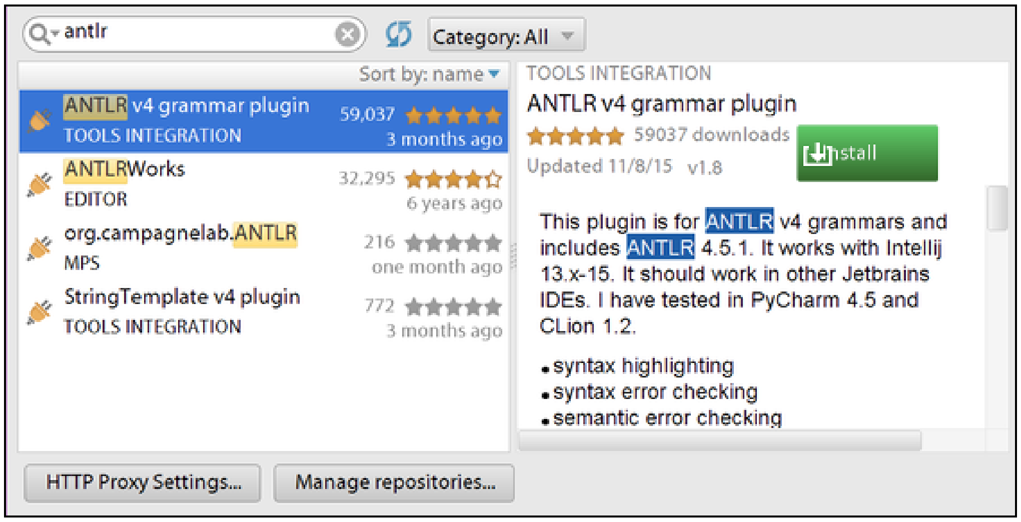
ANTLR

Christopher League[*]

17 February 2016

ANTLR is a **parser generator.** There are other similar tools, such as yacc, flex, bison, etc. We'll be using ANTLR **version 4.** Unfortunately it works fairly differently than version 3, so we need to be careful about which version we're seeing when seeking help or examples.

We can run ANTLR directly from IntelliJ IDE by installing its plugin. From the welcome screen, select **Configure** » **Plugins.** Search for ANTLR in the search box. It probably won't be found among the bundled plugins, but then you hit the **Browse** link and it should be the first match.



ANTLR v4 grammar plugin	59,037 ★★★★★	
TOOLS INTEGRATION	3 months ago	
ANTLRWorks	32,295 ★★★★☆	
EDITOR	6 years ago	
org.campagnelab.ANTLR	216 ★★★★★	
MPS	one month ago	
StringTemplate v4 plugin	772 ★★★★★	
TOOLS INTEGRATION	3 months ago	

TOOLS INTEGRATION
ANTLR v4 grammar plugin
★★★★★ 59037 downloads
Updated 11/8/15 v1.8

This plugin is for ANTLR v4 grammars and includes ANTLR 4.5.1. It works with Intellij 13.x-15. It should work in other Jetbrains IDEs. I have tested in PyCharm 4.5 and CLion 1.2.

- syntax highlighting
- syntax error checking
- semantic error checking

Figure 1: ANTLR plugin installer

Aside: *You can instead install ANTLR as a standalone tool that you run from the command line, or integrate with whatever build system you're using. Contact me if you need instructions for using it in other systems.*

Grammar previewer

Once you install the plugin and restart IntelliJ, new projects will have the "ANTLR Preview" tool in the lower left corner.

Let's begin a new project in IntelliJ for testing the capabilities of ANTLR and the plugin – I'll call it `antlrdemo`.

[*]Copyright 2016, some rights reserved (CC by-sa)

Figure 2: ANTLR preview tool

In the `src` directory, create a **New** » **File** and name it `Example.g4`. The extension `.g4` tells IntelliJ we're using version 4 of ANTLR, and this is called a "grammar file". We'll begin the grammar very simply:

```
grammar Example;   // Name of grammar, used to name generated Java classes

// Here we define a rule for the non-terminal `top`. It's an `expr`
// (expression, not yet defined) follow by built-in `EOF` token.
top : expr EOF            // First rule for `top` introduced by colon
    ;                     // End of rules for `top` indicated by semi-colon

expr : expr '+' expr      // Three different rules for `expr` non-terminal,
     | expr '*' expr      // subsequent rules introduced by `|`, read "OR".
     | NUM                // `NUM` is a terminal (token), not yet defined.
     ;
```

You can see that we're pretty much typing in a BNF grammar here. We don't have to worry that this grammar is left-recursive, and ANTLR has facilities for specifying precedence and associativity, which we'll examine in a moment.

We have yet to specify the definition of the `NUM` terminal for representing numbers. At the bottom of the file, add:

```
NUM : [0-9]+ ;
```

This uses **regular expression** syntax for a token. So the `[0-9]` means it looks for any (base 10) numeric digit, and then the plus sign + means there must be one or more of them. Every definition in ANTLR ends with a semi-colon.

With this grammar open, switch to the **Structure** view (Alt-7) on the left panel. You see it contains an entry for each of our rules. The `top` and `expr` are marked with a blue "P" because they are parser rules (non-terminals). The `NUM` is marked with a green "L" because they are lexical rules (terminals/tokens).

Right-click on the `top` item in the Structure view, and select **Test Rule top.** The previewer panel will open at the bottom. On the left is a big text box where we can type some sample code. On the right is a visualization of a parse tree. Beneath the tree there's a slider that can zoom in and out. Type `1+2+3` into the sample code box, as shown.

It constructs the tree as you type, and indicates any lexical or syntax errors beneath the code box. To see an error, insert a space into your expression: `1 +2+3`. It should say:

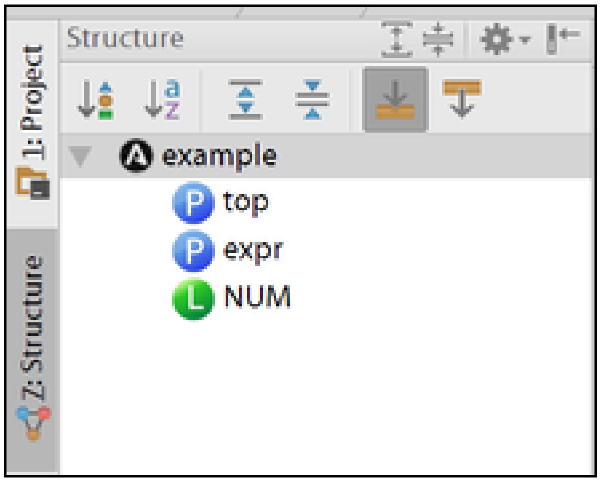
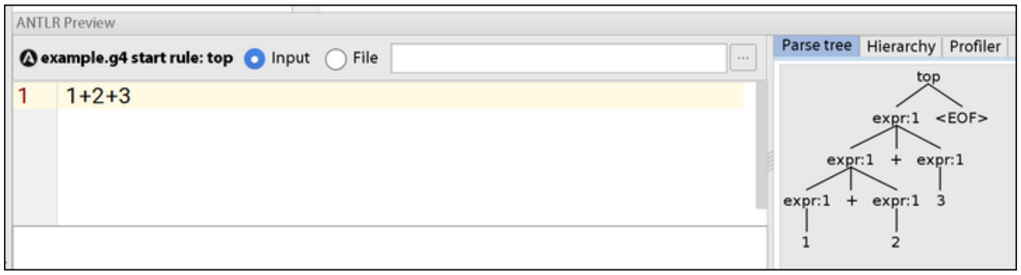


Figure 3: ANTLR structure view



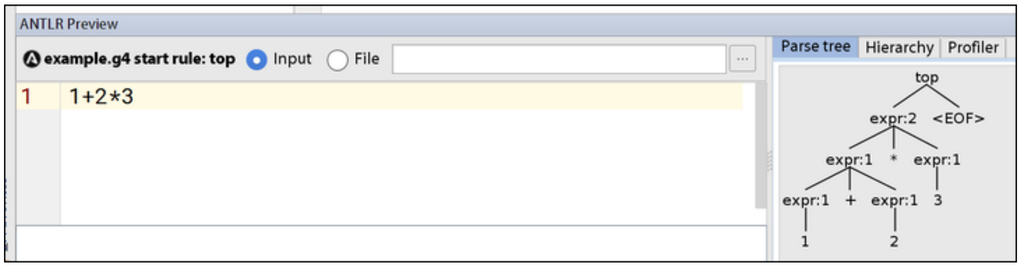
```
1+2+3
```

Figure 4: Preview parse tree for 1+2+3

```
line 1:1 token recognition error at: ' '
```

because we haven't specified in this grammar how to handle white space – we'll do that soon.

Now let's try an expression using both addition and multiplication: 1+2*3.



```
1+2*3
```

Figure 5: Preview parse tree for 1+2*3

We can immediately see from the parse tree that it doesn't support the correct precedence of the multiplication operator. We'll fix that next.

Precedence

The main way that precedence of operators is specified is by the order in which we write the rules. **Rules that appear earlier in the grammar file have higher precedence.** So we should put the rule for multiplication before the rule for addition:

```
expr : expr '*' expr
     | expr '+' expr
     | NUM
     ;
```

As soon as you make that change and save the file, the parse tree should update. If it doesn't, select **Test Rule top** again from the Structure view.

Now, what about supporting subtraction and division? We could attempt to add new rules for them:

```
expr : expr '*' expr
     | expr '/' expr
     | expr '+' expr
     | expr '-' expr
     | NUM
     ;
```

But the problem is that now multiplication has **higher** precedence than division. Usually we to treat *,/ as having the **same** precedence, which is higher than +,-. You

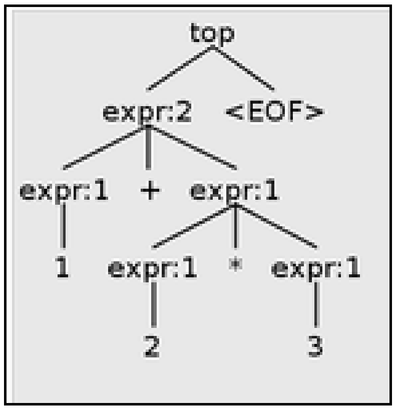


Figure 6: Better parse tree for 1+2*3

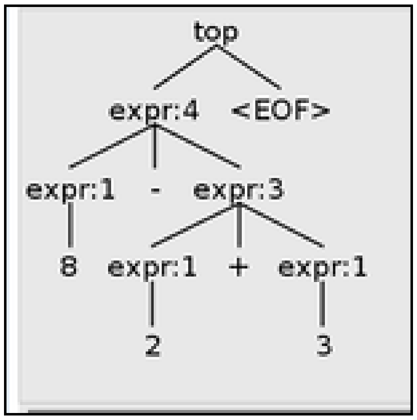


Figure 7: Incorrect parse tree for 8-2+3

can see we're not getting it right by testing the expression 8-2+3. It groups the addition together as if it were written 8-(2+3).

The fix for this is is to write operators with the same precedence as part of the same rule. We group them together using an **or** (|) syntax:

```
expr : expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | NUM
     ;
```

Now it can handle +,- at the same precedence level (left-associative) and multiplication and division are higher precedence.

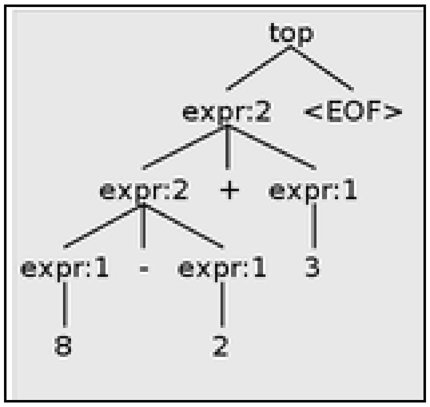


Figure 8: Correct parse tree for 8-2+3

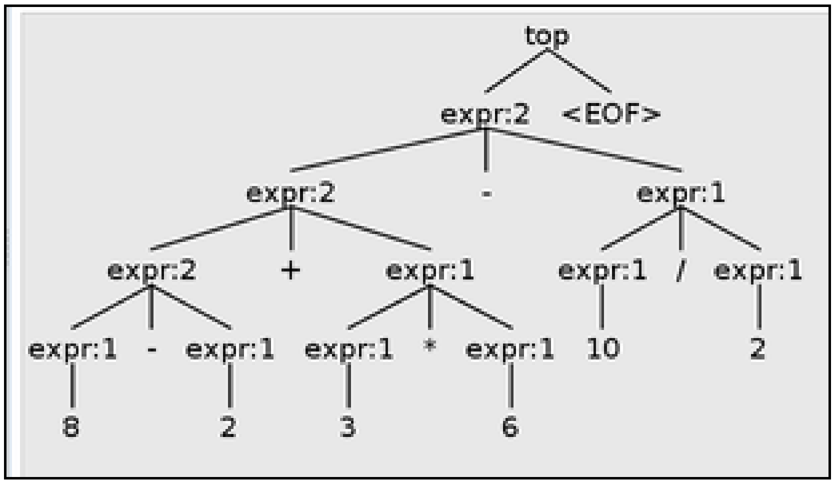


Figure 9: Correct parse tree for 8-2+3*6-10/2

In order to override the standard precedence rules we have encoded in this grammar, we will want to support **parentheses** in expressions. Right now, if we add parentheses, as in 2*(3+4), ANTLR flags them as errors:

```
line 1:2 token recognition error at: '('
line 1:6 token recognition error at: ')'
```

So we add a rule with explicit parentheses surrounding an expression:

```
expr : expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | '(' expr ')'
     | NUM
     ;
```

And now the addition happens first.

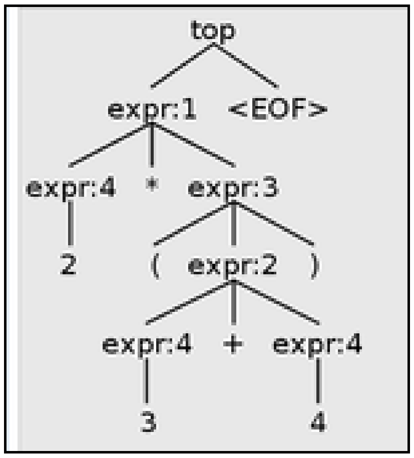


Figure 10: Correct parse tree for parenthesized expression `2*(3+4)`

Associativity

We've seen that, by default, ANTLR associates operators to the left. So `1-2-3-4` is parsed as if it were written `((1-2)-3)-4` rather than `1-(2-(3-4))`.

Some operators in programming languages are defined to be **right-associative.** A good example is the assignment operator in C, which you can chain together to initialize multiple variables:

```
a = b = c = 42;
```

This should be grouped as if it were written

```
a = (b = (c = 42));
```

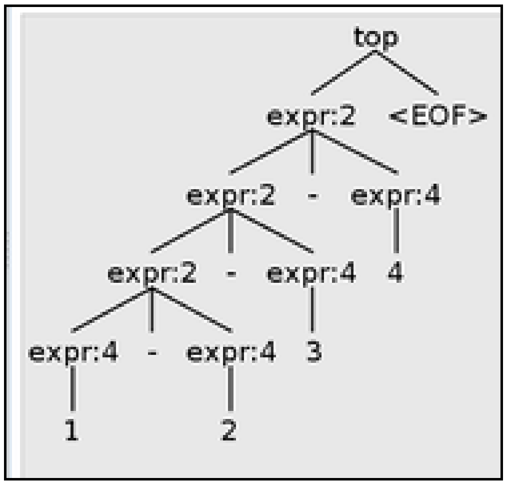


Figure 11: Left-associative tree for `1-2-3-4`

That is, it associates to the right. Another example is the list construction operator (aka "cons") in functional languages. In Haskell, it's written with a colon. The expression `x:xs` takes a list `xs` and attaches a single element `x` onto the front of that list. The empty list is written as `[]`, so we want to support repeated "cons" as:

```
2:3:5:8:13:[]
```

and have that associate to the right. To implement that in ANTLR, we just insert a metadata tag in angle brackets like this:

```
expr : <assoc=right> expr ':' expr
     | NUM
     | '[]'
     ;
```

And it will generate a right-branching tree for that operator.

Spaces and comments

Now, let's solve the issue with white-space. When we inserted spaces around our numbers or operators, it resulted in a syntax error. We need a way to tell ANTLR that our grammar should just ignore spaces. Fortunately, this is pretty simple. Just specify a lexical rule `WS` (near the bottom of the grammar file) with a regular expression for space characters.

```
WS : (' '|'\n'|'\t')+ -> skip;
```

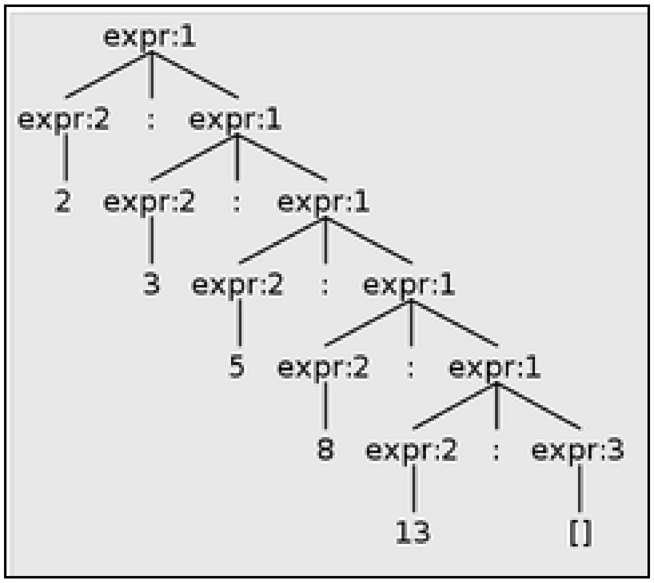


Figure 12: Right-associative list construction

and then we use the `-> skip` notation to say that these spaces should just be ignored. They can separate tokens, but they will not appear as terminals in the grammar.

In this definition, we said white-space is one or more space, newline, or tab characters. Now we can parse spaced-out expressions like:

```
1 +
  2 * 3
```

We also want the ANTLR lexer to ignore comments. The approach is similar: we specify a regular expression and then redirect the content to a "hidden channel" so it doesn't otherwise interfere with the grammar:

```
COMMENT: '#' .*? '\n' -> channel(HIDDEN);
```

The regular expression `.*?` deserves some explanation. The dot means "match any character", and then the asterisk means "zero or more of the previous thing"... so zero or more characters. Finally, the question mark means that the star operation is **non-greedy.** Normally, the star is greedy – it matches as many characters as possible so if there are several lines, it will stretch on to the **last possible newline.** We want the comment to end with the **first newline** it finds. So the non-greedy star does that.

```
1 +   # comment here
  2 * # multiply!
3
```

We can similarly support delimited comments '/*' like those in C `*/` and if we use a different token name then both styles can co-exist in the same language.

```
COMMENT2: '/*' .*? '*/' -> channel(HIDDEN);
```

Here's an example with three different comments:

```
1 +  # comment here
  2/*comment*/*3
  -4#+5
```

After squeezing out the white-space and comments we'd be left with `1+2*3-4`.

In some languages, delimited comments can be **nested** to any depth. This can be a helpful feature because it makes it easy to **comment out** code that already contains comments. If we try to do that in C (which doesn't support nested comments), we can run into problems:

```
int x = 5;
/* COMMENT OUT FOLLOWING THREE LINES
int y = 7;
int z = x+y; /* add numbers */
int k = z*2;
*/
int m = 9;
```

So the intent above was to comment out the declarations of `y`, `z`, **and** `k`, but it doesn't work. Due to the `*/` in the comment following the `z` declaration, the `k` is kept and the stray `*/` produces an error.

Functional languages tend to support nested comments, so SML uses `(*` these `*)` as its delimiters, and Haskell supports `{-` comments between these `-}`. It's easy to support nested comments in ANTLR by defining the `COMMENT` token recursively:

```
NESTCOMMENT : '{-' (NESTCOMMENT|.)*? '-}' -> channel(HIDDEN) ;
```

Main program

Now let's see how to invoke the ANTLR lexer and parser from a Java program. First we'll need to add the ANTLR libraries to our project. You can use **File** » **Project Structure** » **Libraries** and add the Maven library named `org.antlr:antlr4-runtime:4.5.2-1`.

Next, we tell IntelliJ to *run* ANTLR to generate Java classes from our grammar file. With the grammar still open, select **Tools** » **Generate ANTLR Recognizer** (Ctrl-Shift-G). In your Project view (Alt-1), you should now see a gen directory filled with

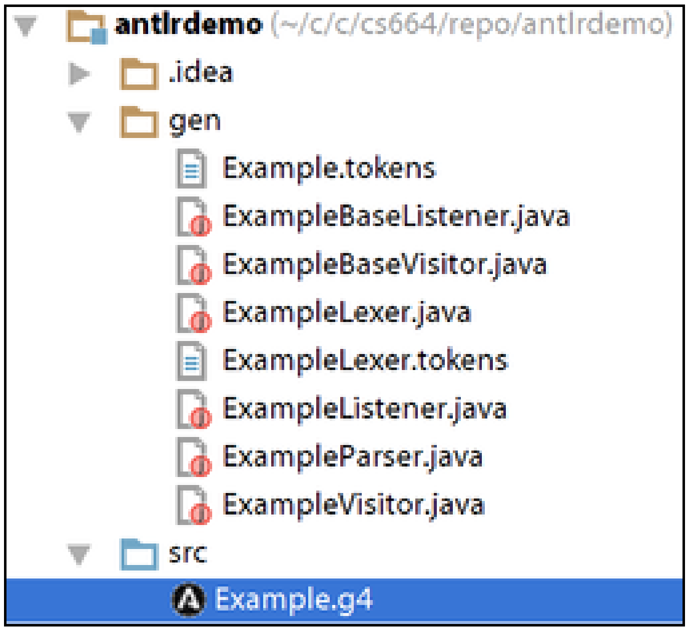


Figure 13: Generated classes in gen/

ANTLR-generated classes like `ExampleLexer` and `ExampleBaseVisitor`. (It uses the name `Example` from our `grammar` declaration at the top of the `.g4` file.)

The gen directory **should be marked as a source directory** in your project. You can tell because it will be light blue, just like `src`. If it's not, go to **File** » **Project Structure** » **Modules**, choose the gen directory from the project hierarchy in the center, click (**Mark as**) **Sources** and hit **OK.**

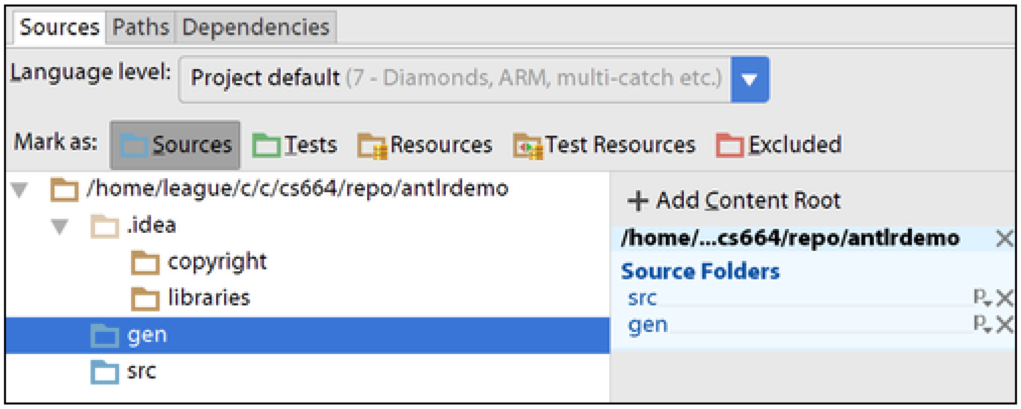


Figure 14: Mark gen as Sources

Now we're ready to create a main program to invoke ANTLR. Right-click on the `src` directory, choose **New** » **Java Class**, and provide the name `Main`. Then write the standard `main` method signature, so it looks like:

```
public class Main {
    public static void main(String[] args) {
    }
}
```

Unfortunately, there are quite a few steps to get all the objects working together. You might add this import statement to the top of your file, or let IntelliJ manage imports for you if you prefer:

```
import org.antlr.v4.runtime.ANTLRInputStream;
```

Then inside the `main` method, we need to create an input stream from which the Lexer will pull characters. The `ANTLRInputStream` class can read from a string, from an input stream (like `System.in`), or from a `Reader` (which is what I used in the PicoScript lexer). We'll start with just a string:

```
        ANTLRInputStream input = new ANTLRInputStream("6*4");
```

Now we provide that input stream to the generated lexer class:

```
        ExampleLexer lexer = new ExampleLexer(input);
```

The lexer generates tokens, but ANTLR wants us to wrap it in an another class that buffers them:

```
        CommonTokenStream tokens = new CommonTokenStream(lexer);
```

Now we can send the token stream to our generated parser:

```
        ExampleParser parser = new ExampleParser(tokens);
```

Finally we have a parser ready to use! The `parser` object has a method corresponding to each non-terminal in our grammar. So we can parse from the top level using `parser.top()` or we could select some other non-terminal like `parser.list()`.

```
        parser.top();
```

Run the `Main` program by selecting **Run** » **Run…** » **Main.** It should just finish with no complaints:

```
Process finished with exit code 0
```

that means it was able to parse our little program: `6*4`. But what if we embed an error in that program? Change the input string to just `6*`, then run it and you'll see:

```
line 1:2 no viable alternative at input '<EOF>'
```

Or how about `6^4`:

```
line 1:1 token recognition error at: '^'
line 1:2 extraneous input '4' expecting {<EOF>, '*', '/', '+', '-'}
```

So ANTLR is reporting its lexical and syntax errors. The complete program is reproduced here, in case you missed something in putting it together:

```
import org.antlr.v4.runtime.ANTLRInputStream;
import org.antlr.v4.runtime.CommonTokenStream;

public class Main {
    public static void main(String[] args)
    {
        ANTLRInputStream input = new ANTLRInputStream("6^4");
        ExampleLexer lexer = new ExampleLexer(input);
        CommonTokenStream tokens = new CommonTokenStream(lexer);
        ExampleParser parser = new ExampleParser(tokens);
        parser.top();
    }
}
```

Lexer precedence

In class, we defined a lexer rule for **signed** (positive or negative) floating-point numbers, like this:

```
NUM : '-'? [0-9]* ('.' [0-9]+)? ;
```

When generating the code from the grammar (**Tools** » **Generate ANTLR Recognizer**) it produces a warning about this rule:

```
warning(146): non-fragment lexer rule NUM can match the empty string
```

This is actually a warning to be taken seriously! When I was trying to use the previewer, it actually went into an infinite loop and I had to kill the entire IntelliJ session. When it can generate a token from the empty string, it can end up generating infinitely many of them without consuming any input.

The core of the problem is the `*` operator in the middle. It means "zero or more", applied to the digits `[0-9]`. But with optional (`?`) things on either side of that, we can have the empty string be a `NUM`! Really, there should be **at least one** digit in a `NUM` so changing the `*` to a `+` (meaning "one or more" digits) will eliminate the warning.

```
NUM : '-'? [0-9]+ ('.' [0-9]+)? ;
```

But there's still another issue using a minus sign as part of the NUM token. It can be difficult, at the lexical level, to distinguish between the subtraction operator and a negative number. To see the problem, we need to understand how ANTLR determines which lexical rules to use when there are multiple possibilities.

Consider the arithmetic expression 5-2. How do we tokenize it? Is it NUM(5) followed by NUM(-2)? Or is it NUM(5) SUB NUM(2), where SUB represents the token for the subtraction operator. Perhaps only one of those will be valid in the parser, but it doesn't matter. The lexer is making its decisions about how to tokenize these before the parser sees them.

The rule that ANTLR uses is a **greedy** one: it matches the *longest possible token.* Because the token NUM(-2) matches two characters and SUB matches only one character, the NUM wins. But it means that 5-2 now has a parse error!

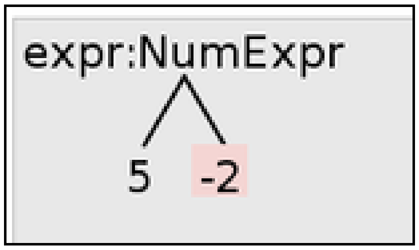


Figure 15: Subtracting 5-2 doesn't work because it sees NUM(5) NUM(-2)

In distinguishing subtraction from negative numbers, the language Standard ML was defined to use the typical dash character - for subtraction, but requires the **tilde** character ~ for negative numbers. That completely removes the ambiguity, but it means programmers have to type strange-looking numbers like ~6.28.

To support the more typical C++/Java syntax which uses the same character, follow this suggestion from the ANTLR mailing list:

> Don't bother trying to handle negative numbers in the lexer, handle unary-minus in the parser instead.

So we remove the subtraction from the NUM rule:

```
NUM : [0-9]+ ('.' [0-9]+)? ;
```

and add an alternative to expr to support negative numbers like this:

```
expr : // previous definitions go here
     | '-' NUM
     ;
```

Now subtraction works properly, and negative numbers too. You can even subtract a negative number with just `5--2` (read "five minus negative two"). That doesn't work in C++/Java because `--` is its own distinct operator in those languages.

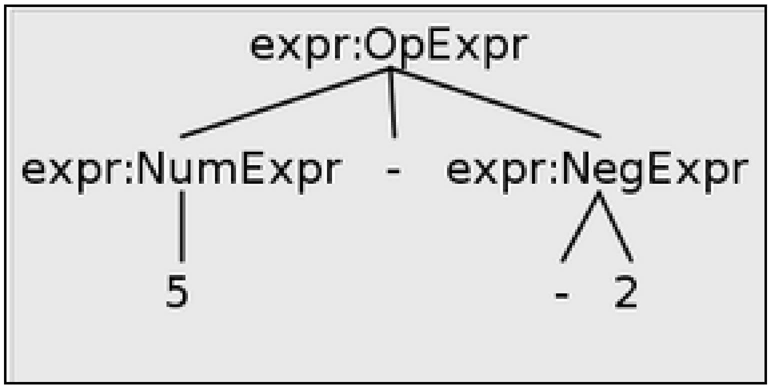


Figure 16: Both subtraction and negation in the same tree

Now that we're supporting a unary minus, it would be nice if it could also negate other expressions like `-(7*2)`. This doesn't currently work because the negation alternative is defined as `'-' NUM`, but we could change it to allow negating any expression.

```
expr : // previous defined go here
     | `-` expr
     ;
```

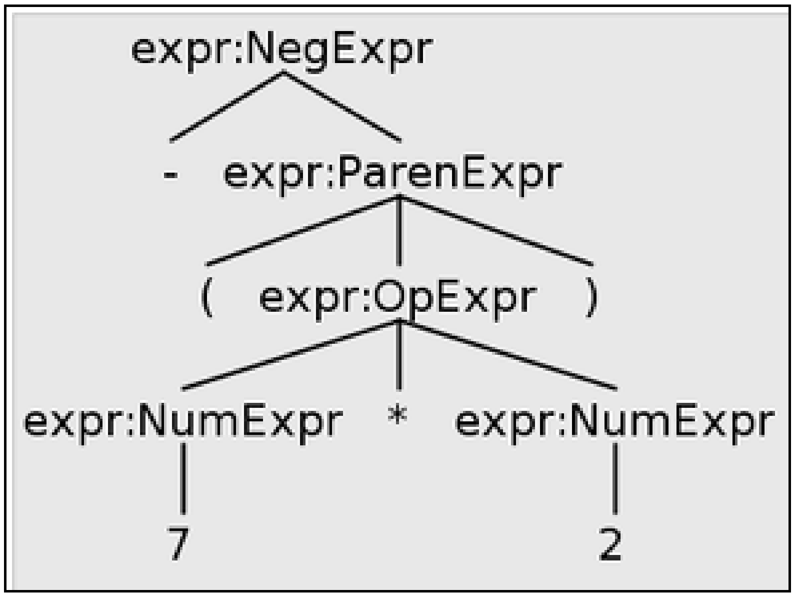


Figure 17: Negating a parenthesized expression: `-(7*2)`

Unfortunately, we're not quite there yet. Because if we remove the parentheses, we find out the order of operations is all wrong: in `-7*2` it actually groups the multiplication together first, and then applies the negation – as if it were written `-(7*2)` with parentheses anyway. You might say it doesn't matter much, because both expressions evaluate to `-14`. But what if it's addition? `-7+2` should evaluate to `-5` and `-(7+2)` to `-9`.

Negation should have **higher** precedence than addition or multiplication. So we'll add it into our expression grammar above any other operators:

```
expr : '-' expr
     | expr ('*'|'/') expr
     | expr ('+'|'-') expr
     | '(' expr ')'
     | NUM
     ;
```

Extended BNF syntax

Regular expression syntax