EVALUATOR

Prepared by Rene Andre B. Jocsing September 30, 2025

Welcome to the third chapter of your language hacking journey. Your scanner can tokenize source code, your parser can build syntax trees, and now it's time for the grand finale: **evaluation**. This is where your language comes to life—where those abstract syntax trees finally compute actual values.

Think of evaluation as the moment Dr. Frankenstein throws the switch. Your previously lifeless syntax tree will open its eyes, take a breath, and execute code. No lightning required (though it adds to the ambiance).

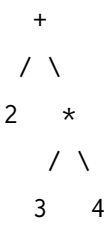
From Trees to Values

So far, you've built a pipeline that transforms source code into abstract syntax trees. But trees alone don't *do* anything. They're just data structures sitting in memory. To make your language useful, you need to traverse these trees and compute the values they represent.

Consider this simple expression:

```
1 2 + 3 * 4
```

Your parser built a tree that correctly captures precedence:



Now your evaluator needs to walk this tree and compute the result: `14`. The process is straightforward—evaluate the leaves first (the literals `2`, `3`, `4`), then work your way up, applying each operator to its operands.

This is called a **post-order traversal** because each node evaluates its children before doing its own work. It's the natural way of evaluating mathematical expressions, and it generalizes perfectly to more complex programming language constructs.

Representing Values

Before you can compute values, you need to decide how to represent them Kotlin. This is where dynamic typing gets interesting.

In your programming language, values are dynamically typed—a variable can hold a number now and a string later. But Kotlin is instead statically typed. How do you bridge this gap?

If you're using a language like Java or C#, you can use their `Object` type as your universal value representation. If you're using Python or JavaScript, the implementing language is already dynamically typed, so this problem doesn't exist. If you're using C++ or Rust, you might implement a variant type or a union. But in Kotlin, you can easily use the `Any?` type.

To summarize, you need:

- A way to store values of different types in the same variable
- A way to determine a value's type at runtime
- Conversion between your language's representation and your host language's types

Evaluator Architecture

Now you need to implement the logic that traverses your AST and computes values. There are several architectural approaches you can take, each with different trade-offs.

If you used the **Visitor pattern** in your parser, you can extend it here. Create an evaluator class that implements visitor methods for each AST node type. This keeps evaluation logic separate from your AST classes.

If you used the **Interpreter pattern**, you likely have an `evaluate()` or `interpret()` method directly on each AST node class. You can extend these methods to return computed values instead of just printing representations.

You might also implement evaluation using a **switch statement** or **method dispatch** based on node types, or use **function pointers/lambdas**.

Regardless of your architectural choice, the core principle remains the same: each AST node type needs corresponding logic that knows how to evaluate that construct. Your evaluation code structure should mirror your grammar—each production rule gets corresponding evaluation logic.

Your chosen approach should provide:

- Clear separation between parsing and evaluation concerns
- Easy maintainability as your language grows
- Consistent handling of the recursive tree traversal
- Clean error propagation when evaluation fails

Evaluating Different Expression Types

Let's walk through how to evaluate each kind of expression your parser can produce.

Literals are the easiest. A literal expression contains a value that was determined during scanning. To evaluate it, just return that value. No computation needed.

Grouping expressions (parentheses) are almost as simple. Just recursively evaluate the expression inside the parentheses and return the result. The parentheses themselves don't affect the value—they only influenced parsing precedence.

Unary expressions like `-5` or `!true` require you to:

1. Recursively evaluate the operand
2. Apply the unary operator to the result
3. Return the new value

For negation, make sure the operand is a number. For logical not, you'll need to decide what counts as "truthy" in your language.

Binary expressions are similar but with two operands:

1. Recursively evaluate the left operand
2. Recursively evaluate the right operand
3. Apply the binary operator
4. Return the result

Arithmetic operators (`+`, `-`, `*`, `/`) work on numbers. The `+` operator might also concatenate strings, depending on your language design. Comparison operators (`<`, `<=`, `>`, `>=`) compare numbers and return booleans. Equality operators (`==`, `!=`) work on any types.

Truthiness

Most dynamically typed languages partition all values into "truthy" and "falsey" categories. This determines how values behave in boolean contexts like `if` statements or the `!` operator.

The exact rules vary by language, but a common approach is:

- `false` and `null` are falsey
- Everything else is truthy

Some languages also treat empty strings, zero, or empty collections as falsey, but simplicity has its virtue.

Runtime Errors

Not all expressions can be successfully evaluated. What happens when the user tries to negate a string or add a number to a boolean? These are **runtime errors**—problems that can only be detected during execution.

When a runtime error occurs, you need to:

1. Stop evaluating the current expression

2. Report a meaningful error message to the user
3. Avoid crashing the entire interpreter

The third point is crucial. If users are running a REPL, they should be able to fix their mistake and continue. Don't let one bad expression kill the whole session.

Your error handling strategy will depend on your implementation. Exceptions work well. Error values or result types work too. The key is to unwind the evaluation stack cleanly when an error occurs.

Laboratory Deliverables

Expression Evaluator

Building on your parser from the previous lab, implement an evaluator that can compute the values of expressions in your language. Your evaluator should handle:

- Literal values (numbers, strings, booleans, nil)
- Unary expressions (negation, logical not)
- Binary expressions (arithmetic, comparison, equality)
- Grouping (parenthesized expressions)
- Proper operator precedence and associativity
- Other constructs as needed in your language

Runtime Error Handling

Implement graceful runtime error handling for type mismatches and other evaluation errors. Your interpreter should:

- Detect when operands have incorrect types for operators
- Report clear, helpful error messages that include line numbers
- Continue running after encountering errors (don't crash)
- Handle edge cases like division by zero (your choice how)

Integration and Testing

Update your REPL to evaluate expressions and print their results. Instead of showing the AST structure, users should now see computed values. Your evaluator marks a major milestone—you now have a working (if limited) programming language!

Expected Output

```
> 42
42
> 3.14159
3.14159
> "Hello, world!"
Hello, world!
> true
true
> false
false
> nil
nil
> -123
-123
> !true
false
> !false
true
> !nil
true
> !"hello"
true
> 1 + 2
3
> 3 - 1
2
> 2 * 3
6
> 8 / 4
2
> "Hello" + " " + "world"
Hello world
> 1 + 2 * 3
7
> (1 + 2) * 3
9
> 1 < 2
true
> 2 > 3
false
> 1 <= 1
true
> 2 >= 1
true
```

```
> 1 == 1
true
> 1 != 2
true
> "hello" == "hello"
true
> "hello" != "world"
true
> nil == nil
true
> true == false
false
> 1 == "1"
false
> (5 > 3) == true
true
> 3 + 4 * 5 == 23
true
> -1 + 2
1
> -(3 + 4)
-7
> 1 / 0
[line 1] Runtime error: Division by zero.
> "hello" + 5
[line 1] Runtime error: Operands must be two numbers or two strings.
> 5 > "hello"
[line 1] Runtime error: Operands must be numbers.
> -"hello"
[line 1] Runtime error: Operand must be a number.
```

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*