1)
When short circuiting we are left associaive and dont evaluate parenthesis until we need to. So we will evaluate the sequence from left to right. If we come across an or we take the left side as argument 1 and the entire sequence to the right of the or as its second argument. If its first argument is true, it doesnt care about the second arg and wont even evaluate it. But if it is false, we must evaluate the sequence to the right. If we come across an and we take in the left side as the first arg and the entire sequence to the right as its second arg. If the first arg is false, we never evaluate the second argument. We evaluate a parenthesis only when we need its value.

2)
a)
Interface A has no supertypes.
Interface B has a supertype of A.
Interface C has a suprtype of A.
Class D has a supertypes of A, B, and C.
Class E has a supertypes of C and A.
Class F has supertypes of A, B, C, D.
Class G has supertypes of A, B.

b)
A function that takes in a parameter of type B can accept a class of type B, D, F, and G

c)
Bletch cannot call bar as A is not a subclass of C so it may not be used where C is used.

3)
Inheritance is the practice of defining a new "subclass/subtype" based on an existing class or interface. Subtype polymorphism does not create new classes but is instead the ability to use a subclass whenever as superclass is expected. Dynamic dispatch is a technique used to determine during runtime which function within a class to use when we are using subtype polymorphism.

4)
We cannot use sybtype polymorphism in a dynamically typed language as variables do not actually have types. We cannot use dynamic dispatch in dyncamically-types langs as we do not use subtype polymorphism an thus dont have to worry about it. When a method is called on a object, we check if the object supports it, if not, we check the superclass object stored within the subclass for the method.

5)
The violation I noticed is a violation of the Dependency Inversion Principle. If there were multiple types of chargers that an electric vehicle could use, then we would want the electric vehicle to support any type of charger rather than just super chargers. So we would have a base interface Charger that SuperCharger would implement and Electric vehicle would take in for its charge() function.

6)
Liskovs substitution principle still applies as we want standardized practices even when ducktyping.

7)

```
class Node:
  def __init__(self, val):
    self.value = val
    self.next = None

class HashTable:
  def __init__(self, buckets):
    self.array = [None] * buckets
  def insert(self, val):
    bucket = hash(val) % len(self.array)
    tmp_head = Node(val)
    tmp_head.next = self.array[bucket]
    self.array[bucket] = tmp_head

a)
def gen(hash_table):
  for i in range(len(hash_table.array)):
    tmp = hash_table.array[i]
    while tmp != None:
      yield tmp
      tmp = tmp.next
#no modification to HashTable needed

b)
class OurIterator:
  def __init__(self, arr):
    self.arr = arr
    self.pos = 0
    self.list = self.arr[self.pos]
  def __next__(self):
    if self.list != None:
      val = self.list
      self.list = self.list.next
      return val
    else:
      self.pos += 1
      if self.pos < len(self.arr):
        self.list = self.arr[self.pos]
        val = self.list
        self.list = self.list.next
        return val
      else:
        raise StopIteration
class HashTable:
  def __init__(self, buckets):
    self.array = [None] * buckets
  def __iter__(self):
    it = OurIterator(self.array)
    return it
  def insert(self, val):
    bucket = hash(val) % len(self.array)
    tmp_head = Node(val)
    tmp_head.next = self.array[bucket]
    self.array[bucket] = tmp_head
```

c)
generator:

```
x = gen(ht)
for n in x:
  print(n.value)
```
or
```
for n in gen(ht):
  print(n.value)
```
---
iterator class:

```
for n in ht:
  print(n.value)
```
---

d)
```
iter = ht.__iter__()
try:
  while True:
    i = iter.__next__()
    print(i.value)
except StopIteration:
  pass
```

e)
```
class HashTable:
  def __init__(self, buckets):
    self.array = [None] * buckets
  def __iter__(self):
    it = OurIterator(self.array)
    return it
  def insert(self, val):
    bucket = hash(val) % len(self.array)
    tmp_head = Node(val)
    tmp_head.next = self.array[bucket]
    self.array[bucket] = tmp_head
  def forEach(self, func):
    for i in range(len(self.array)):
      tmp = self.array[i]
      while tmp != None:
        func(tmp)
        tmp = tmp.next
```

8)
I would use the function type value in my brewin++ implementation. It already took in a environment as a optional parameter so I could modify the code to store the environment of the lambda when yield keyword is encountered.