Lab 5: Variables

New Syntactic Grammar:

```
program       → declaration* EOF ;

declaration   → varDecl
              | statement ;

varDecl       → "def" IDENTIFIER ( "=" expression )? ";" ;

statement     → printStmt
              | block
              | expressionStmt ;

printStmt     → "print" "(" expression ( "," expression )* ")" ";" ;

block         → "{" declaration* "}" ;

expressionStmt → expression ";" ;

expression    → assignment ;

assignment    → IDENTIFIER "=" assignment
              | equality ;

equality      → comparison ( ( "==" | "!=" ) comparison )* ;

comparison    → term ( ( ">" | ">=" | "<" | "<=" ) term )* ;

term          → factor ( ( "+" | "-" ) factor )* ;

factor        → unary ( ( "*" | "/" ) unary )* ;

unary         → ( "!" | "-" ) unary
              | primary ;

primary       → NUMBER
              | STRING
              | "true"
              | "false"
              | "null"
              | IDENTIFIER
              | "(" expression ")" ;
```

Print Functionality:

```
>>>>> PNoggin Interactive Shell <<<<<
> print(100+200, "100,000", "hi hello!");
300.0 100,000 hi hello!
> print("yeah");
yeah
> print("no";
[line 0] Error at ';': Expect ')' before ';'
> print(error);
```

Print is a function in Noggin, and such it looks closer to python adding a semicolon. I also have multiple statements split by commas functionality for my print function.

Variable Declaration, Assignment, and Access Functionality:

```
>>>>> PNoggin Interactive Shell <<<<<
> a=10;
Error: Variable a has not been declared
[line 0]
> print(a);
Error: Variable a has no value (no variable declaration)
[line 0]
> def a=10;
> print(a);
10.0
> a=20;
> print(a);
20.0
>
```

Declaration requires def for initialization. Assignment can occur after a variable has been declared. This code shows that assignment fails before declaration, print fails if variable has not been declared, and declaration, access, and assignment all work when you follow the proper order. The value of ‘a’ changes when assignment is complete.

Block Scope and Environments Functionality:

```
>>>>> PNoggin Interactive Shell <<<<<
> def a = "global a";def b = "global b";def c = "global c";{def a = "outer a";def b = "outer b";{def a = "inner
 a";print(a);print(b);print(c);}print(a);print(b);print(c);}print(a);print(b);print(c);

inner a
outer b
global c
outer a
outer b
global c
global a
global b
global c
>
```

Because we put ‘def’ before every variable assignment they are all declarations which means variables will shadow other variables in the deeper scopes. In the deepest scope a prints from the innermost block’s variable, b from the outer, and c the global.

Variables pull from their most local scope first, then travelling up the chain. This goes on and as we exit each block/scope we see that outer a is intact, and that global a and global b are also intact. After exiting the outer and inner scopes, the outer and global variables did not change.

Error Catching Functionality:

Using the consume function during parsing, and raising a NogginRuntimeError during interpretation, Noggin catches a wide variety of errors, for brevity I will stick to some of the broader required errors we covered.

Missing Semicolons:

At the end of each statement we run the code

```
self.consume(TokenType.SEMICOLON, "Expect ';' after *(insert specific function name here)* .")
```

This allows us to report an error if there is no semicolon detected, and report why we expect it.

```
>>>>> PNoggin Interactive Shell <<<<<
> def a=10
[line 0] Error at end: Expect ';' after variable declaration
> print(100)
[line 0] Error at end: Expect ';' after print function.
> def a=100;
> a=200
[line 0] Error at end: Expect ';' after variable assignment
>
```

Unmatched Braces:

Each block will continue a while loop where it will keep parsing either until it matches a Right Brace, or until it reaches the end of the code. This will ensure that we report all unmatched braces under any circumstances. Once we break out of the while loop we attempt to consume the right brace, if we broke out because we are at the end it will report an error.

```
self.consume(TokenType.RIGHT_BRACE, "Expect '}' at the end of block")
```

```
>>>>> PNoggin Interactive Shell <<<<<
> { print("hello");
[line 0] Error at end: Expect '}' at the end of block
> def a=10; { def b=20; print(b); {
[line 0] Error at end: Expect '}' at the end of block
[line 0] Error at end: Expect '}' at the end of block
>
```

Synchronization and Cascading Errors Functionality:

```
>>>>> PNoggin Interactive Shell <<<<<
> def a=10 def b=10; print(10+2=); "hello"; print("we made it to the end");
[line 0] Error at 'def': Expect ';' after variable declaration
[line 0] Error at '=': Expect ',' in between expressions
we made it to the end
>
```

Noggin will detect errors and continue parsing from approved places, and if possible run what gets successfully parsed. After missing the semicolon in the declaration, it synchronizes at the next def token, and when the print function has an unexpected = it synchronizes at the next print token. Since the print function parses in its entirety this code is still able to run and print out the statement for us.

Extra Credit Functionality:

Extended Interactive Mode Functionality - Expressions:

```
>>>>> PNoggin Interactive Shell <<<<<
> 3+3
6.0
> 5-10==4
false
> print(3+3);
6.0
> print(4+2);
6.0
>
```

Noggin can both parse expressions in place without semicolons, or it will parse for things like print. When there is a single expression detected and nothing else it will ignore the missing semicolon, evaluate the expression, and stringify and print it. This works in tandem with the other functionalities, and is intended for interactive mode use but should work in a file as well.

Extended Print Functionality - Consecutive Inputs:

```
>>>>> PNoggin Interactive Shell <<<<<
> print("Hello", "I", "Love", "Pizza", 105, 10==2, null, "all done!");
Hello I Love Pizza 105.0 false null all done!
>
```

Similarly to the behavior of the expression parsing matching an operator, print will continue parsing additional expressions if it matches a comma after the expression instead of always expecting a parentheses. This allows us to combine all inputs into a list in the Print object. Then when we go to visit_print, it will evaluate them one element at a time adding a space between them, and then print all at once.