FUNCTIONS

Prepared by Rene Andre B. Jocsing October 19, 2025

Welcome to the last leg of your language hacking journey. You've built a scanner, parser, and evaluator, then added scoping to your interpreter. Now it's time to add the features that will transform your interpreter from a sophisticated calculator into a real programming language: **control flow** and **functions**.

Control flow lets your programs make decisions and repeat operations. Functions let you package reusable chunks of code. Together, these features give your language the power to solve real problems, taking one step closer to Turing-completeness. A programming language without these constructs is like a car without a steering wheel—it can move, but you can't really go anywhere interesting.

This final lab represents the culmination of everything you've learned. By the end, you'll have implemented a complete, Turing-complete programming language. That's no small feat!

If-Else

Every useful program needs to make decisions. Should we execute this code or that code? Should we repeat this operation or move on? This is what control flow constructs enable.

The most fundamental control flow construct is the **if statement**. It evaluates a condition and executes different code depending on whether the condition is true or false:

```
1 if (temperature > 30) {
2     print "It's hot!";
3 } else {
4     print "It's comfortable.";
5 }
```

The condition (the expression in parentheses) gets evaluated to a boolean value using your truthiness rules from Lab 3. If it's truthy, execute the first branch. If it's falsey, execute the else branch (if one exists).

Your grammar might look like:

```
statement      → exprStmt | printStmt | varDecl | block | ifStmt ;
ifStmt         → "if" "(" expression ")" statement ( "else" statement )? ;
```

Notice that the branches are statements, not just blocks. This allows single-statement branches without braces, though many style guides discourage this. It also enables **cascading if-else chains**. Like so.

```
if (score >= 90) {
    grade = "A";
} else if (score >= 80) {
    grade = "B";
} else if (score >= 70) {
    grade = "C";
} else {
    grade = "F";
}
```

This works because the `else if` is really an `else` containing another `if` statement.

The Dangling Else Problem

If you allow if statements without braces, you'll encounter a classic parsing ambiguity. Consider:

```
if (first)
    if (second)
        print "both";
    else
        print "???";
```

Which `if` does the `else` belong to? The grammar is ambiguous—it allows two different parse trees. Most languages resolve this by having the `else` bind to the nearest `if`. Your parser should naturally do this if you implement the grammar correctly, but be aware of the issue.

Loops

The second pillar of control flow is repetition. While you could achieve repetition with recursion and if statements (and some languages do!), loops are more intuitive for most programmers.

The **while loop** is the simplest. It repeatedly executes a statement as long as a condition remains true:

```
var i = 0;
while (i < 10) {
    print i;
    i = i + 1;
}
```

Your grammar may add:

```
statement      → ... | whileStmt ;
whileStmt      → "while" "(" expression ")" statement ;
```

The implementation is straightforward: evaluate the condition, and if truthy, execute the body statement. Then loop back and check the condition again. Repeat until the condition becomes falsey.

Most languages also include a **for loop**. The for loop is really just syntactic sugar—a more convenient way to write a common pattern. The classic C-style for loop has three parts: an initializer, a condition, and an increment:

```
for (var i = 0; i < 10; i = i + 1) {
    print i;
}
```

This is equivalent to:

```
{
    var i = 0;
    while (i < 10) {
        print i;
        i = i + 1;
    }
}
```

Your grammar might be:

```
statement      → ... | forStmt ;
forStmt        → "for" "(" ( varDecl | exprStmt | ";" )
                 expression? ";"
                 expression? ")" statement ;
```

The three parts are all optional, which is why you see the `?` markers. An infinite loop would be `for (;;) { ... }`.

When executing a for loop, you can desugar it into the while loop equivalent during parsing or evaluation. This keeps your implementation simpler—you don't need separate logic for for loops.

Logical Operators

With control flow, you'll often need to combine conditions. **Logical operators** `and` and `or` (or `&&` and `||` in C-style syntax) fill this role:

```
if (temperature > 30 and humidity > 80) {
    print "It's hot AND humid!";
}
```

These operators are special because they **short-circuit**. The `and` operator only evaluates its right operand if the left operand is true. The `or` operator only evaluates its right operand if the left operand is false. This isn't just an optimization—it affects program semantics.

Consider:

```
if (denominator != 0 and numerator / denominator > 1) {
    print "Ratio is greater than 1";
}
```

If `denominator` is zero, the right side never evaluates, avoiding a division-by-zero error. Without short-circuiting, this code would crash.

Your grammar needs to place these operators at the right precedence level. For example:

```
expression     → assignment ;
assignment     → IDENTIFIER "=" assignment | logicOr ;
logicOr        → logicAnd ( "or" logicAnd )* ;
logicAnd       → equality ( "and" equality )* ;
```

The `or` operator has lower precedence than `and`, matching how we think about them in natural language.

Functions

Functions are one of the most important abstractions in programming. They let you:

- Package code for reuse
- Give meaningful names to operations
- Hide implementation details
- Reduce code duplication
- Build programs compositionally

A function declaration might look like:

```
fun greet(name) {
    print "Hello, " + name + "!";
}
```

And you call it like:

```
greet("Alice");
greet("Bob");
```

Your grammar may extend to include function declarations and calls:

```
declaration    → funDecl | varDecl | statement ;
funDecl        → "fun" IDENTIFIER "(" parameters? ")" block ;
parameters     → IDENTIFIER ( "," IDENTIFIER )* ;

primary        → ... | IDENTIFIER | "(" expression ")" ;
call           → primary ( "(" arguments? ")" )* ;
arguments      → expression ( "," expression )* ;
```

Notice that calls are postfix operators on expressions. You can call any expression that evaluates to a callable value: `getFunction()()`.

Function Objects and Closures

When you declare a function, what actually happens? You need to store information about the function so you can execute it later when it's called. Most implementations create a **function object** that contains:

- The function's parameter names
- The function body (as an AST node)
- The environment where the function was declared (for closures)

This function object gets stored in your environment just like any other value. The function's name is a variable that holds a reference to the function object.

The environment capture is crucial for **closures**—functions that "close over" and remember variables from their enclosing scope. Do closures exist in your language? If so, this is where you implement them.

```
1  fun makeCounter() {
2      var count = 0;
3      fun increment() {
4          count = count + 1;
5          return count;
6      }
7      return increment;
8  }
9
10 var counter = makeCounter();
11 print counter();  // 1
12 print counter();  // 2
13 print counter();  // 3
```

The `increment` function remembers the `count` variable even after `makeCounter` has returned. This is only possible if the function stores a reference to the environment where it was created.

Calling Functions

When a function is called, you need to:

1. Evaluate all argument expressions
2. Create a new environment for the function execution
3. The new environment's parent should be the function's closure environment (where it was declared), not the current environment (where it's being called)
4. Bind parameter names to argument values in the new environment
5. Execute the function body in this new environment
6. Return the result (if any)

This process is called **dynamic dispatch**—deciding at runtime which code to execute based on what function object you're calling.

Return Statements

Functions need a way to send values back to their callers. The `return` statement does this:

```
fun add(a, b) {
    return a + b;
}

var result = add(3, 5);
print result;  // 8
```

Your grammar adds:

```
statement      → ... | returnStmt ;
returnStmt     → "return" expression? ";" ;
```

The expression is optional—`return;` with no value is valid. This implicitly returns `nil` (or your language's equivalent).

Implementing return is tricky because it needs to unwind the call stack, potentially exiting through multiple nested statements and blocks. Many implementations use exceptions for control flow here. When you execute a return statement, you can trivially throw a special "return exception" that carries the return value. The function call catches this exception and uses the value as the function's result.

Native Functions

Before users can write functions, your interpreter needs at least one function they can call—otherwise there's no way to get the ball rolling. Most languages provide **native functions** (also called built-in or primitive functions) implemented in the host language.

For example, you might want a `clock()` function that returns the current time, or a `readLine()` function for user input. These can't be implemented in your language because they need to access host language capabilities.

Implement native functions as callable objects that execute host language code instead of interpreting AST nodes. Add them to your global environment before executing any user code.

Arity Checking

Functions have a specific number of parameters (their **arity**). Most languages require that calls provide exactly the right number of arguments:

```
fun add(a, b) {
    return a + b;
}

add(1, 2);     // OK
add(1);        // Error: Expected 2 arguments but got 1
add(1, 2, 3);  // Error: Expected 2 arguments but got 3
```

Check arity when calling functions and report clear errors for mismatches. This catches bugs early.

Some languages allow default parameters or variadic functions (accepting any number of arguments). These are optional enhancements if you want to implement them.

Recursion

One beautiful consequence of your implementation is that recursion should "just work." Since function names are variables in the environment, and that environment is accessible during function execution, a function can call itself:

```
fun fibonacci(n) {
    if (n <= 1) return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

print fibonacci(10);  // 55
```

Make sure your implementation handles this correctly. Test with recursive functions, including mutually recursive functions (where function A calls function B, which calls function A).

Laboratory Deliverables

Grammar Extension

Update your language's grammar to include control flow and functions. Add production rules for:

- If statements (with optional else)
- While loops
- For loops
- Logical operators (and, or)
- Function declarations
- Function calls
- Return statements

Document these in your README.md under the Grammar section. Your complete grammar should now describe a Turing-complete programming language.

Control Flow Implementation

Extend your parser and evaluator to support:

- If-else statements with correct precedence
- While loops
- For loops (can be desugared to while loops)
- Short-circuiting logical operators
- Proper scoping for loop variables

Function Implementation

Implement functions with:

- Function declaration statements
- Function call expressions
- Parameter binding

- Return statements (including early returns)
- Closures (capturing enclosing environment)
- Proper environment management for calls
- Arity checking with clear error messages

Native Functions

Add at least two native functions to your language. Suggested examples:

- `clock()`: Returns current time in seconds
- `print(value)`: Prints a value (if not already a statement)
- `readLine()`: Reads user input
- `toString(value)`: Converts value to string

Comprehensive Testing

Create test programs demonstrating:

- Conditional execution (if-else)
- Loops (while and for)
- Nested control flow
- Function declaration and calling
- Recursive functions
- Closures
- Higher-order functions (functions that take or return functions)
- Error handling (wrong arity, etc.)

You should have several script files showcasing your language's capabilities. These will be demonstrated during your final defense.

Language Showcase

Write at least one non-trivial program in your language that demonstrates its capabilities. Add this as an example program in your repository (perhaps in an `examples/` directory).

As always, commit your changes with meaningful messages. This is the final major addition to your interpreter—you should be proud of what you've built!

Expected Output

Control Flow:

```
> var x = 5;
> if (x > 3) print "x is big";
x is big
> if (x > 10) {
>     print "x is huge";
> } else {
>     print "x is not that big";
> }
x is not that big
> var i = 0;
> while (i < 3) {
>     print i;
>     i = i + 1;
> }
0
1
2
> for (var j = 0; j < 3; j = j + 1) {
>     print j;
> }
0
1
2
> if (true or false) print "yep";
yep
> if (false and true) print "nope"; else print "correct";
correct
```

Functions:

```
> fun greet(name) {
>     print "Hello, " + name + "!";
> }
> greet("Alice");
Hello, Alice!
> fun add(a, b) {
>     return a + b;
> }
> print add(3, 5);
8
> fun fibonacci(n) {
>     if (n <= 1) return n;
>     return fibonacci(n - 1) + fibonacci(n - 2);
> }
> print fibonacci(10);
55
> fun makeCounter() {
>     var count = 0;
>     fun increment() {
>         count = count + 1;
>         return count;
>     }
>     return increment;
> }
> var counter = makeCounter();
> print counter();
1
> print counter();
2
> print counter();
3
> greet();
[line 1] Runtime error: Expected 1 arguments but got 0.
> add(1, 2, 3);
[line 1] Runtime error: Expected 2 arguments but got 3.
```

Example script file (fibonacci.txt):

```
// Recursive Fibonacci implementation
fun fibonacci(n) {
    if (n <= 1) {
        return n;
    }
    return fibonacci(n - 1) + fibonacci(n - 2);
}

// Print first 15 Fibonacci numbers
for (var i = 0; i < 15; i = i + 1) {
    print "fib(" + i + ") = " + fibonacci(i);
}
```

Output:

```
$ java -jar language.jar fibonacci.txt
fib(0) = 0
fib(1) = 1
fib(2) = 1
fib(3) = 2
fib(4) = 3
fib(5) = 5
fib(6) = 8
fib(7) = 13
fib(8) = 21
fib(9) = 34
fib(10) = 55
fib(11) = 89
fib(12) = 144
fib(13) = 233
fib(14) = 377
```

Example script file (closures.txt):

```
// Demonstrate closures
fun makePrinter(prefix) {
    fun print(message) {
        print prefix + ": " + message;
    }
    return print;
}

var errorPrinter = makePrinter("ERROR");
var infoPrinter = makePrinter("INFO");

errorPrinter("Something went wrong");
infoPrinter("Program started");
errorPrinter("Connection failed");
infoPrinter("Done");
```

Output:

```
$ java -jar language.jar closures.txt
ERROR: Something went wrong
INFO: Program started
ERROR: Connection failed
INFO: Done
```

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*