# CS61A Lecture 6

Monday, September 9th, 2019

## Announcements

- Hog project party today in Cory 241 from 6:30 pm to 8:00 pm.
- Project due on Thursday, submit on Wednesday for extra point.
- There will be a list of questions that can safely be ignored as they cover content beyond the scope of what we have studied.
- There will be less new content this week.

## Return

### Return statements

A return statement completes the evaluation of a call expression and provides its value.

- `f(x)` for user defined `f` : swtich to a new environment and execute `f` 's body in that new environment
- `return` statement within `f` : switch back to the previous exnvironment; `f(x)` now has a value.

Only one return statement is ever executed while executing the body of a function, any code after is ignored.

```
def end(n,d):
    """Print the final digits of N in reverse order until D is found.
    >>> end(34567,5)
    7
    6
    5
    """
    while n>0:
        last, n = n, n//10
        print(last)
        if d == last:
            return None
```

This function has no True return value because all the important work has been done, and the return value is not important.

### Search

The function below is a higher order function that returns the lowest integer value of `x` for which `f(x)` is a true value.

```
def search(f):
    x=0
    while True:
        if f(x):
```

```
            return x
        x = x + 1
```

What is `f(x)` ? It could be, for example, `positive` :

```
def positive(x):
    """Zero until square(x)-100 is positive, then it's not zero."""
    return max(0,square(x)-100)
```

If we input `search(positive)` , then we return 11, the lowest value for which the value is not None.

We could also write an `inverse` function, to find the inverse of another function:

```
def inverse(f):
    """Find a function G such that G(F(X)) --> X.
    """
    return lambda y: search(lambda x: f(x) == y)
```

`inverse` calls a function because it returns a lambda expression, which takes in whatever parameter is passed in. For example

```
square_root = inverse(square)
```

`inverse` expects a function, and if we try to input a number into it, it will return an error that the value is not callable, e.g. it can't be called, you know, like a function.

# Self-Reference

This concept pertains to number 6 and 7 of the Hog project.

Let's do a really simple function that refers to itself.

```
def print_all(k):
    print(k)
    return print_all
```

And now let's call `print_all` on:

```
print_all(1)(3)(5)
```

In this case, the function as a whole is evaluated and a new frame has been created by the time `print_all(1)` gets to its return statement. The return value of `print_all(1)` is the function called in 3, which just happens to be the same function called for 1.

In an interactive session, the numbers 1, 3 and 5 would be printed, and the final return value would be the function location.

Why would be ever want to do this? Here's an example:

```
def print_sums(k):
    """Displays the cumulative sum of the numbers.
    """
    >>> print_sums(1)(3)(5)
    1
    4
    9
    """
    print(k)
    def next_sum(f):
        return print_sums(n+k)
    return next_sum
```

You might want to do this if you have a program with two parts, one which tracks how many times something is called, and one which has the logic of what to do with that info.

# Control

Here's another thing people frequently answer but don't know what's going on. We cannot just use call expressions for **everything**. There are certain cases, like if statements, that can never be perfectly replicated by a function.

Why can't we just create a function, `if(header expression, the if suite to be executed if true, the else suite)` ? This problem exists due to the evaluation rule for call expressions:

> Evaluate the operator and then **all** the operand subexpressions. That means both the True suite and the False suite are evaluated before the function is called.

Meanwhile, the rules for a statement are different. Only either the True or the False suite is evaluated, not both. It means that if you call this hypothetical if function on a value that errors on either the True or False value, the program would error before it knows which one to return properly.

# Control Expressions

Control doesn't have to be in a statement, it can also be an expression.

To evaluate the expression `<left> and <right>` :

- Evaluate the subexpression `<left>` .
- If the result is a false value `v` , then the expression evaluates to `v` .
- Otherwise, the expression evaluates to the value of the subexpression `<right>` .

Why do people use `and` functions? Well here's a scenario:

```
def has_big_sqrt(x):
    return x > 0 and sqrt(x) > 10
```

This way, the function doesn't evaluate `sqrt(x) > 10` if the function knows `x` is less than 0. Since the value on the left is false, the right is never evaluated because both need to be True for the function to return True.

To evaluate the expression `<left>` or `<right>` :

- Evaluate the subexpression `<left>` .
- If the result is a true value `v` , then the expression evaluates to value `v` .
- Otherwise, the expression evaluates to the value `<right>` .