# One Hour Expression Language

How to make an expression language in one hour (or less)

Daniel Leech, DTL Software. 2025

(these slides are a WIP and cover the first 15-20m of the talk)

## Start your watch

Don't forget to start your stopwatch, Dan.

## What is an Expression?

An expression a sequence of symbols that can be *evaluated*.

The expression `1 + 1` will evaluate to `2`, other examples:

- `1` evaluates to `1`
- `true` evaluates to `true`
- `"foo" + "bar"` evaluates to `"foobar"`
- `true or false` evaluates to `true`
- `1 + 2 = 2` evaluates to `false`
- `1 + 1 = 2` evaluates to `true`
- `isFree or price = 0` evaluates to `true` when `price = 0`

## Why Make One?

- You never know when you need to **write your own parser**.
- Lean what the **AST** is and level up with **static analysis**.
- It's the first step towards **writing your own programming langauge!**

## Concrete Examples

- **Discount Rule Engine**: discount rule builder using criterias.
- **Transpiling code**: ported the VS Code Language Server Protocol from Typescript to PHP.
- **BDF Font Parser**: parser for BDF font files.
- **Syntax Highlighting**: parser for BDF font files.
- ...

## ProCalc2000

ProCalc2000 is a calculator for the year 2000 and **subsequent years**. A true calculator for the ages.

It allows you to evaluate `1 + 1` or `5 * 2 + 1 / 6` and finally **know the answers to PREVIOUSLY UNANSWERABLE MATHMEMATICAL FORMULATIONS**.

It's basic but provides all the machineary to write expression languages of arbitrary complexity.

# How Does ProCalc2000 Work?

- **Tokenize** an expression
- **Parse** the tokens to an **AST** (Abstract Syntax Tree)
- **Evaluate** the AST to a **value**

```
              +-----------+  tokens  +--------+  ast  +-----------+
EXPRESSION => | Tokenizer | -------> | Parser | ----> | Evaluator | => VALUE
              +-----------+          +--------+       +-----------+
```

## The Three Classes

- `Tokenizer`
- `Parser`
- `Evaluator`

# Tokenizer

Scan a string from left to right and produce tokens

```
"10 + 11234 / 20"
```

| **Offset** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** | **8** | **9** | **10** | **11** | **12** | **13** | **14** |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Char | 1 | 0 |  | + |  | 1 | 1 | 2 | 3 | 4 |  | / |  | 2 | 0 |

We **skip the whitespace** and identify 5 tokens:

```
[T_NUMBER] [T_PLUS] [T_NUMBER] [T_DIVIDE] [T_NUMBER]
    10        +       11234         /         20
```

# Tokenizer

The output of the tokenizer is a list of tokens:

```
Tokens(
    Token(T_NUMBER, 10),
    Token(T_PLUS),
    Token(T_NUMBER, 11234),
    Token(T_DIVIDE),
    Token(T_NUMBER, 20),
)
```

We can then feed these tokens **into the parser**.

# Parser

- The parser **makes sense** of the tokens and returns an **AST**.
- The **AST** is esentially the root of a tree of "nodes".
- A **Node** (in this case) is an *expression*.

## Parser

Our parser will have only two node types:

- `Integer`: Integer literal.
- `BinaryOp`: Binary operation.

## Parser: The Tree

The node hierarchy for `1 + 1 / 5`:

```
                        +-------------+
                        | Binary Op + | <-- root of the AST
                        +---+---+-----+
                      left  |   |  right
                    +-------+   +--------+
                    |                    |
                +-----------+      +-------------+
                | Integer 1 |      | Binary Op / |
                +-----------+      +-------------+
                                   left |   |right
                                 +------+   +----+
                                 |               |
                           +-----+-----+    +----+------+
                           | Integer 1 |    | Integer 5 |
                           +-----------+    +-----------+
```

# Parser: BinaryOp Node

```
class BinaryOp implements Node {
    public function __construct(
        public Node $left,
        public string $operation,
        public Node $right
    ) {
    }
}
```

# Parser: Integer Node

```php
class Integer implements Node {
    public function __construct(
        public int $value,
    ) {
    }
}
```

# Parser: The Result

The result of parsing `1 + 1 / 5` will be the AST:

```
$ast = new BinaryOp(
    left:     new Integer(1),
    operator: '+',
    right:    new BinaryOp(
        left:     new Integer(1),
        operator: '/',
        right:    new Integer(5),
    )
);
```

## Evaluator

The evaluator **walks** the AST. Tree Walking is a **massively powerful** and **important** pattern! It's also **extremely simple**. It's basically **Thanos**.

# I can almost fit an evaluator in this slide:

```
class Evaluator {
    public function evaluate(Node $node): int
    {
        if ($node instanceof Integer) {
           return $node->value;
        }

        if ($node instanceof BinaryOp) {
            $leftValue = $this->evaluate($node->left);
            $rightValue = $this->evaluate($node->right);

           return match ($node->operator) {
              '+' => $leftValue + $rightValue,
              '/' => $leftValue / $rightValue,
              // ...
           }
        }

        throw new Exception(sprintf(
            'Do not know how to evaluate node: %s',
            $node::class
        ));
    }
}
```

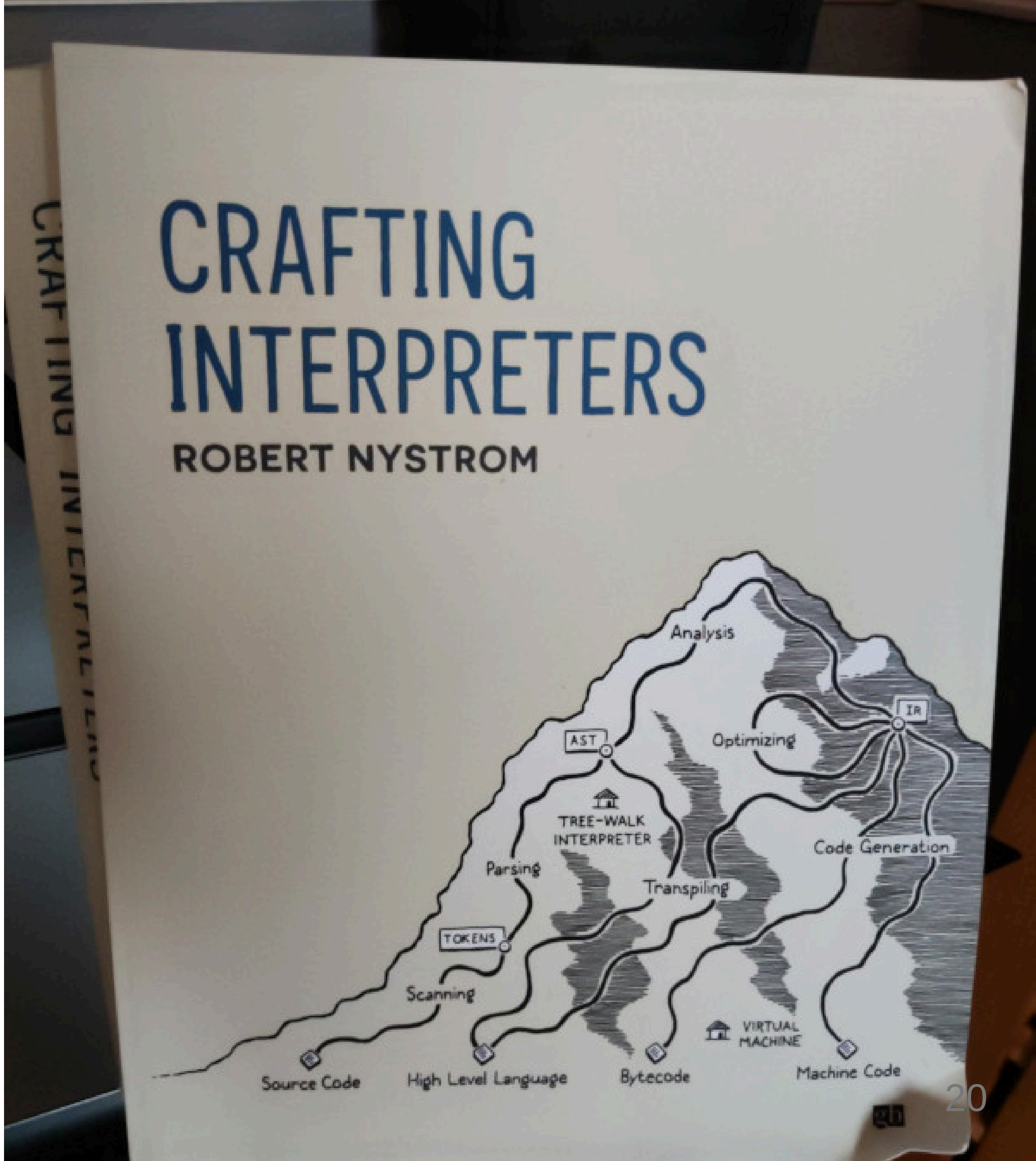
## That's it. That's the intro.

Personal note: the intro should take 22 minutes 12 seconds how long did it take? Oh, you forgot to start your stopwatch didn't you. **How much time have I got**?

(live coding starts here and lasts ~30-40m)

## Further Reading

- Crafting Interpreters
- Pratt Parsing Blog Post



## Parser: Warning About Operator Precedence

Our parser will evaluate `2 * 2 + 3` as `2 * (2 + 3)` which is **not what you'd expect**.

We should perform **multiplication before addition** giving us `(2 * 2) + 3`.

Due to time constraints I won't implement a **Pratt Parser** because recursion gives **you** a headache.

## Parser: Node Inheritance Diagram

All nodes should at *least* implement a marker interface to indicate that they are part of the AST:

```
                +----------+
                |   Node   |
                +-----+----+
                      |
              +------+---------+
              |                |
        +-----------+    +-----+-----+
        | Binary Op |    |  Integer  |
        +-----------+    +-----------+
```