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

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

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

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

interface implementation
class
for each “kind” of expression
How to implement an evaluator

For each kind of expression, define an evaluate method

interface implementation

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

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

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

Expr → Expr + Expr | Expr − Expr
| Expr * Expr | Expr / Expr
| (Expr) | Integer
An intermediate representation

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

An intermediate representation

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

A specification for syntactically valid programs

start symbol

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

production rule

Integer → Digit | Digit Integer

non-terminal

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

alternative

terminal
Tokenization

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

[ '1', '2', '+', '2' ]  
list of characters

[ 12, +, 2 ]  
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 ]  →  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)
• Build the AST as we go
• We can peek or pop the token stack
• Empty token stack + end of start symbol method ⇒ success!

\[
\text{Expr} \rightarrow \text{Expr} + \text{Expr} \mid \text{Expr} - \text{Expr} \\
\mid \text{Expr} \ast \text{Expr} \mid \text{Expr} / \text{Expr} \\
\mid (\text{Expr}) \mid \text{Integer}
\]
Left recursion :-(

A naïve recursive descent parser will recur forever.

```
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)
- Build the AST as we go
- We can peek or pop the token stack
- 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)  
- Build the AST as we go  
- We can peek or pop the token stack  
- Empty token stack + end of start symbol method ⇒ success!
Ambiguous grammars
Can a single list of tokens be interpreted in multiple ways?

\[
\text{Expr} \rightarrow \text{Term} \ + \ \text{Expr} \ | \ \text{Term} \ - \ \text{Expr} \ | \ \text{Term}
\]
\[
\text{Term} \rightarrow \text{Factor} \ * \ \text{Term} \ | \ \text{Factor} \ / \ \text{Term} \ | \ \text{Factor}
\]
\[
\text{Factor} \rightarrow (\text{Expr}) \ | \ \text{Integer}
\]

**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?
\[1 \ - \ 2 \ - \ 3 : \ should \ it \ be \ (1 \ - \ 2) \ - \ 3 \ or \ 1 \ - (2 \ - \ 3)\]
Ambiguous grammars

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

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

The grammar on this slide has the appropriate precedence; but the associativity is unspecified, so the grammar is ambiguous.

The parsing algorithm can determine associativity.

Recursive-descent parsing is right-associative.
Why? And why is this bad?
The “Unicalc” language

Quantity lists + Normalization