Context

Prepared by Rene Andre B. Jocsing October 17, 2025

Congratulations on making it to the fourth lab! Your interpreter can now evaluate expressions, but so far it's been like a very expensive calculator. Real programming languages need more than just arithmetic—they need **statements** and **state**. This is where your language transforms from a toy into something actually useful.

Up until now, everything you've evaluated has been ephemeral. You type an expression, it computes a value, you see the result, and poof—it's gone. But real programs need to remember things. They need variables that persist across multiple statements. They need to modify those variables. They need to execute sequences of operations, not just single expressions.

This lab marks a fundamental shift in how you think about your language. You're moving from the world of pure expressions (which always evaluate to values) into the world of statements (which produce effects). Buckle up!

Expressions vs. Statements

Let's clarify the distinction between these two fundamental concepts, because you'll be working with both from now on.

An **expression** is a piece of code that evaluates to a value. Everything you've implemented so far has been an expression. `2 + 3` is an expression. `"hello" + " world"` is an expression. `!(x > 5)` is an expression. When you evaluate an expression, you get a value back.

A **statement**, on the other hand, is a piece of code that produces an effect but doesn't necessarily evaluate to a value. Statements are about *doing* things rather than computing things. Think of statements as the verbs of your programming language—they make things happen.

In most C-family languages (and you'll likely follow this pattern), you turn an expression into a statement by sticking a semicolon at the end. So `2 + 3;` is an expression statement. It evaluates the expression `2 + 3`, but then discards the result and moves on. Not very useful on its own, but paired with side effects (like function calls with side effects or variable assignments), expression statements become powerful.

Other types of statements include:

- **Print statements**: Execute an expression and display the result
- **Variable declarations**: Create new variables and optionally initialize them
- **Blocks**: Group multiple statements together between curly braces
- **Control flow statements**: We'll tackle these in the next and final lab (if, while, for etc.)

Print Statements

Before we dive into variables, let's implement something simple: a print statement. Yes, you've been seeing output in your REPL, but that's just the REPL automatically printing expression results. A real language needs an explicit way to output values.

The syntax might look like this (adjust accdg. to your language design):

```
print "Hello, world!";
print 2 + 3;
print x + y;
```

When the interpreter encounters a print statement, it should evaluate the expression and display the resulting value. This gives programmers explicit control over output, which is essential once you move beyond single-line REPL interactions to executing entire programs.

Your grammar needs a new production rule. Something like:

```
statement      → exprStmt | printStmt ;
printStmt      → "print" expression ";" ;
exprStmt       → expression ";" ;
```

Notice how statements don't evaluate to values—they just execute. Your evaluator needs to handle this differently than expressions. Instead of returning values, statement execution might return nothing (a null value or it might return some result, depending on your language design).

Variables and Environments

Now for the main event: variables. Variables are how programs remember things. They associate names with values, and those associations need to persist across multiple statements.

Consider this sequence:

```
var x = 10;
var y = 20;
print x + y;
```

The first two lines create variables. The third line needs to look up those variables by name and retrieve their values. This is where **environments** or **contexts** come in.

An environment or context (sometimes called a symbol table) is a data structure that maps variable names to their values. Every time you declare a variable, you add an entry to the context. Every time you reference a variable, you look it up in the context.

The simplest implementation is a hash map (dictionary, associative array, or whatever you call it) where keys are variable names (strings) and values are the runtime values. When you declare `var x = 10;`, you add `"x" -> 10` to the environment. When you later encounter the identifier `x`, you look up `"x"` in the environment and retrieve `10`.

Variable Declaration

Variable declaration creates a new variable and optionally gives it an initial value. The syntax typically looks something like:

```
var identifier;
var identifier = initializer;
```

You'll need to extend your grammar to handle this. The declaration statement should parse an identifier name and an optional initializer expression. If there's no initializer, the variable starts with some default value (commonly `null` or `nil`).

Your grammar might look like:

```
statement      → exprStmt | printStmt | varDecl ;
varDecl        → "var" IDENTIFIER ( "=" expression )? ";" ;
```

When executing a variable declaration:

1. If there's an initializer, evaluate it to get the initial value
2. If there's no initializer, use your default value (nil, null, undefined, etc.)
3. Add the variable name and value to your environment
4. Check if the variable already exists—some languages allow redeclaration, others don't

Variable Access and Assignment

Once variables exist, you need two operations: reading them and writing to them.

Variable access is reading the value. When your parser encounters an identifier in an expression context, it's probably a variable reference. During evaluation, look up the identifier's name in your environment and return the associated value. If the variable doesn't exist, that's a runtime error.

Your grammar already has identifiers as a kind of primary expression (from your previous lab), so this might just work with a small modification to your evaluator.

Assignment is writing a new value to an existing variable. The syntax is typically:

```
identifier = expression;
```

Assignment is interesting because it's both an expression and a statement. As an expression, it evaluates to the assigned value (which is why you can chain assignments: `a = b = 5;`). As a statement, it produces the side effect of updating the variable.

Your grammar might add:

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment | equality ;
```

Notice the right-associativity of assignment—`a = b = 5` should parse as `a = (b = 5)`.

When evaluating an assignment:

1. Evaluate the right-hand side to get the new value
2. Look up the variable in the environment
3. If it doesn't exist, that's a runtime error (in most languages, you can't assign to undeclared variables)
4. Update the variable's value in the environment
5. Return the assigned value (for expression contexts)

Blocks and Scope

With variables come questions of **scope**: where is a variable visible? Most modern languages use **lexical scoping** (also called static scoping), where a variable's scope is determined by where it appears in the source code.

A block is a series of statements wrapped in curly braces. Blocks create nested scopes:

```
1 var x = "outer";
2 {
3     var x = "inner";
4     print x;  // prints "inner"
5 }
6 print x;  // prints "outer"
```

The inner `x` *shadows* the outer `x`. They're different variables that happen to have the same name. When the block ends, the inner `x` goes out of scope, and references to `x` again refer to the outer one.

To implement this, you need nested contexts. When entering a block, create a new environment that points to the current environment as its parent. When looking up a variable, check the current environment first; if it's not there, check the parent, then the grandparent, and so on up the chain.

When exiting a block, discard the current environment and restore the parent as the current environment. This automatically handles variable lifetime—variables declared in a block exist only for that block's duration.

Your grammar adds:

```
statement      → exprStmt | printStmt | varDecl | block ;
block          → "{" declaration* "}" ;
declaration    → varDecl | statement ;
```

Notice the new `declaration` rule. This is how you structure your grammar to allow variable declarations anywhere you can have statements.

Implementing the Environment

Your environment needs to support a few operations:

- **define(name, value)**: Create a new variable in the current scope
- **get(name)**: Look up a variable, checking parent scopes if necessary
- **assign(name, value)**: Update an existing variable, checking parent scopes if necessary

The key difference between `define` and `assign` is that `define` creates a new variable in the current scope (even if one with the same name exists in an outer scope), while `assign` modifies an existing variable (and throws an error if the variable doesn't exist).

For nested scopes, each environment should have a reference to its enclosing environment. When you can't find a variable in the current environment, recursively check the enclosing one. If you reach the outermost environment and still haven't found the variable, throw a runtime error.

REPL vs. Script Mode

Your REPL has been evaluating single expressions and printing the results. But now that you have statements, you need to adjust this behavior. When running a script (a file containing a program), you probably don't want to print the value of every expression statement—that would be noisy and confusing.

The typical solution is to have two modes:

- **REPL mode**: Evaluate expressions and automatically print their values (the current behavior)
- **Script mode**: Execute statements without automatically printing anything (use explicit print statements for output)

You'll need to support both. Your REPL should still be convenient for quick experimentation, but now it should also be able to execute entire programs from source code in external files.

Laboratory Deliverables

Grammar Extension

Update your language's grammar to include statements. Add production rules for:

- Expression statements
- Print statements
- Variable declarations
- Variable assignment
- Blocks and nested scopes

Document these changes in your README.md under the Grammar section. Make sure your grammar remains unambiguous and correctly handles precedence.

Statement Parser

Extend your parser to handle statements in addition to expressions. Your parser should:

- Parse all the statement types listed above
- Build appropriate AST nodes for statements
- Handle the distinction between expressions and statements
- Support blocks with multiple nested statements
- Provide clear error messages for malformed statements

Environment Implementation

Implement an environment class (or equivalent structure) that:

- Stores variable name-to-value mappings
- Supports nested scopes with context chaining
- Provides define, get, and assign operations
- Throws appropriate errors for undefined variables
- Handles variable shadowing correctly

Statement Evaluator

Extend your evaluator to execute statements. Your implementation should:

- Execute expression statements (evaluate and discard result)
- Execute print statements (evaluate and output result)
- Execute variable declarations (add to environment)
- Evaluate variable references (look up in environment)
- Evaluate assignments (update environment)
- Handle blocks (create and destroy nested scopes)
- Report runtime errors for undefined variables

Script Execution

Modify your main program to support running scripts from files, not just the REPL. Users should be able to:

- Run your interpreter in REPL mode (current behavior)
- Pass a file as an argument to execute as a script
- See appropriate error messages

Testing

Create comprehensive test cases that demonstrate:

- Variable declaration and initialization
- Variable access and assignment
- Shadowing in nested blocks
- Print statements
- Expression statements
- Undefined variable errors
- Assignment to undefined variables
- Multiple statements in sequence

Write test scripts (text files containing programs in your language) and demonstrate that your interpreter executes them correctly. You'll present these during your defense.

As always, commit your changes to your GitHub repository with meaningful messages. You're building on the same codebase from previous labs, so keep your code clean and well-documented!

Expected Output

```
> var x = 10;
> print x;
10
> x = 20;
> print x;
20
> var y = x + 5;
> print y;
25
> var name = "Alice";
> print name;
Alice
> print "Hello, " + name;
Hello, Alice
> {
>     var x = "inner";
>     print x;
> }
inner
> print x;
20
> {
>     var a = 1;
>     {
>         var b = 2;
>         print a + b;
>     }
> }
3
> print a;
[line 1] Runtime error: Undefined variable 'a'.
> var z;
> print z;
nil
> z = 100;
> print z;
100
> undeclared = 5;
[line 1] Runtime error: Undefined variable 'undeclared'.
> var result = (x = 15);
> print result;
15
> print x;
15
```

Example script file (test.txt):

```
// A simple program demonstrating variables and scope
var greeting = "Hello";
var name = "World";

print greeting + ", " + name + "!";

var x = 10;
var y = 20;

{
    var x = 5;  // shadows outer x
    print "Inner x: " + x;
    print "Outer y: " + y;
}

print "Outer x: " + x;

// Reassignment
x = x + y;
print "x + y = " + x;
```

Running the built executable on a script:

```
$ java -jar language.jar test.txt
Hello, World!
Inner x: 5
Outer y: 20
Outer x: 10
x + y = 30
```

Rubric for Programming Exercises

Table 1: Rubric for Programming Exercises (50 pts)

Criteria (10 Points Each)	Excellent (9 – 10)	Good (6 – 8)	Fair (3 – 5)	Poor (0 – 2)
Program Correctness	Program executes correctly with no syntax or runtime errors, meets/exceeds specifications, and displays correct output	Program executes and outputs with minor errors, yet meets specifications	Program executes and outputs with major errors, yet somehow meets specifications	Program does not execute or does not meet specs
Logical Design	Program is logically well-designed with excellent structure and flow	Program has slight logic errors that do not significantly affect the results	Program has significant logic errors affecting functionality	Program logic is fundamentally incorrect
Code Mastery	Programmer demonstrates excellent mastery over the program's code	Programmer demonstrates adequate mastery over the program's code	Programmer demonstrates fair mastery over the program's code	Programmer demonstrates poor mastery over the program's code
Engineering Standards	Program is stylistically well designed from an engineering standpoint	Slight inappropriate design choices (i.e., poor variable names, improper indentation)	Severe inappropriate design choices (i.e., code repetition, redundancy)	Program is poorly written
Documentation*	Program is well-documented: comments exist for clarity, not redundancy	Missing one required comment or some redundant comments	Missing two or more required comments or many redundant comments	Most documentation missing or most documentation is redundant

Remember:* *"Code tells you how, comments tell you why."*** *— Jeff Atwood, co-founder of Stack Overflow and Discourse*