



---

# Introduction to ANTLR

---

MIKE CARGAL

# What is ANTLR?

- ▶ A Parser Generator
- ▶ From an ANTLR grammar file, generates:
  - ▶ A recursive Descent Parser in the “target” language
    - ▶ Java, C#, Python, JavaScript, Go, C++, Swift, PHP, Dart (official, others can be found)
  - ▶ Optional tree navigation classes (Listeners and/or Visitors)
  - ▶ Optional artifacts for debugging/documenting a grammar
    - ▶ ATN files (in .dot format) (diagram the Augmented Transition Network used to match Lexer and Parser rules)

# Why ANTLR ?

(It's not **that** hard to write a recursive descent parser)

- ▶ ANTLR grammars are much easier to follow than your parser code
- ▶ Why write (and debug) repetitive code that can be generated?
- ▶ Good error messages are non-trivial to code
- ▶ ANTLR has a good algorithm for either ignoring or inserting tokens to facilitate error recovery
- ▶ ANTLR handles direct left-recursion for you
- ▶ ANTLR handles operator precedence and associativity for you
- ▶ ANTLR ALL(\*) algorithm handles many grammars that would not be possible to parse with simple recursive descent parsers
- ▶ ANTLR generates **very** useful classes for processing the resulting parse tree

---

# What ANTLR is not

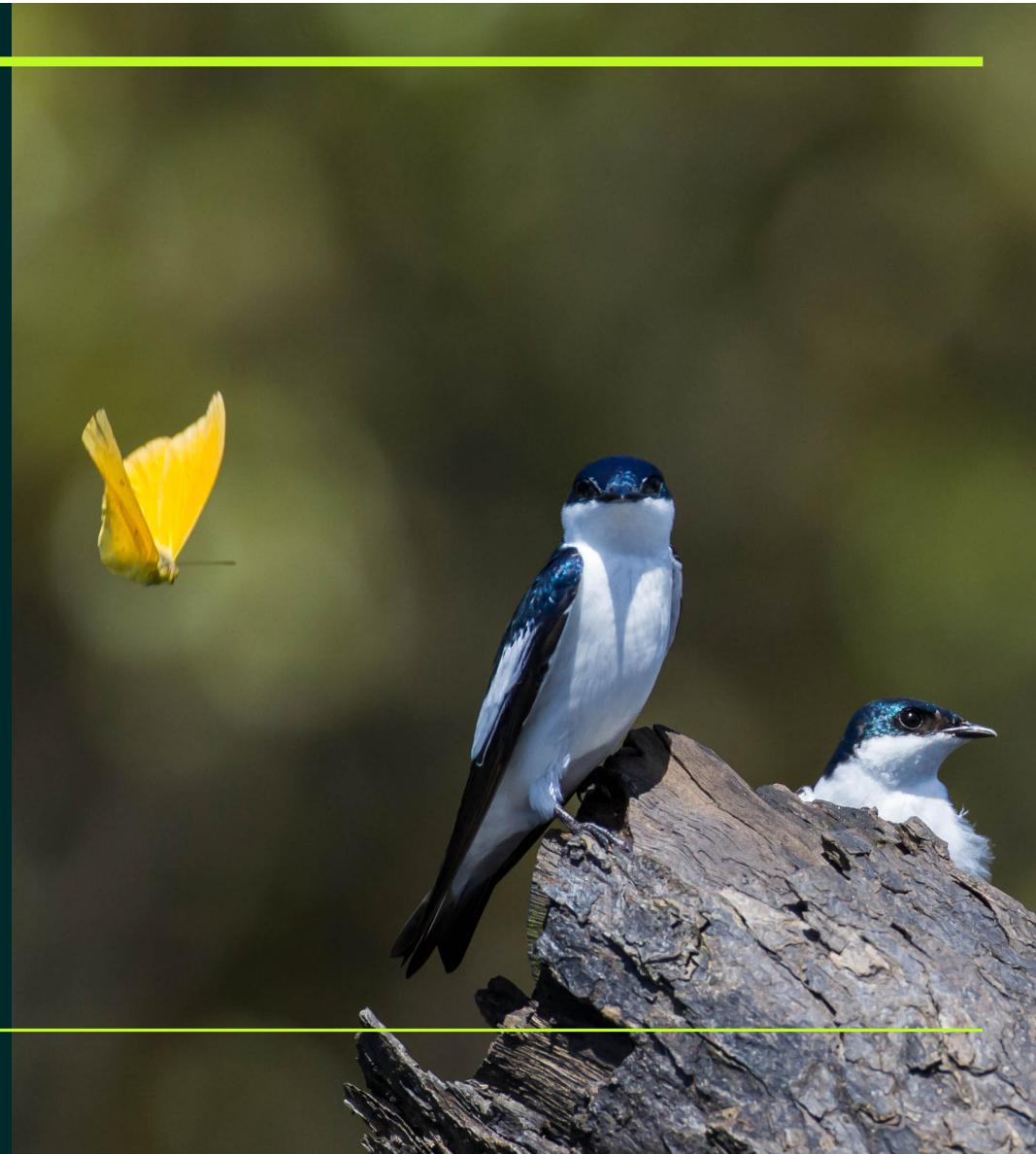
- ANTLR is not intended as a Language Generator
  - With embedded actions, you **can** have ANTLR produce a grammar that will “execute” your code. But:
    - This generally breaks down beyond very simple examples
    - Embedded actions are written in the target language and limit the usefulness of your grammar (especially, in an increasingly polyglot environment)
-

---

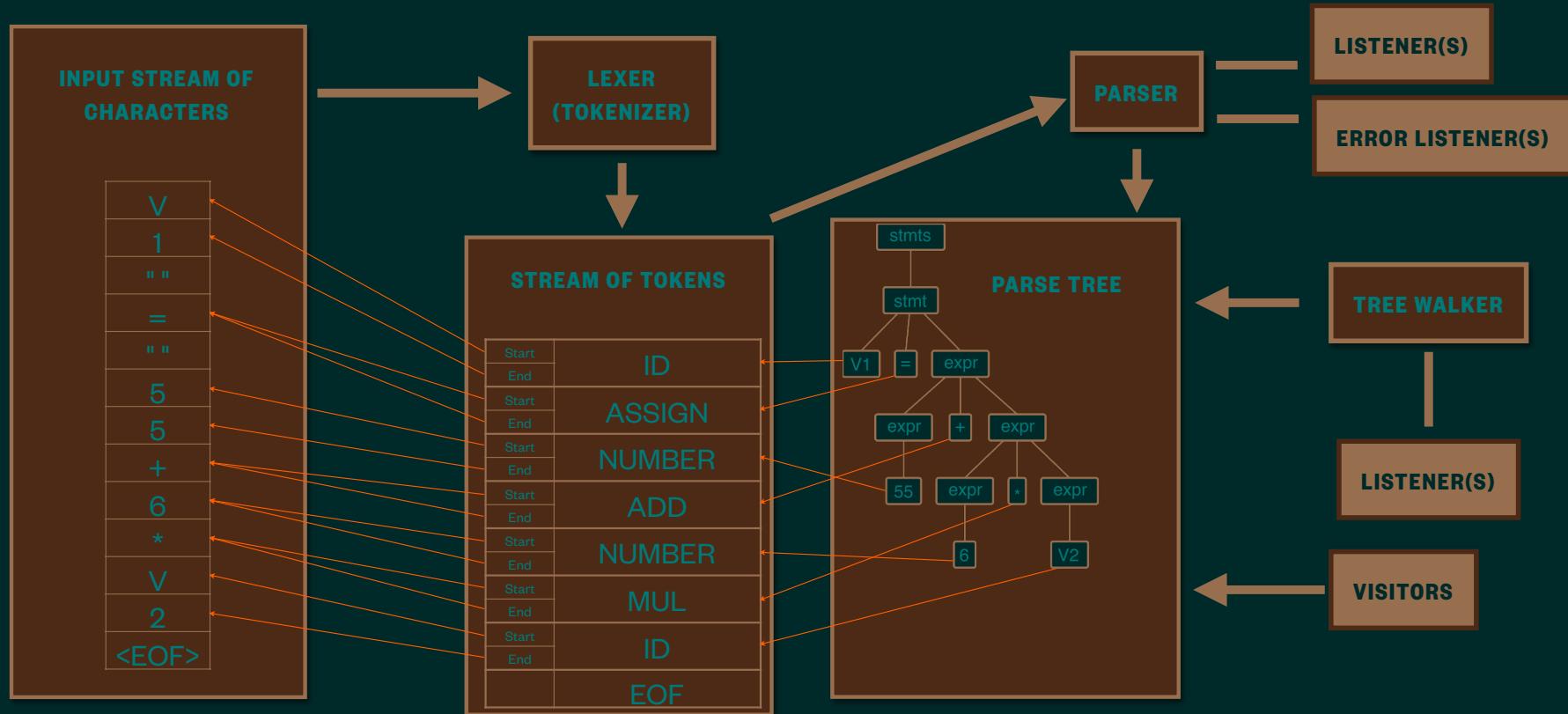
# This is an “Intro”

- ▶ What we **will** cover:
    - ▶ Target Language independent grammars
    - ▶ Most common options
    - ▶ A few “hidden gems”
  - ▶ What we **will not** cover:
    - ▶ Target Language specific grammars
      - ▶ Includes Semantic Predicates, @header {}, etc.
        - ▶ while obviously “useful” these are beyond the scope of what I hope to cover in an intro
        - ▶ lock you into using a grammar to target a specific language
    - ▶ **Every** option in the ANTLR grammar syntax
-

# Overview



# ANTLR parsing pipeline





# ANTLR Grammar Syntax



```
1 grammar MyGrammar;
2
3 options { tokVocab=MyLexer }
4 import ...
5
6 parserRule: TOKEN* EOF;
7
8 TOKEN: [a-zA-Z]+;
9 WS: [ \t\r\n]* → skip;
```

- ▶ 1 - “grammar” followed by Grammar Name (for combined grammar)
  - ▶ “lexer grammar ...” for Lexer Grammars
  - ▶ “parser grammar ...” for parser Grammars
- ▶ 3 - Optional “options { ... }”
  - ▶ “tokenVocab” - reference a “\*.tokens” file listing token names and unique token types (integers)
  - ▶ “language” - specify target language
- ▶ 4 - include content for another \*.g4 file
- ▶ 6... parser and grammar rules

---

# Lexer and Parser Rules

- Lexer rules begin with a upper case letter
  - Parser rules begin with a lower case letter
    - Parser rules have NO impact on Lexer rules.
  - Both allow for cardinality modifiers:
    - **+** - One or more instances of the preceding item
    - **\*** - Zero or more instances of the preceding item
    - **?** - Zero or one instance of the preceding item (aka “optional”)
    - **?** - an additional **?** makes the prior cardinality “non-greedy”
  - Both allow for **|** to indicate “or” (alternatives)
  - Both allow for **( ... )** to group constructs (usually to apply cardinality or alternatives to a set of rules)
-

# Lexer Rule syntax

```
● ● ●
1 MUL:      '*';
2 fragment DIGIT: [0-9];
3 fragment ALPHA: [a-zA-Z];
4 NUMBER:     (DIGIT | '_')+ ('.' (DIGIT | '_')+)?;
5 fragment STRING_CONTENT: ('\\\"' | '\\\\\'' | .);
6 STRING:     '"' STRING_CONTENT*? '"';
7 S_STRING:   '\'' STRING_CONTENT*? '\'' → type(STRING);
8 ID:         (ALPHA | '_') (ALPHA | DIGIT | '_')*;
9 COMMENT:    '/*' .*? ('\n' | EOF) → channel(HIDDEN);
10 WS:        [ \t\r\n]+ → skip;
11 BAD_TOKEN: .;
```

- ▶ 1 - Simple character token
- ▶ 2 - a “fragment rule”
  - ▶ Will not match to generate a Token
  - ▶ Can be used to build up “real” Token rules
- ▶ 2 - [ and ] enclose sets of values (and a - indicates a range of values)
- ▶ 5 - A **NUMBER** is one or more **DIGIT**s or underscores optionally followed by a period and one or more **DIGITS** or underscores.
- ▶ 6 - A **\*?** can follow a cardinality modifier (**+,\*,?**) to indicate that it should be “non-greedy”. ie. This rule says a **STRING** is a " followed by zero or more \" or \' or any character(.) but will consume them non-greedily, so that it stops consuming at the first non-escaped "
- ▶ 7 - The **S\_STRING** rule recognizes strings delimited by 's, but to make things easier in the parser rules -> **type(STRING)** means it will be given a **STRING** token type.
- ▶ 8 / 9 - -> **skip** and -> **channel(HIDDEN)** both result in those tokens not being presented to the Parser
  - ▶ **channel(HIDDEN)** still produces the tokens and includes them in the token stream

# Matching Multiple Lexer Rules

- ▶ The rule that consumes the most input will be chosen
  - ▶ “guilty” will be an **ID** token
  - ▶ while it begins with “gui” the **ID** rule will match all 5 characters
- ▶ If the lengths are the same:
  - ▶ The rule occurring first in the grammar will be chosen
  - ▶ “gui” will be a **GUI** token (not an **ID**)
    - ▶ but... *only* if it occurs before the **ID** rule
  - ▶ “@“ would be a **BAD\_TOKEN**



```
1 GUI:      'gui';
2 TREE:     'tree';
3 REFACTOR: 'refactor';
4 fragment DIGIT: [0-9];
5 fragment ALPHA: [a-zA-Z];
6 ID:        (ALPHA | '_') (ALPHA | DIGIT | '_')*;
7 BAD_TOKEN: '@';
```

# Case-insensitivity



```
1 PRINT: [Pp][Rr][Ii][Nn][Tt];
2 // or
3 VARS:      V A R S;
4 fragment A: [Aa];
5 fragment B: [Bb];
6 fragment C: [Cc];
7 fragment D: [Dd];
8 ...
```

- ▶ 1 - PRINT matches upper or lower case P, followed by upper or lower case R, followed by...
  - ▶ will match “PRINT”, “print”, “pRiNt”, etc.
  - ▶ can be tedious
- ▶ 3 - VARS uses a set of fragment rules we create for the whole (Latin) alphabet.
  - ▶ will match “VARS”, “vars”, “vArS”, etc.
  - ▶ rule is a bit easier to read

# Parser Rule Syntax

● ● ●

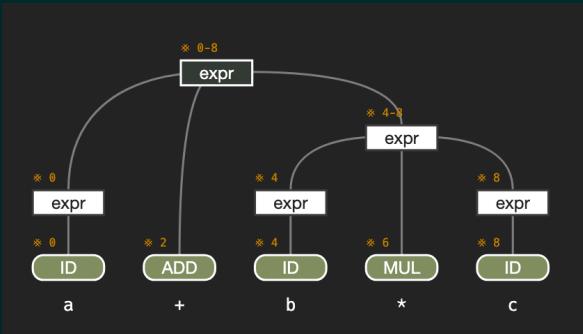
```
1  expr
2  : '(' expr ')'
3  | <assoc = right> base = expr '^' exp = expr
4  | lhs = expr op = ('*' | '/') rhs = expr
5  | lhs = expr op = ('+' | '-') rhs = expr
6  | lhs = expr op = ('<' | '≤' | '≥' | '>') rhs = expr
7  | lhs = expr op = ('=' | '≠') rhs = expr
8  | cond = expr '?' tv = expr ':' fv = expr
9  | lhs = expr '&&' rhs = expr
10 | lhs = expr '||' rhs = expr
11 | '!' expr
12 | NUMBER
13 | TRUE
14 | FALSE
15 | STRING
16 | ID
17 ;
18 ;
```

- ▶ Tokens may be referred to by their Lexer Rule name
  - ▶ (ex: **LT**, **AND**, **STRING**)
- ▶ Literal Strings can also be used for Token references
  - ▶ (ex: '**(**', '**\***', '**:**')
  - ▶ Will match any defined tokens (**LT**, and '**<**' are equivalent)
  - ▶ If no Token rule matches:
    - ▶ Will synthesize a Token Rule (ex: **T\_\_0**)
    - ▶ Will throw a warning in a parser grammar
- ▶ Alternatives are separated by a | character.
  - ▶ Top level alternatives establish precedence.

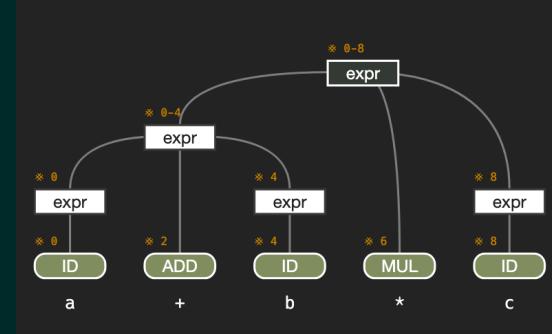
# Precedence



```
1 expr
2   :
3   | expr ('*' | '/') expr
4   | expr ('+' | '-') expr
5   | ID
6 ;
```



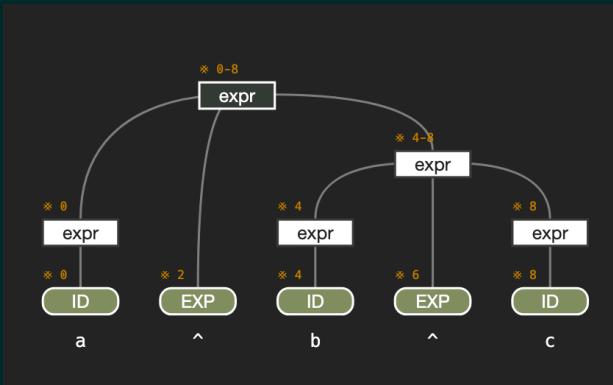
```
1 expr
2   :
3   | expr ('+' | '-') expr
4   | expr ('*' | '/') expr
5   | ID
6 ;
```



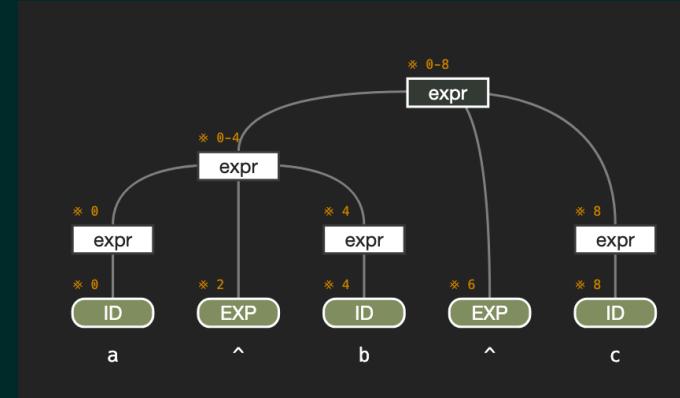
# Associativity



```
1 expr
2     : <assoc = left> expr '^' expr
3     | ID
4 ;
```



```
1 expr
2     : expr '^' expr
3     | ID
4 ;
```



# Left Recursion



```
1 expr
2     : expr ('*' | '/') expr
3     | NUMBER
4     | ID
5 ;
```



```
1 rule1 : rule2;
2 rule2 : rule1 | ID;
```

- ▶ When the first item in a rule recursively references itself, this is called “Left Recursion”
  - ▶ In a naive Recursive Descent Parser, this results in infinite recursion.
  - ▶ ANTLR can deal with “direct” left recursion
    - ▶ ex: **expr**
  - ▶ ANTLR will not allow for “indirect”
    - ▶ ex: **rule1 / rule2**

```
The following sets of rules are mutually left-recursive [rule1, rule2]
View Problem (36.) No quick fixes available
rule1:: rule2;
rule2:: rule1 | ID;
```

# Unlabelled Alternatives



```
1 expr
2   : '(' expr ')'
3   | <assoc = right> expr '^' expr
4   | expr ('*' | '/') expr
5   | expr ('+' | '-') expr
6   | expr ('<' | '<=' | '>' | '>') expr
7   | expr ('=' | '≠') expr
8   | expr '?' expr ':' expr
9   | expr '??' expr
10  | expr '||' expr
11  | '!' expr
12  | NUMBER
13  | TRUE
14  | FALSE
15  | STRING
16  | ID
17  ;
```



```
1 public static class ExprContext extends ParserRuleContext {
2     public TerminalNode O_PAREN() { return getToken(LittleCalcParser.O_PAREN, 0); }
3     public List<ExprContext> expr() {
4         return getRuleContexts(ExprContext.class);
5     }
6     public ExprContext expr(int i) {
7         return getRuleContext(ExprContext.class,i);
8     }
9     public TerminalNode C_PAREN() { return getToken(LittleCalcParser.C_PAREN, 0); }
10    public TerminalNode NOT() { return getToken(LittleCalcParser.NOT, 0); }
11    public TerminalNode NUMBER() { return getToken(LittleCalcParser.NUMBER, 0); }
12    public TerminalNode TRUE() { return getToken(LittleCalcParser.TRUE, 0); }
13    public TerminalNode FALSE() { return getToken(LittleCalcParser.FALSE, 0); }
14    public TerminalNode STRING() { return getToken(LittleCalcParser.STRING, 0); }
15    public TerminalNode ID() { return getToken(LittleCalcParser.ID, 0); }
16    public TerminalNode EXP() { return getToken(LittleCalcParser.EXP, 0); }
17    public TerminalNode MUL() { return getToken(LittleCalcParser.MUL, 0); }
18    public TerminalNode DIV() { return getToken(LittleCalcParser.DIV, 0); }
19    public TerminalNode ADD() { return getToken(LittleCalcParser.ADD, 0); }
20    public TerminalNode SUB() { return getToken(LittleCalcParser.SUB, 0); }
21    public TerminalNode LT() { return getToken(LittleCalcParser.LT, 0); }
22    public TerminalNode LE() { return getToken(LittleCalcParser.LE, 0); }
23    public TerminalNode GE() { return getToken(LittleCalcParser.GE, 0); }
24    public TerminalNode GT() { return getToken(LittleCalcParser.GT, 0); }
25    public TerminalNode EQ() { return getToken(LittleCalcParser.EQ, 0); }
26    public TerminalNode NE() { return getToken(LittleCalcParser.NE, 0); }
27    public TerminalNode QMARK() { return getToken(LittleCalcParser.QMARK, 0); }
28    public TerminalNode COLON() { return getToken(LittleCalcParser.COLON, 0); }
29    public TerminalNode AND() { return getToken(LittleCalcParser.AND, 0); }
30    public TerminalNode OR() { return getToken(LittleCalcParser.OR, 0); }
31    ...
32 }
```

# Labeled Alternatives



```
1  expr
2    : '(' expr ')'
3    | <assoc = right> expr '^' expr
4    | expr ('*' | '/') expr
5    | expr ('+' | '-') expr
6    | expr ('<' | '≤' | '≥' | '>') expr
7    | expr ('=' | '≠') expr
8    | expr '?' expr ':' expr
9    | expr '&&' expr
10   | expr '||' expr
11   | '!' expr
12   | NUMBER
13   | TRUE
14   | FALSE
15   | STRING
16   | ID
17   ;
```



```
1  public static class MulDivExprContext extends ExprContext {
2    public List<ExprContext> expr() {
3      return getRuleContexts(ExprContext.class);
4    }
5    public ExprContext expr(int i) {
6      return getRuleContext(ExprContext.class,i);
7    }
8    public TerminalNode MUL() { return getToken(LittleCalcParser.MUL, 0); }
9    public TerminalNode DIV() { return getToken(LittleCalcParser.DIV, 0); }
10   public MulDivExprContext(ExprContext ctx) { copyFrom(ctx); }
11   // ...
12 }
13 public static class EqualityExprContext extends ExprContext {
14   public List<ExprContext> expr() {
15     return getRuleContexts(ExprContext.class);
16   }
17   public ExprContext expr(int i) {
18     return getRuleContext(ExprContext.class,i);
19   }
20   public TerminalNode EQ() { return getToken(LittleCalcParser.EQ, 0); }
21   public TerminalNode NE() { return getToken(LittleCalcParser.NE, 0); }
22   public EqualityExprContext(ExprContext ctx) { copyFrom(ctx); }
23   // ...
24 }
```

# Labeled Rule Elements



```
1 | expr ('*' | '/') expr      # MulDivExpr
```



```
1 public static class MulDivExprContext extends ExprContext {  
2     public List<ExprContext> expr() {  
3         return getRuleContexts(ExprContext.class);  
4     }  
5     public ExprContext expr(int i) {  
6         return getRuleContext(ExprContext.class,i);  
7     }  
8     public TerminalNode MUL() { return getToken(LittleCalcParser.MUL, 0); }  
9     public TerminalNode DIV() { return getToken(LittleCalcParser.DIV, 0); }  
10    public MulDivExprContext(ExprContext ctx) { copyFrom(ctx); }
```



```
1 public LittleValue visitMulDivExpr(MulDivExprContext ctx) {  
2     var res = ctx.MUL() != null //  
3             ? number(ctx.expr(0)) * number(ctx.expr(1)) //  
4             : number(ctx.expr(0)) / number(ctx.expr(1));  
5     return lvNumber(res, ctx);  
6 }
```



```
1 | lhs = expr op = ('*' | '/') rhs = expr      # MulDivExpr
```



```
1 public ExprContext lhs;  
2 public Token op;  
3 public ExprContext rhs;  
4 public List<ExprContext> expr() {  
5     return getRuleContexts(ExprContext.class);  
6 }  
7 public ExprContext expr(int i) {  
8     return getRuleContext(ExprContext.class,i);  
9 }  
10 public TerminalNode MUL() { return getToken(LittleCalcParser.MUL, 0); }  
11 public TerminalNode DIV() { return getToken(LittleCalcParser.DIV, 0); }  
12 public MulDivExprContext(ExprContext ctx) { copyFrom(ctx); }
```



```
1 public LittleValue visitMulDivExpr(MulDivExprContext ctx) {  
2     var res = ctx.op.getType() == LittleCalcLexer.MUL //  
3             ? number(ctx.lhs) * number(ctx.rhs) //  
4             : number(ctx.lhs) / number(ctx.rhs);  
5     return lvNumber(res, ctx);  
6 }
```

---

# Getting Started

---



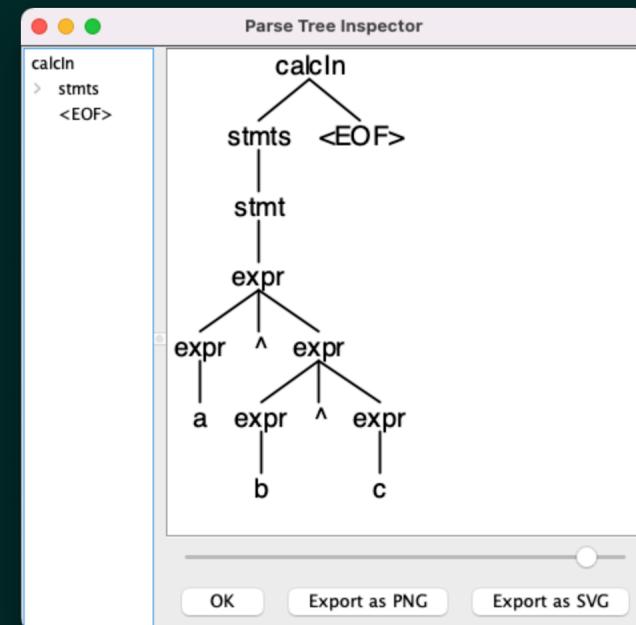
---

# Setup and basic commands

- ▶ Getting started guide at: <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>
  - ▶ sets up **antlr** and **grun** aliases
- ▶ **antlr** command used to generate source files
  - ▶ > **antlr4 MyGrammar.g4**
    - ▶ generates Java source code in current directory
  - ▶ > **java \*.java**
    - ▶ compiles classes so you can use **grun**
  - ▶ > **grun MyGrammar tokens -tokens <sourceFile**
    - ▶ Good start is dumping the stream of tokens to verify Lexer rules
  - ▶ > **grun MyGrammar startRule -gui <sourceFile**
    - ▶ Graphical tree view of Parse Tree

# Setup and basic commands

- ▶ Getting started guide at: <https://github.com/antlr/antlr4/blob/master/doc/getting-started.md>
  - ▶ sets up **antlr** and **grun** aliases
- ▶ **antlr** command used to generate source files
  - ▶ > **antlr MyGrammar.g4**
    - ▶ generates Java source code in current directory
  - ▶ > **java \*.java**
    - ▶ compiles classes so you can use **grun**
  - ▶ > **grun MyGrammar tokens -tokens <sourceFile**
    - ▶ Good start is dumping the stream of tokens to verify Lexer rules
  - ▶ > **grun MyGrammar startRule -gui <sourceFile**
    - ▶ Graphical tree view of Parse Tree



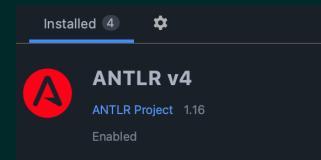
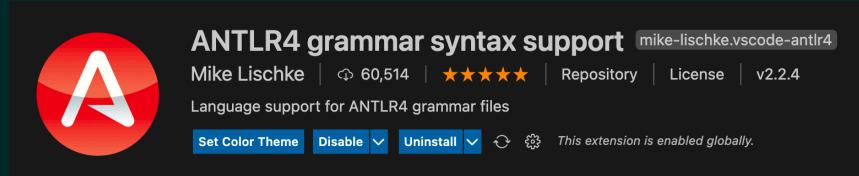
# Gradle integration (for Java)



```
1 plugins { id 'antlr' }
2 dependencies { antlr "org.antlr:antlr4:4.9.2" }
3 generateGrammarSource {
4     maxHeapSize = "64m"
5     arguments += [
6         "-visitor",
7         "-listener",
8         "-long-messages",
9         "-package", "net.cargal.littlecalc",
10        "-lib", "src/main/antlr/net/cargal/littlecalc"
11    ]
}
```

- ▶ antlr plugin defaults to:
  - ▶ grammars under “src/main/antlr/<pkg>”
  - ▶ generated code at “build/generated-src/<pkg>”
- ▶ you’ll need to specify “-lib” if you use “import”

# ANTLR plugins



- ▶ I'll be using Mike Lischke's VS Code plugin in any demos
- ▶ For IntelliJ fans, there's a nice ANTLR plugin as well.
- ▶ Eclipse plugin seems to have “fallen on hard times”

# VS Code plugin

- ▶ “Railroad” (RRD) diagrams
- ▶ ATN graphs for specific rules (\*)
- ▶ Interactive Grammar Call Graph
- ▶ Very flexible code formatting
- ▶ Debugging
- ▶ Example setup:



```
1  {
2      "name": "antlr4-littleCalcExpr",
3      "type": "antlr-debug",
4      "request": "launch",
5      "input": "${workspaceFolder}/expr.llt",
6      "grammar": "${workspaceFolder}/src/main/antlr/net/cargal/littlecalc/LittleCalc.g4",
7      "startRule": "expr",
8      "printParseTree": true,
9      "visualParseTree": true
10 }
```

---

# IntelliJ plugin

- Very nice “Parse Tree” view that updates immediately if you’re using the “input” from the UI.
  - An alternative “Hierarchy” view that is more navigable for large parse trees.
  - A “Profiler” view that provides quite a lot of valuable expert level information if you’re seeking working on performance issues.
-

---

# Listeners and Visitors

---



---

# Listeners

- ▶ Walks starting node and all descendent nodes of a parse tree
  - ▶ May be root node of parse tree, but can be any node (will visit all children)
  - ▶ You don't handle the navigation, you just "listen in" as nodes are visited
  - ▶ each Node type has an `enter*()` method and an `exit*()` method
    - ▶ a `*Listener` interface is generated with all method signatures and a `*BaseListener` class is generated that implements a "do nothing" implementation of all methods.
    - ▶ override the `enter*()` method to take action before visiting child nodes
    - ▶ override the `exit*()` method if you need child nodes to have already been processed
  - ▶ Methods do not return values, but you can maintain whatever state you need in your listener

# Visitors

---

- ▶ ANTLR can create an `*Visitor<T>` interface that returns an object of type `T` from each `visit*`() method.
  - ▶ Can use java `Void` class to return nothing (each method must `return null;`)
  - ▶ Navigation is entirely up to the implementor
    - ▶ Being in control of which children to “visit”, when, and how often is very flexible (can implement an interpreter)
  - ▶ Can be useful to define multiple Visitors returning different result types, and “mix and match”
  - ▶ The `*BaseVisitor` class defines a default implementation of each method that calls `visitChildren()`
    - ▶ By default, this method returns `null`

---

# Visitors (easing the pain)

- The bare default can be tedious, but methods exist that you can override to assist.
  - `AbstractParseTreeVisitor<T>` defines default methods for:
    - `T visit(ParseTree tree)`
    - `T visitChildren(RuleNode node)`
    - `T visitTerminal(TerminalNode node)`
    - `T visitErrorNode(ErrorNode node)`
    - `T defaultResult()`
    - `T aggregateResult(T aggregate, T nextResult)`
    - `boolean shouldVisitNextChild(RuleNode node, T currentResult)`
  - Overriding some of these methods and using them can make parse trees much easier to deal with using Visitors
-

# Visitors (easing the pain)

- The bare default can be tedious, but methods exist that you can override to assist.
- `AbstractParseTreeVisitor<T>` defines default methods for:
  - `T visit(ParseTree tree)`
  - `T visitChildren(RuleNode node)`
  - `T visitTerminal(TerminalNode node)`
  - `T visitErrorNode(ErrorNode node)`
  - `T defaultResult()`
  - `T aggregateResult(T aggregate, T nextResult)`
  - `boolean shouldVisitNextChild(RuleNode node, T currentResult)`
- Overriding some of these methods and using them can make parse trees much easier to deal with using Visitors

```
● ● ●
1  @Override
2  public T visitChildren(RuleNode node) {
3      T result = defaultResult();
4      int n = node.getChildCount();
5      for (int i=0; i<n; i++) {
6          if (!shouldVisitNextChild(node, result)) {
7              break;
8          }
9
10         ParseTree c = node.getChild(i);
11         T childResult = c.accept(this);
12         result = aggregateResult(result, childResult);
13     }
14
15     return result;
16 }
17
```

---

# “Humble” Advice

---



---

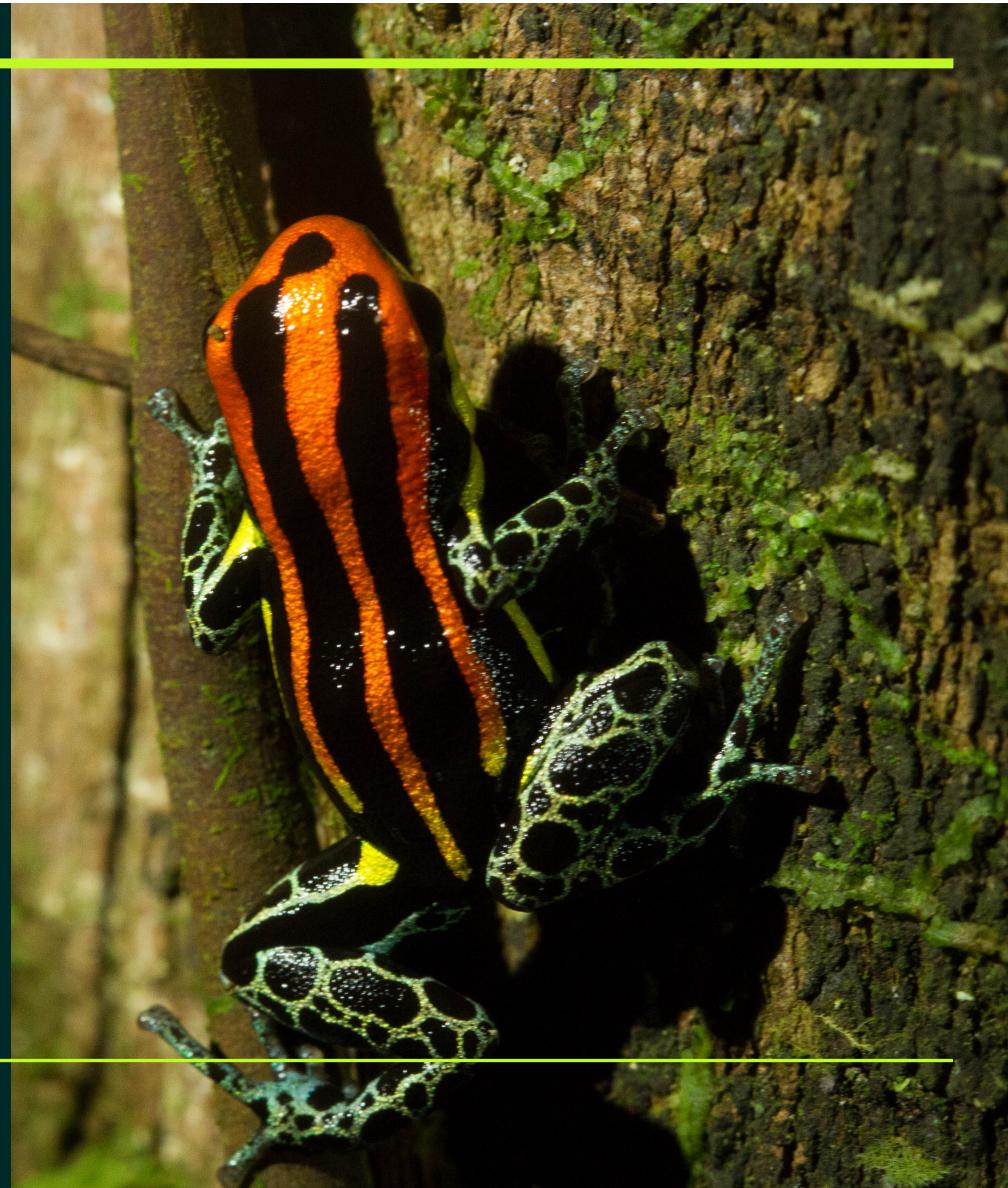
# Advice

- ▶ Don't try to put "all the things" into the grammar
    - ▶ Goal: an unambiguous and correct tree representation of your input
  - ▶ Understand the basics we've covered here
    - ▶ So many StackOverflow questions are users trying to just throw some stuff at ANTLR without "getting" the basics.
  - ▶ Put an **EOF** token at the end of all start rules.
    - ▶ without it, ANTLR will may recognize part of your input and stop parsing at the first unrecognized input (no error)
  - ▶ It's not wrong to have multiple "start rules".
  - ▶ Put a **BAD\_TOKEN: '.'**; lexer rule at the end of lexer grammar
  - ▶ Create a lexer rule for all tokens (generated lexer rules get horrible names)
    - ▶ Separate lexer and parser grammars and you'll get a warning if you specify an implied token.
    - ▶ It's OK to use the literals in your parser grammar if it makes it easier to read. They will be matched to the defined rule by ANTLR.
-

---

# Extra Credit

---



---

# Utility parsing

Sometimes you want to parse input just to extract certain content or to inject new content

- Once you have a parse tree you can extract specific content with `getText(Interval interval)`, `getText(RuleContext ctx)`, or `getText(Token start, Token stop)`
  - `TokenStreamRewriter` is quite powerful
    - methods to `delete`, `insertAfter`, `insertBefore`, `replace` (all with several signatures)
    - can keep up with multiple “`programNames`” (sets of changes)
    - can begin changes and `rollback` if necessary
    - `LittleCalcExecutionVisitor` uses this for some simple refactoring
-

---

# Parse Tree Matching and XPath

- ▶ <https://github.com/antlr/antlr4/blob/master/doc/tree-matching.md>
- ▶ Very powerful method of locating Context nodes that match a query
  - ▶ Parse Tree Patterns allow flexible matching of nodes against a query (using the same terms as your grammar)
  - ▶ XPath has a syntax similar to XML XPath and performs similar task of locating nodes in your tree.
  - ▶ Both are used in the **LittleCalcExecutionVisitor** refactoring sample code.

---

# Improving common error messages

- ▶ ANTLR's error messages are pretty good, but are necessarily limited by being a general purpose tool.
  - ▶ Cool Trick:
    - ▶ If:
      - ▶ Commonly observed syntax error
      - ▶ Default message from ANTLR is “not so useful”
      - ▶ You can write a parser rule to recognize that situation
    - ▶ Write the rule to recognize it, and use a visitor to create your own custom error message
-

# REPL

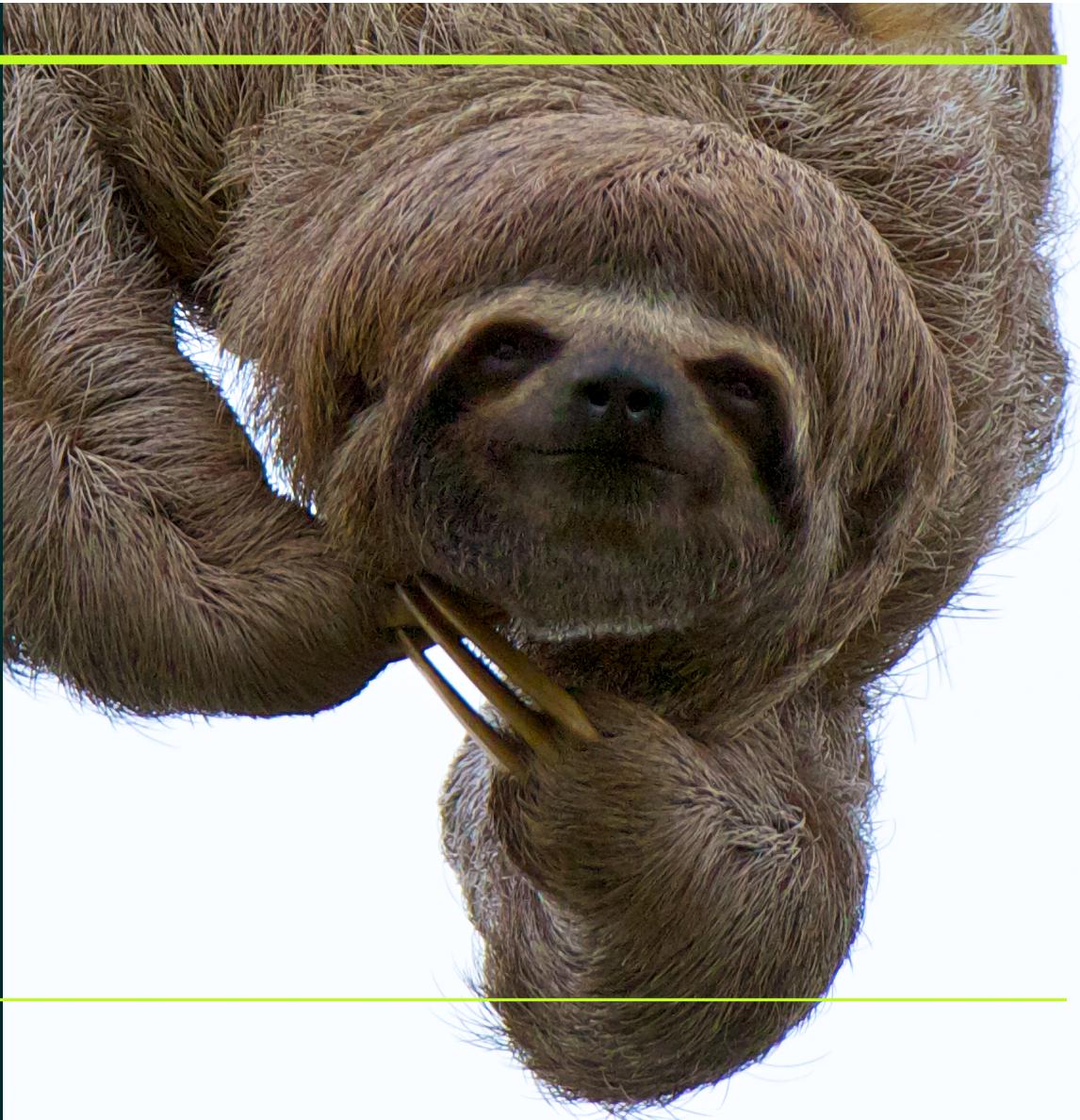
ANTLR default CharStreams read the entire contents before returning results

- ▶ LittleCalcREPL
  - ▶ Uses **JLine** project to create a terminal line reader
  - ▶ Each line is parsed on **Enter**
  - ▶ If input has a Syntax error at **EOF** (or ends with **\**) REPL requests another line and parses with accumulated input prepended
  - ▶ If no syntax errors are encountered then parse tree is validated with a listener, and if it encounters no errors, is evaluated by a visitor.
  - ▶ State is maintained in the REPL

---

# Questions?

---



---

# Attaching a Listener to a Parser

- ▶ Not a way to parse a continuing stream of input.
    - ▶ All methods that CharStreams.\* factory classes provide for obtaining a CharStream have the notation “Reads the entire contents of the file into the result before returning.”
      - ▶ So, nothing will happen until the input stream is closed.
      - ▶ \*Might\* get past this by writing your own CharStream class. but...
  - ▶ Listener methods are passed Context objects that refer to the entirety of their context.
    - ▶ The top-level context will be “everything” so you’re not releasing content you’ve already listened to.
  - ▶ Topic was brought up here: <https://stackoverflow.com/questions/14864777/using-antlr-for-parsing-data-from-never-ending-stream>
    - ▶ “ANTLR 4 has no guaranteed lookahead bound (and no way to tell it to look for or even attempt to enforce one), so any implementation that operates on a blocking stream has the possibility of deadlock without returning information about the parse leading up to that point” - Sam Harwell
  - ▶ NOT a way to listen in on terminal input and create a REPL (feel free to ask how I know this 😊)
  - ▶ Asked on ANTLR mailing list if there was any known use of this, and got “Crickets”
  - ▶ It **IS** used by ANTLR itself if you for “parser.setTrace(true)”
-