October 28, 2025

Context

Rene Andre Bedonia Jocsing

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:** (if, while, for, etc.) — handled in the next lab

---

Print Statements

Before we dive into variables, let's implement something simple: a print statement.

Example syntax:

```
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.

Your grammar needs a new production rule:

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

Variables and Environments

Variables are how programs remember things. They associate names with values that persist across statements.

Example:

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

An **environment** (or **context**) is a data structure mapping variable names to values. Typically a hash map.

Variable Declaration

Variable declaration creates a new variable and optionally gives it an initial value.

Example:

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

Grammar:

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

Execution steps:

1. Evaluate initializer (if present)
2. Use default value if none (e.g., `nil`)
3. Add variable to environment
4. Check for redeclaration (depending on language design)

Variable Access and Assignment

Access (read): lookup variable name in environment.
Assignment (write): update variable value.

Example:

```
identifier = expression;
```

Grammar:

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

Evaluation steps:

1. Evaluate right-hand side.
2. Lookup variable.
3. Error if undefined.
4. Update environment.

5. Return assigned value.

Blocks and Scope

Blocks define **nested scopes**:

```
var x = "outer";
{
    var x = "inner";
    print x;  // inner
}
print x;      // outer
```

When entering a block, create a new environment that references its parent. When exiting, discard it.

Grammar:

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

Implementing the Environment

Required operations:

- **define(name, value):** Create new variable in current scope
- **get(name):** Lookup variable
- **assign(name, value):** Update existing variable

Nested scopes require an enclosing reference to the parent environment.

REPL vs. Script Mode

Two modes:

- **REPL mode:** Automatically print evaluated expressions
- **Script mode:** Execute statements silently unless explicitly printed

---

Laboratory Deliverables

Grammar Extension

Add grammar rules for:

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

Statement Parser

Extend your parser to:

- Parse all statement types
- Build AST nodes
- Handle expressions vs. statements
- Support nested blocks

- Provide clear error messages

Environment Implementation

Implement an environment that:

- Stores variable name-value pairs
- Supports nested scopes
- Provides define, get, assign
- Throws errors for undefined vars
- Handles shadowing correctly

Statement Evaluator

Evaluator must:

- Execute expression, print, var, and block statements
- Handle assignments and variable references
- Report runtime errors

Script Execution

Add script execution capability:

- Run interpreter with REPL or file input
- Show appropriate errors

Testing

Test cases should cover:

- Declaration, access, assignment
- Shadowing
- Print and expression statements

- Undefined variable handling
- Multiple statements

Commit all changes with meaningful messages.

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 (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

---

Criteria (10 Points Each)	Excellent (9-10)	Good (6-8)	Fair (3-5)	Poor (0-2)
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

---

TAGS
CMSC-124

← Back to home