** and **f_list** (the two list results returned by the recursive call to **separate**) store the separated values for every value in **l** other than the one at index **0**, then this function returns **l[0]** concatenated onto the correct list, with the other list remaining the same, which is the correct tuple for all values in the list **l**.

7. (20 pts) Write a generator named **skip_until** that takes as arguments a predicate and any iterable; it doesn't produce the **initial** values produced by the iterable for which the predicate returns **False**, and then produces the first value for which the predicate evaluates to **True** and produces all others after that one whether or not they satisfy the predicate.

For example, iterating over **skip_until(lambda x : len(x) > 1, ['x', 'y', 'abc', 'a', 'xyz'])** would skip producing **'x'** and **'y'** (because their lengths are not **> 1**) but then produces the values **'abc'**, **'a'**, and then **'xyz'** (because **'abc'** is the first value whose length is **> 1**, so it and all subsequent values are produced). Hint: call **iter** explicitly.

```
def skip_until(p,iterable):
    it = iter(iterable)
    for i in it:
        if p(i):
            yield i
            break
    while True:
        yield next(it)
```

8. (10 pts) State the fundamental equation of object-oriented programming.

```
o.m(...) = type(o).m(o,...)
```

9. (15 pts) The **Fraction** class (in question 2 in this exam) defines only a **<** operator that can correctly compare a **Fraction** either to another **Fraction** or to an **int**.

(a) Explain in detail (hint: use 8 above) what happens if we ask Python to evaluate **1 < Fraction (1,2)** and why it fails to successfully compute a result with only **<** defined..

By FEOOP, it first calls **int.__lt__(1, Fraction (1,2))** but it finds that this method raises an exception (the **int** class, written before the **Fraction** class, doesn't know about comparing **int** values to **Fractional** values).

It tries to call **Fraction.__gt__** with the operands reversed as **Fraction.__gt__(Fraction (1,2), 1)** but according to the "only the **<** operator is defined condition" above, this method doesn't exist so Python raises an exception that propagates to the user.

(b) Explain what we need to do to allow Python to evaluate this expression successfully and explain why it succeeds.

Define a **>** operator in **Fraction**. Now calling **Fraction.__gt__(Fraction (1,2), 1)** succeeds: the function exists and it will compute the right answer when called after **int.__lt__**fails.

10. (10 pts) The argument to the **defaultdict** constructor is a reference to a class/function object, **that if called with no argument** returns the object to associate automatically with a new key in the dictionary: e.g., **d = defaultdict(int)** associates each new key with **int()**; **d = defaultdict(list)** associates each new key with **list()**.

Suppose that we wanted to make a **defaultdict** such that its new keys were to be automatically associated with a list storing the single value **None**. Fill in the parentheses with a **lambda** to accomplish this goal.

```
d = defaultdict(lambda : [None]                                    )
```