Lab6: Control Flow

If and while statements:

```
>>>>> PNoggin Interactive Shell <<<<<
> if(10 == 5) { print("failure"); }
> if(10-10 == 0 or 5/0 == 10) { print("success"); }
success
> if(false){ print("You Lose!"); } else { print("Victory!"); }
Victory!
> if(false == true) { print(3+5); } else if (10+10==20) { print(5); } else { print(10); }
5.0
> if(false == true) { print(3+5); } else if (100==20) { print(5); } else { print(10); }
10.0
```

- Successfully interprets condition and skips the print for 10 == 5
- Successfully short circuits and does not run 5 / 0 instead printing success
- Successfully prints Victory! within the else statement as the condition was false
- Successfully prints 5 using Noggin's else if use case
- Successfully prints final print in the final else after all if, else if, statements

```
def ifStatement(self):
        self.consume(TokenType.LEFT_PAREN, "Expect '(' after if token")
        condition = self.expression()
        self.consume(TokenType.RIGHT_PAREN, "Expect ')' after if
condition")
        self.consume(TokenType.LEFT_BRACE, "Expect '{' before if statement
contents")
        block = self.block()
        if self.match(TokenType.ELSE):
            if self.match(TokenType.IF):
                return If(condition, block, self.ifStatement())
            self.consume(TokenType.LEFT_BRACE, "Expect '{' before else
statement contents")
            else_block = self.block()
            return If(condition, block, Else(else_block))
        return If(condition, block)
```

The if statement method parses an infinite number of Else If statements, keeping the else and if as two separate tokens rather than implementing Elif like python.

My language does not require a solution for the single line variable declaration within an if statement as it requires that all if, while, for, else, etc. statements use blocks. All if statements will look like: if (expression) block

And then potentially have the following any number of else if statements and potentially one final else statement.

This also solves the dangling else problem because brackets prevent ambiguity in the way described in the lecture.

Implement for statement by desugaring

For loops get interpreted with each section ending with a semicolon, and the definition section being optional. The modifier section can be pretty much any single statement that the while loop will run at the end.

```
def forStatement(self):
        self.consume(TokenType.LEFT_PAREN, "Expect '(' after for token")
        definition = None
        if self.match(TokenType.DEF):
            definition = self.varDeclaration()
        condition = self.expression()
        self.consume(TokenType.SEMICOLON, "Expect ';' after loop
condition")
        modifier = self.statement()
        self.consume(TokenType.RIGHT_PAREN, "Expect ')' after for
expression")
        self.consume(TokenType.LEFT_BRACE, "Expect '{' before for
statement contents")
        block = self.block()
        block.statements.append(modifier)
        if definition:
            return Block([definition, # Define
                          While(condition, block)])
        return Block([While(condition, block)])
```

After parsing through the syntax and storing the definition, condition, modifier, and block we then add the modifier to the end of the block. Then we return a block with 1 or 2 elements: Definition (optional), and the while loop. The While function can now receive the condition, and the pre-modified block.

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = 0;
> def temp;
>
> for (def b = 1; a < 10000; b = temp + b;){ print(a); temp = a; a = b; }
0.0
1.0
1.0
2.0
3.0
5.0
8.0
13.0
21.0
34.0
55.0
89.0
144.0
233.0
377.0
610.0
987.0
1597.0
2584.0
4181.0
6765.0
>
```

This code from the examples successfully runs, and is running as a while loop in the backend. The code looks like this in the backend after desugared:

```
def a = 0;
def temp;
{
        def b = 1;
        while( a < 10000; )
        {
                print(a);
                temp = a;
                a = b;
                b = temp + b;
        }
}
```

Syntax Errors

All syntax errors for each new control loop are handled by using the pre-existing method consume. If we expect something 100% certain we can call consume and it will either advance us, or throw the proper error using the custom string we give it when we call it. This is used for every parentheses and bracket for the for, if, else, and while loops.

Extra Credit

If anything that I did might count for extra credit it would be the way I implemented chaining if, else if, else statements such that else if is functionally included in the syntactical grammar and we can have an infinite number of checks that go in order and the else if's short circuit if a previous if is triggered. Might be a long shot, but it does go a little more in depth than the lox implementation.