Closures

Closures are what we use to wrap all of the necessary information for a function. They provide a self-contained way for us to use it later.

Let's say we have the function `f` defined as:

```
f x = x+1
```

In [1]:
```
def f(x: Int): Int = x+1
```
Out[1]:
```
defined function f
```

When we call this function with input, it works as expected.

In [2]:
```
f(2)
```
Out[2]:
```
res1: Int = 3
```

What if we had `g x = x+y` ?

In [2]:
```
def g(x: Int): Int = x+y
g(2)
```

```
cmd2.sc:1: not found: value y
def g(x: Int): Int = x+y
                       ^

Compilation Failed
```

Obviously this errors out. We don't know what the variable `y` is!

Bound vs Free Symbols

This brings us to the idea of "bound" and "free" symbols (variables).

A "bound" symbol is one that is self-contained in the expression. This means that we require no extra information. The `x` in `f x = x+1` is bound as it is defined in the expression itself.

A "free" symbol is a symbol that requires us to look somewhere else to find its meaning. The `y` in `g x = x+y` is a free symbol as we can't just look to `g` to get its definition. (Technically, `1` and `+` are also free symbols because they aren't defined in the expressions.)

An expression is considered "closed" if every symbol is bound. Otherwise, it is considered "open". Now wouldn't it be nice if we could "close" these expressions?

Closing an Open Expression

Wow! We can close the expression using a `closure`. The closure of a function includes the set of symbols already defined in the environment. This lets us give values to the free symbols in that expression.

Let's say we have the function `h x = x+y`. We might expect it to error out like `g` did but now we are first defining two variables above it, `y` and `z`.

```
In [3]: val y = 3
        val z = "i love pl!"
        def h(x: Int): Int = x+y
        h(2)
```

```
Out[3]: y: Int = 3
        z: String = "i love pl!"
        defined function h
        res2_3: Int = 5
```

It worked! This should make some sense to you. `y` and `z` were already defined within the scope of our `h` definition. This means that we can find a value for `y` without erroring out completely. This is done by creating an environment snapshot called π which is a direct copy of σ at the time of function definition. The environment snapshot for `h` would be something like:

```
{
    y: 3,
    z: "i love pl!",
    +: [language defined]
}
```

Notice how even though `z` isn't used in the function itself, it is still included in the environment snapshot. Also note how `+` is included because it's also not defined in `h`. This isn't *super* pertinent to us but it's still pretty cool.

Now what if we were to change the value of `y` after `h` is defined? What happens to the value within π? Well, in Lettuce, that value within π won't change. π is immutable and will never be updated for this closure. That means that if we were to have the cell:

```
val y = 3
def h(x: Int): Int = x+y
h(2)
val y = 39584372526398
h(2)
```

Both executions of `h(2)` should return `5`. We would not return `5` the first time then that insane number the second. You may be wondering why I didn't just put that into a code cell and let you run it yourself. It turns out that Scala doesn't follow the same rules that we define for Lettuce.

Just to be clear:

Even if the variable is updated after the function is defined, the original value from the environment snapshot is still used.

It should be `5`, not something crazy. If anyone tries to come up and say, "well Scala doesn't do it that way," I'm going to be very upset.

```
In [ ]:
```