When working on this quiz, recall the rules stated on the Academic Integrity statement that you signed. You can download the **q5helper** project folder (available for Friday, on the **Weekly Schedule** link) in which to write/test/debug your code. Submit your completed **q5solution** module online by Thursday, 11:30pm. I will post my solutions to Edu Resources reachable via the **Solutions** link on Friday morning.

Ground Rules: The purpose of your solving problems on this quiz is to learn how to write directly-recursive functions. Try to synthesize your code carefully and deliberately, using the general form of all recursive functions and the 3 proof rules discussed in the notes for synthesizing recursive functions. Remember, it is Elephants all the way down. Try to write the minimal amount of code in each function. Some errors will not be easy to debug using the debugger or print statements; in such cases, use the 3 proof rules.

Use **no `for`** loops or comprehensions (which include `for` loops) in your code. I did not use local variables in any of my functions; if you use local variables, each can be **assigned a value only once** in a call, and it **cannot be re-assigned or mutated**; try to use no local variables. Of course, **do not mutate** any parameters. Don't use `try`/`except`. To simplify adherence to these rules, some problems use `tuple` (instead of `list`) parameters/results: you will often concatenate `tuples` to create larger `tuples`: e.g., `(1) + (2, 3)` computes the `tuple (1, 2, 3).`

1. (2 pts) Define a **recursive** function named `odd`; it is passed one `tuple` argument; it returns a `tuple` of the 1st, 3rd, 5th, etc. values from the `tuple` argument. For example, `odd( ('a','b','c','d','e','f','g') )` returns `('a','c','e','g')`.

2. (4 pts) Define a **recursive** function named `compare`; it is passed two `str` arguments; it returns one of three `str` values: `'<'`, `'='`, or `'>'` which indicates the relationship between the first and second parameter (how these strings would compare in Python). For example, `compare('apples','oranges')` returns `'<'` because in Python, `'apples' < 'oranges'`. Hint: My solution had 3 base cases that compare the parameter strings to the empty string; the non-base case compares only the **first character** in one string to the **first character** in the other. Your solution must not do much else: it **cannot** use relational operators on the entire strings, which would make the solution trivial; it can use relational operators, **but only on empty strings and single character strings**.

3. (9 pts) The three functions below all concern a functional way to store a dictionary. We store a dictionary as a `tuple` of 2-`tuples`: index `0` of each 2-`tuple` is a key; index `1` is that key's associated value. The order of the keys is irrelevant, but keys are unique: no 2-`tuples` can have the same keys. For example the dictionary `{'a': 1, 'b': 2, 'c': 3}` might be stored as `(('b',2), ('a',1), ('c', 3))` –or any other `tuple` with these 2-`tuples` in any order. We call such a data structure an **association tuple**.

- (3 pts) Define a **recursive** function named `get_assoc`; it is passed an association `tuple` and key as arguments; it returns the value in the association `tuple` associated with the key. If the key is not in the association `tuple`, raise the `KeyError` exception.
- (3 pts) Define a **recursive** function named `del_assoc`; it is passed an association `tuple` and key as arguments; it returns an association `tuple` that contains all associations except the one specified by the key parameter (keep the order of the other keys in the association `tuple` the same). If the key is not in the association `tuple`, raise the `KeyError` exception. Hint: build a new association `tuple` with all associations but the deleted one.
- (3 pts) Define a **recursive** function named `set_assoc`; it is passed an association `tuple`, key, and associated value as arguments; it returns an association `tuple` that contains all associations with (a) a new 2-`tuple` added at the end for this association (if the key is nowhere in the association **tuple**) or (b) one of the associations changed (if the key is already in the `tuple`). Keep the key order the same. For example,

  `set_assoc( ( ('a',1), ('c',3)), 'b', 2 )` returns `( ('a',1), ('c',3), ('b',2) )`
  `set_assoc( ( ('a',1), ('c',3), ('b',2) ), 'c', 13)` returns `( ('a',1), ('c',13), ('b',2) )`

Hint: build a new association tuple with all associations adding the new 2-**tuple** at the end for a new key or creating a new 2-**tuple** for an existing key.

4. (5 pts) Define a **recursive** function named **immutify**; it is passed any data structure that contains **int**, **str**, **tuple**, **list**, **set**, **frozenset**, and **dict values** (including nested versions of any of these data structures as an argument). It returns an **immutable equivalent data structure** (one that could be used for values in a **set** or keys in a **dict**). The types **int**, **str**, and **frozenset** are already immutable. Convert a **set** to a **frozenset**; convert all the values in a **tuple** to be their immutable equivalents, in the same order); convert a **list** to a **tuple** (with immutable equivalents of its values, in the same order); convert a **dict** to **tuple** of 2-**tuples** (see the association tuple in question 2) but here with the 2-**tuples** sorted by the **dict**'s keys. If **immutify** ever encounters a value of another type (e.g., **float**) it should raise a **TypeError** exception. The following call (with many mutable data structures)

```
immutify( {'b' : [1,2], 'a' : {'ab': {1,2}, 'aa' : (1,2)}} )
```

returns the immutable data structure

```
(('a', (('aa', (1, 2)), ('ab', frozenset({1, 2})))), ('b', (1, 2)))
```

Hints: What is the base case, returning simple results? You can convert a **set** into a **frozenset** just by using its constructor. When converting values is a **tuple** to be immutable, or converting a **list** to a **tuple**, you must use recursion (so what are the base and recursive cases?) - not use a **for** loop or comprehension. In this problem, for **dict** only, you may use a **for** loop or comprehension to iterate through the **dict** to convert it to an association tuple (although one is not necessary, it is more efficient). Note that values in **sets** and keys in **dicts** must already be immutable.

5. (5 pts) Define a **recursive** function named **balanced**; it is passed a **str** containing only the left and right parenthesis characters. It returns a **bool** that indicates whether or not the parentheses appear in a way that they correctly match. For example `()` correctly match; `((()))` correctly match; `((()()))(()())` correctly match; but `)(` doesn't correctly match; `(()()()` doesn't correctly match; etc. Here are the two general rules to use to write your recursive program. Parentheses correctly match if

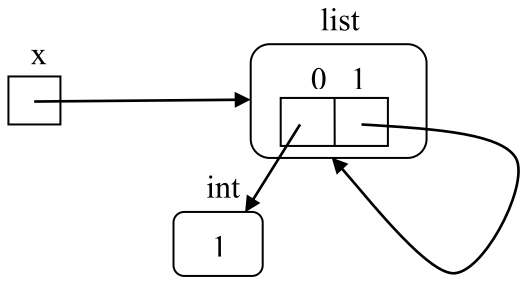
- (nested rule): the first and last characters of a **str** are `(` and `)` respectively, and the **str** inside them has parentheses that correctly match.
- (sequential rule): there is **any** index **i** in the **str**, such that it splits the **str** into two substrings that each have parentheses that correctly match

In this problem you can make full use of Python, including rebinding, iteration, comprehension, mutation etc. Use such power sparingly.

There is simple iterative solution that repeated replaces all `'()'` by `''` until it results in the empty string (parentheses are balanced) or cannot make any more replacements (parentheses aren't balanced). Do not use this solution.

(extra credit problem on next age)

6. (1 pt extra credit) Define a **recursive** function named `my_str`; it is passed an `int` or `list` whose elements are `ints` or `lists` of `ints`, or `lists` of `lists` of `ints`, etc. It returns a string: the same string that `str` returns when called on such an argument (but, of course, you cannot call `str` or `__str__` etc. in your answer). You must write your own recursive `my_str` function that solves this problem). `my_str` must deal with references inside the `list` that refer to the `list`. For example, if we wrote `x = [1,1]` and then `x[1] = x` the following structure results.



`str(x)` returns `[1, [...]];`

`my_str(x)` should return the same.

In a recursive call, when asked to convert a `list` that is already being converted, immediately return `[...]`.

An even more subtle example (draw it) is `x,y = [1],[1]` and `x[0]=y` and `y[0]=x`; `str(x)` is `[[[...]]]`.

The driver/`bsc` contains other, more complex examples. Create your own self-referential `lists` and call `str` to see what `my_str` should return. Hint: I wrote my function to define a `dict` and a recursive helper function (that can examine/mutate the `dict`); `my_str` just returns the result of calling the helper. The helper converts the `list` (and sublists, and subsublists, …) into a string. The key observation, which I'll state non-procedurally, is to return `[...]` for a `list` while in the process of building its string (while recursively exploring that `list` object if another reference to it shows up inside itself), but for later occurrences of that `list`, return the string actually built from its constituent values. You must build a string for every list object while avoiding infinite (mutual) recursion. Use the `id` of a list as the key to the `dict`. Recall the `id` function: calling `id(o)` returns an `int` corresponding to the location in memory of object `o`. We know `x is y` is `True` exactly when `id(x)==id(y)` is `True`. In this problem you may use `for` in a comprehension to help (I called `.join` on it). Don't worry if you cannot solve this problem, it is extra credit.