How to make a programming language
Preface: trees

a unique path from root to every element
The “Calc” language

Operator tree
The leaves are numbers.
The non-leaves are operators.

Post-order traversal
Evaluate the left sub-tree,
Evaluate the right sub-tree,
Combine the results.
An inductive definition of operator trees

\[ 
\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\
\mid \text{Expr} \times \text{Expr} \mid \text{Expr} / \text{Expr} \\
\mid (\text{Expr}) \mid \text{Integer} 
\]
How to turn trees into OO (Java) code

object, constant, enumeration, or nothing
for each “literal” thing

interface implementation
class
for each “kind” of expression

Expr → Expr + Expr | Expr - Expr
Expr * Expr | Expr / Expr
(Expr) | Integer
How to implement an evaluator

For each kind of expression, define an evaluate method

```
Expr → Expr + Expr | Expr - Expr
Expr * Expr | Expr / Expr
(Expr) | Integer
```

interface implementation

*a method* for each “kind” of expression
Is this a programming language?

What is a programming language?
An inductive definition of operator trees

\[ \text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\
\mid \text{Expr} \ast \text{Expr} \mid \text{Expr} / \text{Expr} \\
\mid (\text{Expr}) \mid \text{Integer} \]
Abstract Syntax Trees (ASTs)

An intermediate representation

\[
\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\
\mid \text{Expr} \times \text{Expr} \mid \text{Expr} / \text{Expr} \\
\mid (\text{Expr}) \mid \text{Integer}
\]
Grammars
A specification for syntactically valid programs

```
Grammars

Expr → Expr + Expr | Expr - Expr
     | Expr * Expr | Expr / Expr
     | (Expr)     | Integer

Integer → Digit | Digit Integer

Digit → 0 | 1 | 2 | 3 | 4
     | 5 | 6 | 7 | 8 | 9
```

start symbol

alternative

production rule

non-terminal
Tokenization

Group the characters of an input string into meaningful “chunks” (i.e., tokens)

```
['1', '2', ' ', '+', ' ', '2']
```

```
[12, +, 2]
```

list of characters

list of tokens

```
tokenize
```

""
How to (start to) implement a tokenizer

Each terminal is an object, constant, enumeration, or etc.

Expr → Expr + Expr | Expr - Expr
| Expr * Expr | Expr / Expr
| (Expr) | Integer
Parsing
Transform a string to an AST

['1', '2', ',', '+', ',', '2']  [12, +, 2]  +

list of characters  list of tokens  AST
How to implement a parser
using recursive descent

```
Expr → Expr + Expr | Expr – Expr
    | Expr * Expr | Expr / Expr
    | (Expr)     | Integer
```

**Input:** a *stack* of tokens

**Output:** an AST (or a syntax error)
How to implement a parser
using recursive descent

Input: a stack of tokens
Output: an AST (or a syntax error)

• Each non-terminal is a method name
• Each rule is the body of a method
• Start at the top (i.e., the start symbol)
• We can: peek or pop the token stack or call a method
• Build the AST as we go
• Empty token stack + end of start-symbol method ⇒ success!

```
Expr → Expr + Expr | Expr − Expr
    | Expr * Expr | Expr / Expr
    | (Expr) | Integer
```
Left recursion :(
a naïve recursive-descent parser will never terminate

```
Expr → Expr + Expr | Expr − Expr
    | Expr * Expr | Expr / Expr
    | (Expr) | Integer
```

Input: a stack of tokens
Output: an AST (or a syntax error)

- Each non-terminal is a method name
- Each rule is the body of a method
- Start at the top (i.e., the start symbol)
- We can: peek or pop the token stack or call a method
- Build the AST as we go
- Empty token stack + end of start-symbol method ⇒ success!
An equivalent (but not left-recursive) grammar

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} + \text{Expr} \mid \text{Term} - \text{Expr} \mid \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \ast \text{Term} \mid \text{Factor} / \text{Term} \mid \text{Factor} \\
\text{Factor} & \rightarrow ( \text{Expr} ) \mid \text{Integer}
\end{align*}
\]

**Input:** a stack of tokens

**Output:** an AST (or a syntax error)

- Each non-terminal is a method name
- Each rule is the body of a method
- Start at the top (i.e., the start symbol)
- We can: peek or pop the token stack or call a method
- Build the AST as we go
- Empty token stack + end of start-symbol method $\Rightarrow$ success!
Ambiguous grammars

Can a single list of tokens be interpreted in multiple ways?

\[
\begin{align*}
\text{Expr} & \rightarrow \text{Term} + \text{Expr} \mid \text{Term} - \text{Expr} \mid \text{Term} \\
\text{Term} & \rightarrow \text{Factor} \ast \text{Term} \mid \text{Factor} / \text{Term} \mid \text{Factor} \\
\text{Factor} & \rightarrow (\text{Expr}) \mid \text{Integer}
\end{align*}
\]

**Precedence:** In a list of tokens that contains different operators, which operator should be applied first?

1 + 2 * 3 : * should it be (1 + 2) * 3 or 1 + (2 * 3) *

**Associativity:** In a list of tokens that contains multiple instances of the same operator, which operator should be applied first?

1 − 2 − 3 : * should it be (1 − 2) − 3 or 1 − (2 − 3) *
The “Unicalc” language

Quantity lists + Normalization