Lab 8: Variable Resolution

Function Closure

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = "globals";
> def functO() { print(a); }
> { def a = "block"; functO(); }
globals
> functO();
globals
None
>
```

Function closure has been implemented such that the function will run using the environment it was originally defined inside of.

```
class NogginFunction(NogginCallable):
    def __init__(self, declaration:Function, closure:Environment):
        self.declaration = declaration
        self.closure = closure

    def call(self, interpreter, arguments:List):
        previous = interpreter.environment
        # Create a new environment for the function call
        interpreter.environment = Environment(self.closure)
        # Define all local variables
        for i in range(0, self.arity()):
            interpreter.environment.define(self.declaration.parameters[i],
arguments[i])
        try:
            for stmt in self.declaration.block.statements:
                interpreter.evaluate(stmt)
        except NogginReturn as r:
            interpreter.environment = previous
```

```
            return r.value
        finally:
            interpreter.environment = previous
        return None
```

By having the parent be the environment passed to us in the constructor we have function closure. If you change a variable within that same environment (usually applies in the global environment) this will affect the function as it does not copy the current environment at definition, but stores a reference to it.

Resolver

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = "globals";
> { def showA() { print(a); } showA(); def a = "block"; showA(); }
globals
globals
>
```

By counting how many environments we pass through to find the variable when resolving, we can assure that the proper variable will be accessed even if we create a new variable with the same name later in a deeper block.

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = "globals";
> { def showA() { print(a); } showA(); a = "block"; showA(); }
globals
block
>
```

If you change the value of a variable that the function is referencing this will still affect the final outcome as the resolver makes sure we access the same instance of a variable every time, not that the value stays the same forever.

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = "globals";
> def showA() { print(a); }
> showA();
globals
None
> def a = "change";
> showA();
change
None
>
```

Redefinition is allowed for all scopes in Noggin, but if I were to implement something like pylint or a warning system I would probably add a warning or optional squiggly lines when you redefine a variable within any scope, but this wouldn't make sense for interactive mode. Because of this, whether we are in a global or local scope Noggin will allow the redefinition and still access the variable "a" in that same environment mapped out by the resolver.

Resolution Errors

We do not allow self-initialization, so running this code will result in an error thrown by the resolver.

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = 10;
> { def a = a; }
[line 0] Error at 'a': Cannot read local variable in its own initializer.
> { a = a; }
> { def a = 200; a = a; }
>
```

Using a variable within its own definition is okay as long as it is an assignment and not an initialization. This makes it so that we are only ever referring to one same-name variable within the same line of code. Allowing self-initialization as in line 2 would mean the first a and the second a would be two different a's and this is inherently confusing. In lines 3 and 4 they refer to the same a, regardless of whether it is referring to a local scope or a global scope.