Notes to the lab assistant:

- Your job is not to lecture. Your job is to guide the students into thinking for themselves.
  - Let the students lead. When they get stuck, ask them questions to help them along. (Unless they're stuck on something trivial or fundamental, then just tell them)
  - Don't be a jerk. Don't condescend. Don't patronize. You were once a noob.
- If they can try it out in the interpreter, don't answer the question. Tell them to try it out on the interpreter. Bit by bit, we want to get them to rely on themselves. Self-reliance = good
  - "What if" questions are answered MUCH faster by a computer than by us
- When checking off, have them explain the solution to you as a group. If the problem has multiple parts, have each person explain a part.
  - Ask follow up questions to the students who didn't get to speak.
  - Don't allow students to take over for other students.
  - There's more than one solution on the coding questions, so you'll have to be careful on those. Have them check in the interpreter.
- **Don't be afraid to refuse to check them off. Half assed or incomplete answers should be rejected.**
  - Target questions at the student that has the worst understanding.
- **If a student is bored, maybe move them to another group that's ahead.**
  - Conversely, if a student seems like they can't keep up with the rest of the group, move them to another group that's going closer to their pace

NOTE: If they haven't tested the function by trying it out in the interpreter, don't check them off.
Link to this document: tinyurl.com/mswehqd

Higher Order Functions and Environment Diagrams

Instructions: Form a group of 4-5. Start on problem 0. Check off with a lab assistant when you think everyone in your group understands how to solve problem 0. Repeat for problem 1, 2, ...
You're not allowed to move on from a problem until you check off with a lab assistant.
You are allowed to use any and all resources at your disposal, including the interpreter, lecture notes and slides, discussion notes, labs, and the lab assistants. The purpose of this section is to have all the students working together to learn the material and get better at Computer Science together.

0a. What do lambda expressions do?
Lambda expressions create functions. When you evaluate one, you get out a function as a value

0b. Explain the difference between the following:
```
>>> def square(x):
...   return x * x
```
Defines a function square

```
>>> square(4)
```
Calls the function square

```
>>> square
```
Evaluates to the actual function that does squaring

```
>>> square = lambda x: x * x
```
Defines the squaring function using assignment and a lambda expression

0c. Determine if each of the following will error:
```
>>> 1/0
```
`Error`

```
>>> def boom():
      return 1/0
```
`No error, since we don't evaluate the body of the procedure when we define it.`

```
>>> boom()
```
`Error`

1a. Express the following expressions using lambdas instead of their named counterparts:

i. `square(4)`
```
(lambda x: x * x)(4)
```

ii. `sum_of_squares(3, 4)`
```
(lambda x, y: x * x + y * y)(3, 4)
```

iii.
```
def hello_world():
  return "hello world!"
hello_world()
(lambda: "hello world!")()
```

1b. What will Python output?

i. `(lambda x: x(x))(lambda y: 4)`

4

Hint 1: What is the operator? What are the operands?
Hint 2: Explain what each lambda expression does in English.

ii. `(lambda x, y: y(x))(mul, lambda a: a(3, 5))`

15

2a. Draw an environment diagram for the following

```
>>> def make_adder(n):
...   return lambda x: x + n
...
>>> add_4 = make_adder(4)
>>> add_4(3)
7
```

Hint: http://www.ocf.berkeley.edu/~shidi/cs61a/guerrilla/env.txt

```
 _______________
|Global         |
|  make_adder [-----> func make_adder(n) parent=Global
|       add_4 [--------.
|_______________|      |
|f1    parent=G |      |
|  make_adder(4)|      v
|return value [-----> func lambda(x) parent=f1
|_______________|
|f2    parent=f1|
|      lambda(3)|
|return value [7|
|_______________|
```

Stack

Global

~~F1   make_adder~~

~~F2   lambda~~

2b. Draw an environment diagram for the following

```
>>> def curry(f, x):
...   return lambda y: f(x, y)
...
>>> def square(x):
...  return x * x
...
>>> sum_of_squares = lambda x, y: square(x) + square(y)
>>> warped_square = curry(sum_of_squares, 3)
>>> warped_square(4)
25
# Tricky; Use a new environment diagram (otherwise it'd be cluttered)
>>> (lambda: curry(sum_of_squares, 3)(4))()
25
# OPTIONAL; Start with a clean sheet of paper for this
>>> curry(sum_of_squares, 3)((lambda x: 4)(lambda: 1/0))
25
```

Ask Andrew for a copy of these. Not going to try to ASCII these out. =_=. It's ok, I will ascii them. Lol I can see why you didn't ascii them. It's going onto the next page. Actually, I'll just move it all to the next page

```
warped_square(4)

 ______________________
| Global               |
|             curry [-------> func curry(f, x) parent=Global
|            square [-------> func square(x) parent=Global
|    sum_of_squares [-------> func lambda(x, y) parent=Global   <--.
|     warped_square [---------.                                    |
|______________________|       |                                    |
 ______________________        |                                    |
| curry                |       |                                    |
|                 f [---------+------------------------------------'
|                  x [  3|      v
|      return_value [-------> func lambda(y) parent=curry
|______________________|

 ______________________
|         parent=curry |
|                  y [  4|
|      return_value [25|
|______________________|

 ______________________
|        parent=Global |
|                  x [  3|
|                  y [  4|
|      return_value [25|
|______________________|

 ______________________
| square               |
|                  x [  3|
|      return_value [  9|
|______________________|

 ______________________
| square               |
|                  x [  4|
|      return_value [16|
|______________________|
```

Stack_______

Global

~~F1   curry~~

~~F2   lambda~~

~~F3   lambda~~

~~F4   square~~

~~F5   square~~

```
(lambda: curry(sum_of_squares, 3)(4))()

 ______________________
| Global               |
|            curry [-------> func curry(f, x) parent=Global
|           square [-------> func square(x) parent=Global
|   sum_of_squares [-------> func lambda(x, y) parent=Global  <--.
|______________________|                                          |
 ______________________                                           |
|        parent=Global |         __________________________________|
|      return_value [25|        |
|______________________|        |
 ______________________         |
| curry                |        |
|                  f [----------’
|                  x [ 3|
|      return_value [-------> func lambda(y) parent=curry
|______________________|
 ______________________
|         parent=curry |
|                  y [ 4|
|      return_value [25|
|______________________|
 ______________________
|        parent=Global |
|                  x [ 3|
|                  y [ 4|
|      return_value [25|
|______________________|
 ______________________
| square               |
|                  x [ 3|
|      return_value [ 9|
|______________________|
 ______________________
| square               |
|                  x [ 4|
|      return_value [16|
|______________________|
```

Stack

Global

~~F1   lambda~~

~~F2   curry~~

~~F3~~	~~lambda~~
~~F4~~	~~lambda~~
~~F5~~	~~square~~
~~F6~~	~~square~~

```
curry(sum_of_squares, 3)((lambda x: 4)(lambda: 1/0))

 _______________________
| Global                |
|              curry [-------> func curry(f, x) parent=Global
|             square [-------> func square(x) parent=Global
|     sum_of_squares [-------> func lambda(x, y) parent=Global   <--.
|_______________________|                                         |
 _______________________                                          |
| curry                 |                                           |
|                  f [----------------------------------------------'
|                  x [ 3|
|       return_value [-------> func lambda(y) parent=curry
|_______________________|
 _______________________
|          parent=curry |
|                  y [ 4|
|       return value [25|
|_______________________|
 _______________________
|         parent=Global |
|                  x [-------> func lambda() parent=Global
|       return_value [ 4|
|_______________________|
 _______________________
| square                |
|                  x [ 3|
|       return_value [ 9|
|_______________________|
 _______________________
| square                |
|                  x [ 4|
|       return_value [16|
|_______________________|
```

Stack________

Global

~~F1   curry~~

~~F2   lambda~~

~~F3   lambda~~

~~F4   square~~

~~F5   square~~

3. Write a `make_skipper`, which takes in a number `n` and outputs a function. When this function takes in a number `x`, it prints out all the numbers between 0 and `x`, skipping every `n`th number.

```
>>>a = make_skipper(2)
>>>a(5)
1
3
5
def make_skipper(n):
  def skipper(x):
    c = 0
    while c <= x:
      if c % n != 0:
        print(i)
      c += 1
  return skipper
```

4. Write `compose`:

```
>>> a = compose(lambda x: x * x, lambda x: x + 4)
>>> a(2)
36
def compose(f, g):
  return lambda x: f(g(x))
```

5. Write make_alternator.

```
>>> a = make_alternator(lambda x: x * x, lambda x: x + 4)
>>> a(5)
1
6
9
8
25
```

```
def make_alternator(f, g):
  def alternator(n):
    i = 1
    while i <= n:
      if i % 2 == 1:
        print(f(i))
      else:
        print(g(i))
      i += 1
  return alternator

# Alternatively
>>> def make_alternator(f, g):
...   def alternator(n):
...     for i in range(1, n+1):
...       print(f(i)) if i % 2 == 1 else print(g(i))
...   return alternator
```

6. Here is a version of cons:

```
def cons(a, b):
  return lambda m: m(a, b)
```

Write car and cdr:

```
def car(f):
 return f(lambda a, b: a)
```

```
def cdr(f):
 return f(lambda a, b: b)
```

(Optional) draw an environment diagram for the following:

```
z = cons('hello, 'world')
cdr(z)
```

7. The logician Alonzo Church invented a system of representing non-negative integers entirely using functions. Here are the definitions of 0, and a function that returns 1 more than its argument:

```
def zero(f):
    return lambda x: x

def successor(n):
    return lambda f: lambda x: f(n(f)(x))
```

This representation is known as Church numerals. Define one and two directly, which are also functions. To get started, apply successor to zero. Then, give a direct definition of the add function (not in terms of repeated application of successor) over Church numerals. Finally, implement a function church_to_int that converts a church numeral argument to a regular Python int:

```
def one(f):
    """Church numeral 1."""
    return lambda x: f(zero(f)(x)) # Alternatively f(x)

def two(f):
    """Church numeral 2."""
    return lambda x: f(one(f)(x)) # Alternatively f(f(x))

def church_to_int(n):
    """Convert the Church numeral n to a Python integer.

    >>> church_to_int(zero)
    0
    >>> church_to_int(one)
    1
    >>> church_to_int(two)
    2
    """
    return (n(lambda x: x + 1))(0)
```

```
def add_church(m, n):
    """Return the Church numeral for m + n, for Church numerals m and n.

    >>> three = successor(two)
    >>> church_to_int(add_church(two, three))
    5
    """
    return lambda f: lambda x: m(f)(n(f)(x))

def mul_church(m, n):
    """Return the Church numeral for m * n, for Church numerals m and n.

    >>> three = successor(two)
    >>> four = successor(three)
    >>> church_to_int(mul_church(two, three))
    6
    >>> church_to_int(mul_church(three, four))
    12
    """
    return lambda f: lambda x: m(n(f))(x) # Alternatively, lambda f:
m(n)(f)
```