ANTLR 4

Introduction

ANTLR is a tool used to generate/process/translate parsers. The parser is output as code that can be interfaced with from your own code. In addition, ANTLR provides features that help with debugging/visualizing the parser and handling parse errors.

The type of parsers that ANTLR generates are top-down recursive descent parsers called ALL(*). The A is short for Adaptive and LL(*) refers to a style of parser that parses left-to-right and has the ability to look ahead by an infinite number of tokens. A discussion on parser types

is out of scope for this document, but the general idea behind ALL(*) is that it enables you to design/write grammars with less effort and fewer restrictions.

> **NOTE**: See https://github.com/antlr/antlr4/blob/master/doc/faq/general.md.

> **NOTE**: There are a lot of new terms here. They're discussed further in this document / can be looked up in the Terminology section.

Setup

There are a couple of different ways to interface with ANTLR -- you need to have Java installed for all of these.

Command-line

Setup command-line access by doing the following...

```
$ cd /usr/local/lib
$ wget http://www.antlr.org/download/antlr-4.7.1-complete.jar
$ export CLASSPATH=".:/usr/local/lib/antlr-4.7.1-complete.jar:$CLASSPATH"
$ alias antlr4='java -jar /usr/local/lib/antlr-4.7.1-complete.jar'
$ alias grun='java org.antlr.v4.gui.TestRig'
```

The above code downloads a JAR and adds it to the global Java CLASSPATH, then sets up some shell aliases to run the basic ANTLR commands. You can move the export/alias lines to .bashrc if you want to make these permanent.

The aliases are...

- antlr4 → compiles your grammars into code (e.g. Java source code)
- grun → tool for testing grammars

Java

The antlr4 and grun shell aliases setup above can also be called programmatically in Java. First, load up the ANTLR dependency…

```
<dependencies>
    <dependency>
        <groupId>org.antlr</groupId>
        <artifactId>antlr4</artifactId>
        <version>4.7.1</version>
    </dependency>
</dependencies>
```

To call the ANTLR tool (antlr4 alias), you can invoke org.antlr.v4.Tool.main()...

```
Tool.main(args);
```

To call the TestRig tool (grun alias), you can invoke org.antlr.v4.gui.TestRig.main()...

```
TestRig.main(args);
```

> **NOTE**: If you want to know the process that Tool.main goes through to create the output, read the source for Tool.processGrammarsOnCommandLine().

There is also a plugin available for Maven that lets you automatically convert your grammars to Java files…

```
<build>
    <plugins>
        <plugin>
            <groupId>org.antlr</groupId>
            <artifactId>antlr4-maven-plugin</artifactId>
            <version>4.7.1</version>
            <executions>
                <execution>
                    <id>antlr</id>
                    <phase>generate-sources</phase>
                    <goals>
                        <goal>antlr4</goal>
                    </goals>
                </execution>
            </executions>
        </plugin>
    </plugins>
</build>
```

> **NOTE**: Your grammar (g4 extension) files go under src/main/antlr4. The folder hierarchy defines the package that the generated Java files use… for example src/main/antlr4/org/company/product/Test.g4 would generate Java sources for the package org.company.product.

> **NOTE**: If a project uses this, remember that you need to do a build to get the generated antlr Java files once you open up your project in a IDE. Otherwise it'll show a bunch of Java parsing errors.

Terminology

- **syntax**
  Syntax is the structure of the rules that make up a language.
- **semantics**
  Semantics are the meaning of the rules that make up a language.
- **grammar**
  A grammar is a formal declaration of a language's syntax.
- **parser**
  A parser is a program that takes a raw input and converts it into some structured format (e.g. abstract syntax tree). For programming languages, this typically involves generating an abstract syntax tree from a grammar.
- **abstract syntax tree**
  Abstract syntax trees (also called ASTs) are the tree representation of some parsed input, typically for a programming language.
- **top-down parsing**
  Top-down parsing is a parsing strategy that starts from the top-most rule and works its way down to child rules.
- **recursive descent parsing**
  Recursive descent parsing is a form of top-down parsing where each rule is a method that recurses into another rule/method.
- **look-ahead token**
  Look-ahead tokens are tokens ahead of the current token being processed. Look-ahead tokens are used to determine which sub-rule a rule should drill down into. This is discussed further in the Alternatives section.
- **left-recursive rule**
  A parsing rule that invokes itself at the beginning if an alternative (see the Defining Grammar section for more information). Left-recursive rules are known to be problematic when generating parsers.
- **island grammars**
  Island grammars are grammars for mini-languages that are included as part of a main language. For example, if you're parsing C code may need an extra grammar for the preprocessor. Another example is Javadoc tags inside of comments in of a Java source file.
- **lexical modes**
  Lexical modes are a feature to help deal with mixed grammar formats (e.g. island grammars). A special sentinel lexing rule is used to automatically switch between formats. For example, if a pound sign (#) is detected, the lexical mode may switch to parsing C preprocessor code.

- **scannerless parsers**
  Scannerless parsers are parsers that they don't define complex lexing rules to derive tokens. They use standard characters as tokens.
- **tail recursion**
  ANTLR supports looping via the * and + quantifiers (see the Quantifiers sections for more information), but some other parser generators don't. So simulate looping, they use a technique called tail-recursion. For example..

  r: (ID ',')* ID;

  would get re-written as…

  r: ID | ID ',' r;

  Notice that the re-written version has 2 alternatives. The first alternative takes in an ID while the second alternative takes in an ID then a comma then recurses into itself. The fact that it's recursing into itself at the end is what makes it "tail recursion".

Concepts

The core concepts when working with ANTLR...

- syntax → structure of the rules that make up a language.
- semantics → meaning of the rules that make up a language.
- grammar → declaration of a language's syntax.
- parser → program that recognizes some language.

Languages have a syntax, and that syntax is specified using a grammar. A grammar is what gets used to create the parser. When a parser reads in a program, the operations it performs will conform to the semantics of the language it's parsing.

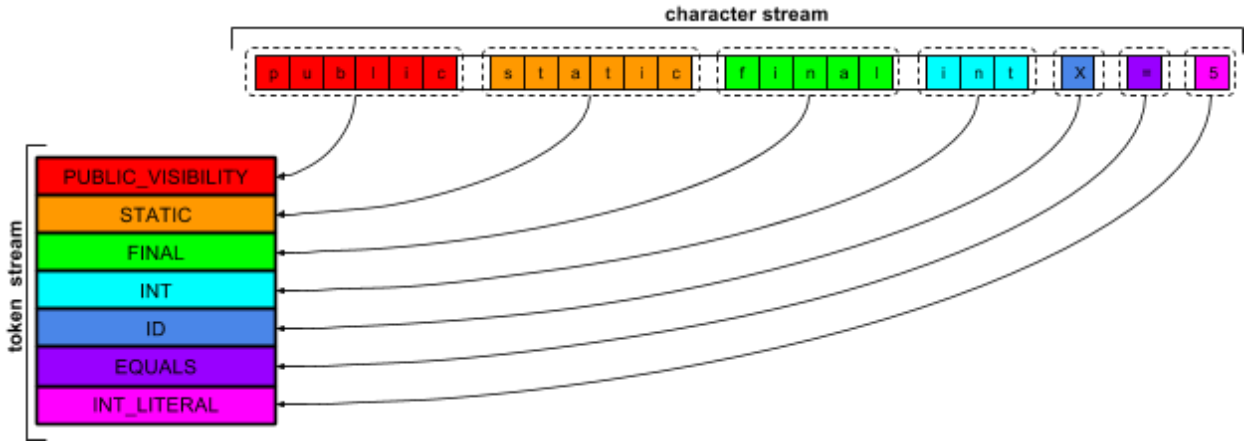
For example, if the parser comes across an if statement, it will evaluate it and possibly skip over the instructions nested within it if it evaluates to false.

> **NOTE**: The example above is specific to building interpreters. Semantics may not come into play if you're doing other than interpreting.

ANTLR's grammar is a language in and of itself -- it's a language for specifying other languages. Other tools have their own grammar languages. But, most tools process a grammar by reading in a stream of characters and breaking them up in 2 distinct steps: lexical analysis and producing an syntax tree.

Lexical Analysis

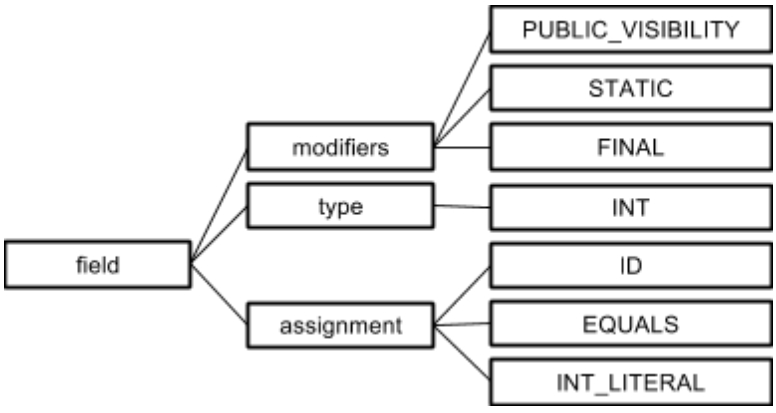
Lexical analysis (also called lexing or tokenization) is the process of breaking up a character stream into higher-level tokens. For example...



These tokens are then used to match the higher-level syntax rules.

Syntax Tree

Syntax trees (also called abstract syntax trees or ASTs) are trees that represent a hierarchical view of the input that was tokenized. These trees are formed based on the grammar rules using a top-down approach. For example...



Notice how all the leaf nodes in the example tree are tokens (uppercase). The leaf nodes of a syntax tree will always be tokens, while higher level nodes are the rules that eventually led to those tokens (lowercase).

Once you have an AST for your input, you can easily build tools such as...

- compilers → reducing the instructions into a more primitive set of instructions.
- interpreter → executing commands as the program is being parsed.

- translator → converting to another language as the program is being parsed.

Defining Grammars

According to the ANTLR4 book, there are 4 top-level patterns for designing grammars…

- Sequence → a sequence of elements.
  (see Sequence and Quantifier subsections)
- Choice → branching point to support multiple different variants of a rule.
  (see Alternatives subsection)
- Token dependence → presence of one token requires the presence of another token.
  (see Sequence subsection -- e.g. matching braces)
- Nested phrase → essentially recursive grammar rules?
  (see Recursive subsection)

There 4 patterns are the basis of ANTLR's grammar format, as well as other grammar formats such as Backaus-Naur form (BNF) and its variants. In addition to these 4 patterns, ANTLR also provides some extra grammar features to help streamline things even further (e.g. sub-rules).

An ANTLR grammar consists of a header that names the grammar, followed by a set of syntax parsing rules, followed by a set of lexing rules. For example...

```
grammar MyG;

start_rule: rule1 | rule2;
rule1: STUFF;
rule2: NUMS;

STUFF: [a-z]+;
NUMS: [0-9]+
```

Both the lexical analysis and syntax tree generation are done using this grammar file. Rules that start with…

- uppercase are lexing/tokenizing rules -- these are put at the top of the grammar file
- lowercase are syntax tree parsing rules -- these are put at the bottom of the grammar file

Lexing rules are put in at the bottom of the grammar file, while syntax tree rules are put at the top.

> **NOTE**: In many cases, it may be ambiguous as to which token is chosen by the lexer (the input may match multiple lexing rules). See the Ambiguities section for how this gets resolved.

Lexer

More indepth documentation can be found at https://github.com/antlr/antlr4/blob/master/doc/lexer-rules.md.

Characters

Characters that make up a lexer rule can be…

- normal characters / string literals → 'a' 'b' 'abc' 'xyz1234'
- escape characters → \r \n \t \p{...} \\
- unicode characters → \u000D
- wildcard character → . (matches any character)

Character Classes

In addition to string literals, ANTLR has support for character classes. Character classes in ANTLR are similar to character classes in regex. You can specify…

- ranges → 'a'..'z'
- range sets → ('a'..'z'|'A'..'Z')
- regex sets → [aBC] or [\n\r\t] or or [a-zA-Z0-9\u1234] or [\p{White_Space}]
- not (tilde) → ~[a-z] or ~('a'..'z')

Quantifiers

When specifying your rule, you can add quantifiers to the contents of your lexer rule...

- + → one or more elements
- * → zero or more elements
- ? → zero or one elements (the element is optional)

+ and * quantifiers are greedy by default. To make them non-greedy, add a question mark: +? or *?.

> **NOTE**: These are very similar to regex quantifiers.

Examples are as follows...

```
DATA: 'abc'+;
OTHER_DATA: 'abc' .*? 'abc';
ID: [a-zA-Z]+;
INT: [0-9]+;
```

Sequences

A lexer rule can be comprised of a sequence of characters, character literals, and character sets. For example…

```
MALENAME: 'Mr.' [A-Z][a-z]+
```

> **NOTE**: Parser rules also have sequences. Unlike parsing rules, you're parsing a single token per lexer rule. There is no breaking down what you get back..

Fragments

A lexer rule can be a fragment for use in other lexing rule. Fragment are just like regular lexing rules, except that they…

- intended to be used as part of another lexing rule.
- won't be parsed as a lexing rule on its own.

For example…

```
MALENAME: 'Mr.' NAME;
FEMALENAME: 'Mrs.' NAME;
fragment NAME: 'A'..'Z' 'a'-z'+;
```

Sub-rules

A lexer rule can define zero or more sub-rules. Sub-rules are basically just another rule (group of symbols) defined directly in the main rule.

Sub-rules are defined by wrapping symbols in brackets. For example, if we wanted to have a rule that matched a sequence of numbers where each number started with abc ...

```
OPERATOR: ('abc' INT)+;
```

> **NOTE**: Parser rules also have subrules. Unlike parsing rules, you're parsing a single token per lexer rule. There is no breaking down what you get back..

Alternatives

When there's more than one choice/path for a single lexer rule, this is called an alternative. Anytime you think to yourself that a language construct is 'either this or that', alternatives are what you should use.

Alternatives are expressed using a pipe (|). For example if we wanted a single rule for defining a variable type following by a number (e.g. int5 or string91), alternatives would likely be used…

```
TYPE: ('float' | 'int' | 'void') INT;
```

> **NOTE**: Parser rules also have alternatives. Unlike parsing rules, you're parsing a single token per lexer rule even though there are multiple choices. There is no breaking down what you get back.

Recursion

You can make rules that reference themselves. For example…

```
stat: ID;

ID: 'z' ('a' | ID+);
```

For the input zzzzzza…

> **NOTE**: Parser rules also have recursion. Unlike parsing rules, you're parsing a single token per lexer rule. There is no breaking down what you get back..

There are some very important gotchas when it comes to recursion. The same gotchas for parsing show up for lexing. See the Recursion gotchas section for more information.

Ignore Tokens

You can skip a lexing rule by using the -> skip directive. This directive reads in the token but doesn't return it to the parser -- it throws it out. This is typically used for skipping whitespace. For example…

```
ID: [a-zA-Z]+;           // ids
WS: [ \t\r\n]+ -> skip;   // toss out whitespace
```

> **NOTE**: The above example skips whitespace. If the work you're doing with your parser needs the original whitespace (or whatever you're skipping), you can use channels instead. See the Channels section below for more information.

Channels

You can assign tokens to specific channels using the -> channel(name) directive. Tokens will still be parsed, but the parser these tokens are fed into will only process tokens for a specific channel. For example…

```
WS: [ \t\r\n]+ -> channel(HIDDEN);   // whitespace to HIDDEN channel
```

By default, 2 channels are available: DEFAULT_TOKEN_CHANNEL and HIDDEN. If you're going to feed into a channel other than the these 2 defaults, you need to explicitly declare the channel first. For example…

```
channels { MYCHANNEL1, MYCHANNEL2 }
WS: [ \t\r\n]+ -> channel(MYCHANNEL1);// whitespace to MYCHANNEL1 channel
```

> **NOTE**: The most common usecase for this feature is retaining the original whitespace but not actually parsing it. See the Retaining Whitespace gotcha section for more information.

Lexical Modes

Lexical modes are a feature to help deal with mixed grammar formats (e.g. island grammars). Sentinel rules are used to automatically switch between formats.

To define new a lexical mode, append a MODE name directive followed by the new lexing rule to the end of the grammar. To switch to the new mode, use the -> pushMode(name) directive. To switch out of the new mode, use the -> popMode directive. For example...

```
JAVADOC_START: '/**' -> pushMode(JAVADOC);

MODE JAVADOC;
TAG: '@' [a-z]+
END_COMMENT: '*/' -> popMode;
```

Parser

More indepth parsing rules can be found at https://github.com/antlr/antlr4/blob/master/doc/parser-rules.md.

Sequences

A rule can be comprised of a sequence of other rules (can be both lexing rules or parsing rules). For example, imagine we had a rule for declaring int and initializing it to some literal…

```
type: 'int';
int_var: type ID '=' INT ';';

ID: [a-z]+;
INT: [0-9]+;
```

The rule int_var in the above example is a rule that's comprised of a sequence of other rules.

Quantifiers

When specifying your rule, you can add quantifiers to the contents of your rule...

- + → one or more elements
- * → zero or more elements
- ? → zero or one elements (the element is optional)

> **NOTE**: These are very similar to regex quantifiers.

Examples are as follows...

```
number : INT+;
name : 'name' WORD?;
```

Sub-rules

A rule can define zero or more sub-rules. Sub-rules are basically just another rule (group of symbols) defined directly in the main rule.

Sub-rules are defined by wrapping symbols in brackets. For example, if we wanted to have a rule that takes in one or more emails separated by a comma, a subrule can be used...

```
emails :  (email ',')* email;
```

> **NOTE**: Note the * quantifier on the sub-rule.

In many cases, you want your sub-rule to be labelled. When you execute your grammar, labeling allows you to target the sub-rule directly. Labels are specified using a equals (=). For example, if we wanted to label each alternative in the example above…

```
expr: INT op=('*'|'/'|'+'|'-') INT;
```

```
@Override
public Integer visitExpr(EvalSimpleParser.ExprContext ctx) {
    int left = visit(ctx.INT(0).getText());
    int right = visit(ctx.INT(1).getText());
    switch (ctx.op.getType()) {
        case EvalSimpleParser.ADD:
            ...;
        case EvalSimpleParser.SUB:
            ...;
        case EvalSimpleParser.MUL:
            ...;
```

```
        case EvalSimpleParser.DIV:
            ...;
        default:
            throw new IllegalStateException();
    }
}
```

> **NOTE**: See the Executing grammars section for more info on executing grammars, but ultimately what this does is allow you to directly target the sub-rule as a field. In this case, the sub-rule can be accessed using the name op.

Alternatives

When there's more than one choice/path for a single rule, this is called an alternative. Anytime you think to yourself that a language construct is 'either this or that', alternatives are what you should use.

Alternatives are expressed using a pipe (|). For example if we wanted a single rule for when a type is defined in some C-style language, alternatives would likely be used…

```
type: 'float' | 'int' | 'void';
```

> **NOTE**: In many cases, it may be ambiguous as to which alternative gets chosen (the input may match both alternatives). See the Ambiguities section for how this gets resolved.

In many cases, you want your alternatives to be labelled. When you execute your grammar, labeling allows you to identify which alternative you're in. Labels are specified using a pound (#). For example, if we wanted to label each alternative in the example above…

```
type: 'float'      # FloatType
        | 'int'    # IntType
        | 'void'   # VoidType
        ;
```

```
@Override
public Integer visitVoidType(EvalSimpleParser.VoidTypeContext ctx) {
    ...
}

@Override
public Integer visitIntType(EvalSimpleParser.IntTypeContext ctx) {
    ...
```

```
}

@Override
public Integer visitFloatType(EvalSimpleParser.FloatTypeContext ctx) {
    ...
}
```

> **NOTE**: See the Executing grammars section for more info on executing grammars, but ultimately what this does is generate a unique event (method invocation) for each alternative labelled.

Recursion

You can make rules that recurse into themselves. For example...

```
stat: block | ID;
block: '{' stat* '}';

ID: [A-Za-z0-9]+;
WS: [ \t\r\n]+ -> skip;
```

There are some very important gotchas when it comes to recursion, specifically left recursion. Left recursion is when the rule recurses at the first element / leftmost element. ANTLR supports left recursion so long as the the recursion...

1. isn't indirect left recursion -- can only recurse into itself directly.
2. fits into one of 4 predefined patterns -- avoids infinitely recursing into itself.

See the Recursion gotcha sections for more information.

Imports

Grammars can be split up into different files. Just like with Java files, you can import another grammar into your main grammar. For example, imagine that we had the following grammar…

```
lexer grammar CommonGrammar; // a grammar of only lexing rules

ID: [a-z]+;
INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

That grammar can then be imported into other grammars using an import statement...

```
grammar MainGrammar;
```

```
import CommonGrammar; // import grammar

main: ID '=' INT;
```

> **NOTE**: When you compile your grammar to Java source files, you only have to specify the main grammar file. The imported grammars will automatically get pulled in.

Executing Grammars

The syntax tree for your input gets generated via the Parser class that ANTLR creates for your grammar. This parser class notifies you (the user) of the pieces of the syntax tree as it walks over it -- very similar to SAX-style XML parsing.

Inside the Parser class, each leaf node (token) is a TerminalNode class while higher-level nodes are subclasses of ParserRuleContext. Instances of these objects get passed to you as the parser class walks the tree. They contain information about what was matched. And, in the case of ParserRuleContext objects, allow further drilling into sub-rules that the rule matched.

Imagine the following grammar...

```
grammar Hello;
r: 'hello'ID;
ID: [a-z]+;
WS: [\t\r\n]+ -> skip;
```

This will be the output once this grammar is compiled down to Java source files...

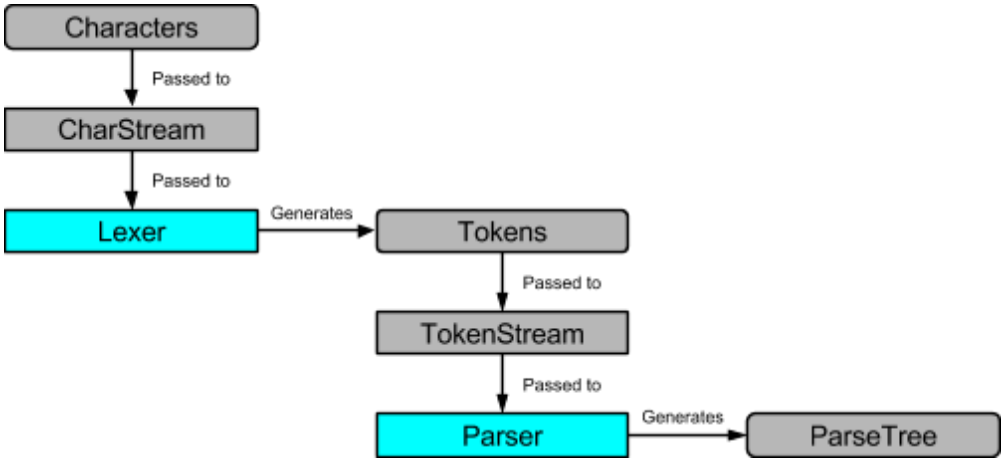
```
$ ls -1
HelloBaseListener.java
Hello.g4
Hello.interp
HelloLexer.interp
HelloLexer.java     <-- THIS IS THE LEXER CLASS FOR THIS GRAMMAR
HelloLexer.tokens
HelloListener.java
HelloParser.java    <-- THIS IS THE PARSER CLASS FOR THIS GRAMMAR
Hello.token
$ cat HelloParser.java | grep TerminalNode
        public TerminalNode ID() { return getToken(HelloParser.ID, 0); }
$ cat HelloParser.java | grep RContext
      public static class RContext extends ParserRuleContext {
            public RContext(ParserRuleContext parent, int invokingState) {
```

```
    public final RContext r() throws RecognitionException {
        RContext _localctx = new RContext(_ctx, getState());
```

Notice that the…

- token ID gets returned as a TerminalNode object.
- rule r gets returned as a RContext object (which inherits from ParseRuleContext).

There are 2 types of SAX-style parsing that ANTLR provides: listeners and visitors. Regardless of which you use, the basic setup is the same. You create a CharStream from the input, pass that to the Lexer to get a TokenStream, then pass that a Parser to walk the tree...




```
CharStream charStream = CharStreams.fromFile("Test.java");
Java9Lexer lexer = new Java9Lexer(charStream);
TokenStream tokenStream = new CommonTokenStream(lexer);
Java9Parser parser = new Java9Parser(tokenStream);

ParseTree tree = parser.compilationUnit(); // top-level rule
```

Listeners

By default, ANTLR will generate listener interfaces that it invokes as it walks the tree.

Imagine the following grammar…

```
grammar Hello;

r: 'hello' names;
names: (ID',')* ID;

ID: [a-z]+;
```

```
WS: [\t\r\n]+ -> skip;
```

The following code walks a sample input for this grammar via a Listener…

```
public static void main(String[] args) {
    CharStream charStream = CharStreams.fromString("hello ti, ann");

    HelloLexer lexer = new HelloLexer(charStream); // extends Lexer
    TokenStream tokenStream = new CommonTokenStream(lexer);
    HelloParser parser = new HelloParser(tokenStream);

    ParseTree parseTree = parser.r(); // top rule -- extends ParseTree

    ParseTreeWalker walker = new ParseTreeWalker();
    HelloListener listener = new HelloListener() {
        private String indent = "";

        @Override
        public void enterR(HelloParser.RContext ctx) {
            System.out.println(indent + "Enter r");
            indent += " ";
        }

        @Override
        public void exitR(HelloParser.RContext ctx) {
            indent = indent.substring(1);
            System.out.println(indent + "Exit r");
        }

        @Override
        public void enterNames(HelloParser.NamesContext ctx) {
            System.out.println(indent + "Enter names");
            indent += " ";
        }

        @Override
        public void exitNames(HelloParser.NamesContext ctx) {
            indent = indent.substring(1);
            System.out.println(indent + "Exit names");
        }

        @Override
```

```
        public void visitTerminal(TerminalNode node) {
            System.out.println(indent + "Token: " + node.getText());
        }

        @Override
        public void visitErrorNode(ErrorNode node) { /* ??? */ }

        @Override
        public void enterEveryRule(ParserRuleContext ctx) { /* ??? */ }

        @Override
        public void exitEveryRule(ParserRuleContext ctx) { /* ??? */ }
     };

    walker.walk(listener, parseTree);
}
```

NOTE: This example implements the HelloListener interface directly, but there's also a BaseHelloListener class that you can use if you don't plan on implementing all the interface methods.

NOTE: Classes prefixed by Hello are classes that ANTLR generated for the grammar.

Output is...

```
line 1:5 token recognition error at: ' '
line 1:9 token recognition error at: ' '
Enter r
 Token: hello
 Enter names
  Token: ti
  Token: ,
  Token: ann
 Exit names
Exit r
```

Visitors

ANTLR will generate a visitor (in addition to a listener) if you explicitly tell it to. The main difference between a listener and a visitor is that a visitor gives the user explicit control of when and where to drill down.

To have ANTLR generate a visitor as well as a listener, use the…

- -visitor flag when running the antlr4 shell alias...
  antlr4 Hello.g4 -visitor

- set the visitor configuration to true when using the Maven plugin…
  <configuration>
  <visitor>true</visitor>
  </configuration>

Imagine the following grammar…

```
grammar EvalSimple;

expr: expr op=('*'|'/') expr       # MulDiv
        | expr op=('+'|'-') expr   # AddSub
        | '(' expr ')'             # parens
        | INT                      # int
        ;

MUL: '*';
DIV: '/';
ADD: '+';
SUB: '-';
INT: [0-9]+;
WS: [ \n\r\t]+ -> skip;
```

The following code evaluates an sample expression for this grammar via its Visitor. Note that, unlike the listener, a visitor…

- has visit...() methods for rules instead of enter...()/exit...() methods.
- explicitly drills down using visit(<name>), where name is one of the labels (e.g. expr)

```
public static void main(String[] args) throws Exception {
    CharStream charStream = CharStreams.fromString("(1+2)*3");

    EvalSimpleLexer lexer = new EvalSimpleLexer(charStream);
    TokenStream tokenStream = new CommonTokenStream(lexer);
    EvalSimpleParser parser = new EvalSimpleParser(tokenStream);

    ParseTree tree = parser.expr();

    EvalSimpleVisitor<Integer> eval = new EvalSimpleBaseVisitor<Integer>()
{
        @Override
        public Integer visitInt(EvalSimpleParser.IntContext ctx) {
```

```
            return Integer.valueOf(ctx.INT().getText());
        }

        @Override
        public Integer visitAddSub(EvalSimpleParser.AddSubContext ctx) {
            int left = visit(ctx.expr(0));
            int right = visit(ctx.expr(1));
            switch (ctx.op.getType()) {
                case EvalSimpleParser.ADD:
                    return left + right;
                case EvalSimpleParser.SUB:
                    return left - right;
                default:
                    throw new IllegalStateException();
            }
        }

        @Override
        public Integer visitMulDiv(EvalSimpleParser.MulDivContext ctx) {
            int left = visit(ctx.expr(0));
            int right = visit(ctx.expr(1));
            switch (ctx.op.getType()) {
                case EvalSimpleParser.MUL:
                    return left * right;
                case EvalSimpleParser.DIV:
                    return left / right;
                default:
                    throw new IllegalStateException();
            }
        }

        @Override
        public Integer visitParens(EvalSimpleParser.ParensContext ctx) {
            return visit(ctx.expr());
        }

    };

    int val = eval.visit(tree);
    System.out.println(val);
}
```

> **NOTE**: Classes prefixed by EvalSimple are classes that ANTLR generated for the grammar.

Output is...

```
9
```

Sharing Data

> **NOTE**: Documented in further detail in Chapter 7 of the ANTLR4 book.

Sharing data during tree traversal can be down multiple ways.

If you're using a visitor, each visit…() method can return a value back up to the invoker…

```
EvalSimpleVisitor<Integer> eval = new EvalSimpleBaseVisitor<Integer>() {
    @Override
    public Integer visitInt(EvalSimpleParser.IntContext ctx) {
        return Integer.valueOf(ctx.INT().getText());
    }

    @Override
    public Integer visitAddSub(EvalSimpleParser.AddSubContext ctx) {
        int left = visit(ctx.expr(0));
        int right = visit(ctx.expr(1));
        switch (ctx.op.getType()) {
            case EvalSimpleParser.ADD:
                return left + right;
            case EvalSimpleParser.SUB:
                return left - right;
            default:
                throw new IllegalStateException();
        }
    }
};

int val = eval.visit(tree);
System.out.println(val);
```

If you're using a listener instead of a visitor, you can mimic this behaviour using a Stack object…

```
EvalSimpleListener<Integer> listener = new EvalSimpleBaseListener {
    Stack<Integer> stack = new Stack<>();
```

```
    @Override
    public void enterInt(EvalSimpleParser.IntContext ctx) {
        int val = Integer.valueOf(ctx.INT().getText());
        stack.push(val);
    }

    @Override
    public void exitAddSub(EvalSimpleParser.AddSubContext ctx) {
        int right = stack.pop();
        int left = stack.pop();
        switch (ctx.op.getType()) {
            case EvalSimpleParser.ADD:
                stack.push(left + right);
            case EvalSimpleParser.SUB:
                stack.push(left - right);
            default:
                throw new IllegalStateException();
        }
    }
};

walker.walk(listener, parseTree);
System.out.println(stack.pop());
```

Another option is to use a IdentityMap instead of Stack. Using a map, you can persist values and the order of popping items doesn't really matter...

```
EvalSimpleListener<Integer> listener = new EvalSimpleBaseListener {
    Map<ParseTree, Integer> map = new IdentityHashMap<>();

    @Override
    public void enterInt(EvalSimpleParser.IntContext ctx) {
        int val = Integer.valueOf(ctx.INT().getText());
        map.put(ctx, val);
    }

    @Override
    public void exitAddSub(EvalSimpleParser.AddSubContext ctx) {
        int left = map.get(ctx.expr(0));
        int right = map.get(ctx.expr(1));
        switch (ctx.op.getType()) {
```

```
            case EvalSimpleParser.ADD:
                map.put(ctx, left + right);
            case EvalSimpleParser.SUB:
                map.put(ctx, left - right);
            default:
                throw new IllegalStateException();
        }
    }
};

walker.walk(listener, parseTree);
System.out.println(map.get(parseTree));
```

> **NOTE**: This MUST be an IdentityHashMap. You can't use a regular Map (e.g. HashMap) because those maps use the equals()/hashCode() determine equality of the keys -- the Context objects we're passing in don't implement those. We need reference equality, which is what IdentityHashMap provides

Testing Grammars

You can test grammars by using the TestRig tool that comes with ANTLR. It allows you to pass in a string and have it parsed by a specific rule in your grammar, outputting different views.

Imagine the following grammar…

```
grammar Hello;
r: 'hello'ID;
ID: [a-z]+;
WS: [\t\r\n]+ -> skip;
```

> **NOTE**: This example is from the book.

> **NOTE**: Remember that we created aliases in the setup section. The TestRig tool got bound to the grun alias and the ANTLR compiler got bound to the antlr4 alias.

Before testing it, you need to compile the grammar like your typically would any other grammar...

```
$ antlr4 Hello.g4
$ javac *.java
```

Then, run the TestRig tool. The string to be parsed will need to be passed into stdin and the tool expects the following arguments…

1. grammar name

2. rule to parse
3. output type

   NOTE: If you don't feed in a file to stdin, remember that in Linux you can type out the input and send an EOF character by hitting Ctrl+D.

For example...

```
$ grun Hello r -tokens
hello worlddddd
line 1:5 token recognition error at: ' '
[@0,0:4='hello',<'hello'>,1:0]
[@1,6:14='worlddddd',<ID>,1:6]
[@2,16:15='<EOF>',<EOF>,2:0]
```

Tokens

For example, if we wanted to get the tokens for the r rule of our Hello grammar…

```
$ grun Hello r -tokens
hello worlddddd
line 1:5 token recognition error at: ' '
[@0,0:4='hello',<'hello'>,1:0]
[@1,6:14='worlddddd',<ID>,1:6]
[@2,16:15='<EOF>',<EOF>,2:0]
```

Each line of output is a token (not including error/warning messages). The format is…

[@<token_idx>,<start_idx>:<end_idx>,<text>,<token_id>,<line_idx>:<line_char_idx>]

- token_idx → index of token in the tokenized input string
- start_idx → character index that this token starts at in the input string
- end_idx → character index that this token starts at in the input string
- text → text that makes up this token
- token_id → identifier assigned to this token by ANTLR
- line_idx → line number that this token starts at in the input string
- line_char_idx → character index that this token starts at in the line

Text Tree

For example, if we wanted to get the tree in LISP style for the r rule of our Hello grammar…

```
$ grun Hello r -tree
hello world
line 1:5 token recognition error at: ' '
```

```
(r hello world)
```

> **NOTE**: Don't understand how to read LISP trees? Check out https://www.tutorialspoint.com/lisp/lisp_tree.htm.

GUI Tree

For example, if we wanted to visualize the tree for the r rule of our Hello grammar…

```
$ grun Hello r -gui
hello world
line 1:5 token recognition error at: ' '
```

Grammar Gotchas

There are multiple different gotchas when it comes to ANTLR grammars.

Alternative Ambiguities

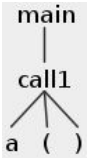
For alternatives, if the input ends up matching more than 1 alternative matches more than 1 alternative, the first alternative matched is the one that gets drilled down to. For example, imagine the following grammar…

```
grammar Test;

main: call1 | call2;

call1: ID '(' ')';

call2: ID '(' args ')';
```

```
args: ID | ID (',' ID)*;

ID: [a-z]+;
```

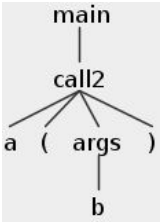
If you use this grammar to parse the string a(), it will match both the call1 and call2 rules. However, since call1 is first, it will drill down into call1…

```
$ antlr4 Test.g4
$ javac *.java
$ grun Test main -gui
a()
```



If you use this grammar to parse the string a(b), it will only end up matching the call2 rule…

```
$ antlr4 Test.g4
$ javac *.java
$ grun Test main -gui
a(b)
```



Lexing Ambiguities

For lexing, if the input ends up matching multiple lexing rules, ANTLR resolves the ambiguity by using the first matching rule as specified in the grammar file. For example, imagine the following grammar…

```
grammar Test;

main: LEXRULE1 | LEXRULE2;

LEXRULE1: [a-z]+;
LEXRULE2: 'aaa'[a-z]+;
```

If you use this grammar to parse a string starting with aaa, it may look like it matches LEXRULE2 but it will actually match LEXRULE1. LEXRULE1 is the first rule in the grammar that matches the input...

```
$ antlr4 Test.g4
$ javac *.java
$ grun Test main -tokens
aaahi
line 1:5 token recognition error at: '\n'
[@0,0:4='aaahi',<LEXRULE1>,1:0]
[@1,6:5='<EOF>',<EOF>,2:0]
```

Indirect Left Recursion

Imagine the following grammar...

```
grammar Hello;

stat: block | '}';
block: stat ID;

ID: [A-Za-z0-9]+;
WS: [ \t\r\n]+ -> skip;
```

At first glance the above grammar looks valid, but it isn't...

```
$ antlr4 Hello.g4
error(119): Hello.g4::: The following sets of rules are mutually
left-recursive [stat, block]
```

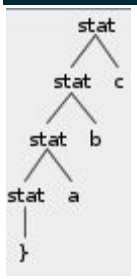
The problem here is that the left recursion is indirect -- it's drilling down to block which is drilling back into stat (all on the left). The term for this is indirect left recursion, and ANTLR doesn't support it. To get it to work, you need to embed the rules such that the recursion is directly happening…

```
grammar Hello;

stat: stat ID | '}';

ID: [A-Za-z0-9]+;
WS: [ \t\r\n]+ -> skip;
```

Note how the block rule was embedded directly in this stat rule. This will now compile and run…

```
$ antlr4 Hello.g4
```

```
$ javac *.java
$ grun Hello expr -gui
} a b c
```



What's the reason for this requirement? This is what I think is happening...

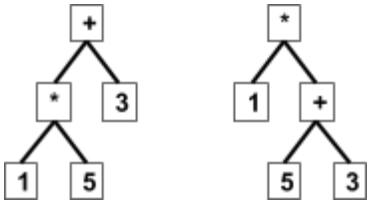
1. Remember that ANTLR is a top-down recursive descent parser -- each rule gets converted to a method, and drilling-down into a rule is essentially one method invoking another.
2. Remember that alternatives have a built-in mechanism for disambiguating the different paths for a rule. See the Alternatives Ambiguities section for more information.

I think what's happening is that once it drills down into another rule, the disambugation mechanism of alternatives no longer applies (it enters another method). So in the above uncompilable example, stat invokes block which invokes stat which invokes block which invokes stat etc..

The ANTLR book's example for this seems to be a bit more intuitive to understand. There's no way for the following grammar to properly be parsed because a consistent tree can't be generated…

```
grammar Hello;

expr: mul
      | add
      | NUM
      ;
add: expr '+' expr;
mul: expr '*' expr;

NUM: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

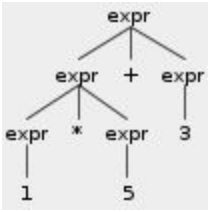
For example, the input 1*5+3 could result in either of the following trees…



We want the 1st form because want to respect operator precedence -- we want to evaluate multiplication first and then addition. However, it's ambiguous as to which rule should be chosen first because the tie-breaking mechanism of alternatives no longer apply when recursion happens indirectly. To bring them back into play for the above example, simply embed the recursing rules…

```
grammar Hello;

expr: expr '*' expr    // this path will always take precedence
      | expr '+' expr  // then this path
      | NUM            // then finally this path
      ;

NUM: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello expr -gui
1*5+3
```



Left Recursion Patterns

> **NOTE**: This is from Chapter 14 of the ANTLR book. For more information, read that chapter.

ANTLR only supports 4 different patterns when it comes to left recursion. These 4 specific patterns are supported by internally re-writing the rules such that they are no longer left-recursive.

These 4 patterns are…

- **binary**

  expr: expr '*' expr
  | NUMBER;

  The separator in between the recursive rule invocations can be a single token, a choice of tokens, or another rule.

- **ternary**

  expr: expr '?' expr ':' expr
  | NUMBER;

  The separator in between the recursive rule invocations must be single token.

- **unary prefix**

  expr: 'A' expr
  | ID;

  This is referred to as tail-recursion (see Terminology section for definition). It support any number of elements before the final recursive rule invocation, so long as those elements aren't recursive themselves.

- **unary suffix**

  expr: expr 'A'
  | ID;

  This is similar to the unary suffix operator. It support any number of elements after the left recursive rule invocation, so long as those elements aren't recursive themselves.

Note that these rules aren't generic -- each pattern in the list above has important caveats.

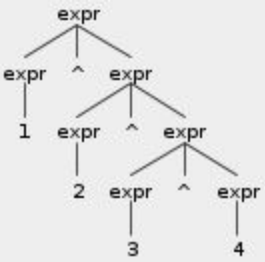
Right-to-Left Recursion

In certain cases, you may want to recursively drill down right-to-left instead of the default of left-to-right. You would want to do this to generate the AST properly, so that you can evaluate it properly.

To perform right-to-left parsing you need to tack on the <assoc=right> directive to the front of a alternative. For example, operators such as exponent need to be drilled down into right-to-left to properly create the AST…

```
grammar Hello;

expr:
        <assoc=right> expr '^' expr
        | expr ('*' | '/') expr
        | expr ('+' | '-') expr
        | INT
        ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello expr -gui
1^2^3^4
```



The above AST is the correct for evaluating the input. What should be happening when you chain exponents like this is 1^(2^(3^4)). Written in standard math notation that would be $1^{2^{3^4}}$. This is exactly what the tree does. The bottom most branch is what would get evaluated first: 3^4. The branch above it would get evaluated next: 2^(3^4). Finally, the root would get evaluated: 1^(2^(3^4)).

> **NOTE**: Previous versions of ANTLR v4 had you place the <assoc=right> directive on an actual item in the sequence. This is no longer supported. You need to place it at the beginning of the alternative.

Retaining Whitespace

In certain cases, you may need access to the original whitespace of whatever it is you're parsing. If you use...

- -> skip to skip whitespace, it won't be included in the token stream.
- -> channel(name) to skip whitespace, it will be included in the token stream.

A parser can only target a single channel, so if you use ->channel(name) the tokens will be available in the token stream but the parser won't make use of them so long as it isn't targeting that specific channel name. The most common channel to use for this specific usecase is the HIDDEN channel (this is one of the default channels).

```
WS: [ \t\r\n\u000C]+ -> channel(HIDDEN);  // included in tokenstream
WS: [ \t\r\n\u000C]+ -> skip;             // not included in tokenstream
```

When you dump tokens into the hidden channel, the whitespace is available in the tokenstream but it won't be used unless you explicitly tell the parser/lexer to target that channel. Here's an example of how to dump the original text for a rule…

```
public class ExtractInterfaceListener extends Java9BaseListener {
    TokenStream tokens;

    public ExtractInterfaceListener(TokenStream tokens) {
        this.tokens = tokens;
    }

    @Override
    public void enterImportDeclaration(Java9Parser.ImportDeclarationContext
ctx) {
        Token start = ctx.getStart();
        Token stop = ctx.getStop();

        tokens.getText(start, stop);

        System.out.println(txt);
    }
}
```

Grammar Patterns

The following subsections provide examples for common language syntax patterns in ANTLR's grammar format.

Common Tokens

For new grammars, the smart thing to do would be to get a grammar for a somewhat similar existing language copy over whatever base lexing rules you need, tweaking as necessary. For example, the following are lexer rules common to most languages (slight variations may apply)..

```
INT: [0-9]+;
FLOAT: [0-9]+ '.' [0-9]*;

STRING: '"' (ESC|.)*? '"';
fragment ESC : '\\"' | '\\\\';

ID: [A-Za-z]+;
WS: [ \t\r\n]+ -> skip; // can also dump to hidden channel
```

> **NOTE**: In the STRING lexing rule there's a * quantifier followed by ?. The ? makes it non-greedy (similar to Regex?). Also, even though we're skipping whitespace, dot (.) will include whitespace. See the Lexer Ambiguities section for more information.

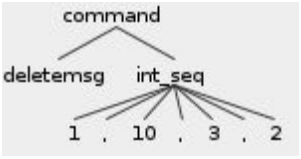
Delimited Lists

You can define arbitrary-sized lists that are separated by some delimiter. For example…

```
grammar Hello;

command: ID int_seq;
int_seq: (INT ',')* INT;// INT (',' INT)*    <-same
                        // (INT (',' INT)*)? <-same but can also be nothing

INT: [0-9]+;
ID: [A-Za-z]+;
WS: [ \t\r\n]+ -> skip;
```

> **NOTE**: Remember that the term for this is 'sequence'. This is using a zero-or-more quantifier (*). For a full list of quantifiers, see the Quantifiers parsing section.

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello command -gui
deletemsg 1,10,3,2
```



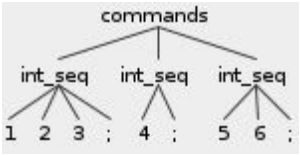
Terminated Lists

You can define lists that are ended by a terminator. For example…

```
grammar Hello;

commands: int_seq+;
int_seq: INT+ ';';


INT: [0-9]+;
ID: [A-Za-z0-9]+;
WS: [ \t\r\n]+ -> skip;
```

NOTE: Remember that the term for this is ‘sequence’. This is using a one-or-more quantifier (+). For a full list of quantifiers, see the Quantifiers parsing section.

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello command -gui
1 2 3;
4;
5 6;
```



Matching Braces

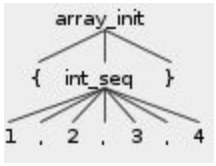
You can define matching tokens directly within rules. For example...

```
grammar Hello;

int_seq: (INT ',')* INT;
array_init: '{' int_seq '}';

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

```
WS: [ \t\r\n]+ -> skip;
```

NOTE: Remember that the term for this is ‘token dependence’.

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello array_int -gui
{1,2,3,4}
```



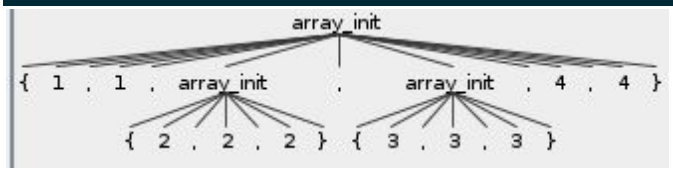
Nesting Statements

You can support nesting by making a rule invoke itself. For example, the following rule is similar to the rule in the Matching Brace section above, except that the rule can also invoke itself...

```
grammar Hello;

array_init:
        '{'
        ((INT | array_init) ',')*
        (INT | array_init)
        '}'
        ;

INT: [0-9]+;
WS: [ \t\r\n]+ -> skip;
```

NOTE: Remember that the term for this is ‘nested phrase’.

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello array_int -gui
{1,1,{2,2,2},{3,3,3},4,4}
```



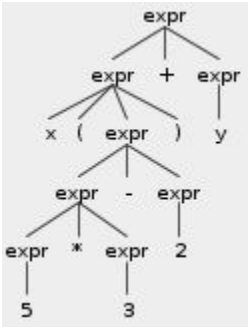
Expressions, Functions, and Operator Precedence

You can process basic math expressions (with functions). For example, the following rule grammar will parse an equation while fully respecting BEDMAS/PEDMAS ordering requirements (note the order of the alternatives and resulting hierarchy for the input)...

```
grammar Hello;

expr:
        '(' expr ')'                    # bracket
        | <assoc=right> expr '^' expr   # exponent
        | expr ( '/' | '*' ) expr       # divmul
        | expr ( '+' | '-' ) expr       # addsub
        | ID '(' (expr ',')* expr? ')' # function
        | INT                           # constant
        | ID                            # variable
        ;

OPEN_BRACKET: '(';
CLOSE_BRACKET: ')';
EXPONENT: '^';
DIVIDE: '/';
MULTIPLY: '*';
ADD: '+';
SUBTRACT: '-';

INT: [0-9]+;
ID: [A-Za-z0-9]+;
WS: [ \t\r\n]+ -> skip;
```

> **NOTE**: The order that the alternatives are in defines the operator precedence here. See the Alternative Ambiguities section for more information.
>
> **NOTE**: You may be tempted to try breaking up the alternatives in the rule above into multiple rules, but you can't do this. See the Indirect Left Recursion section.

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello expr -gui
x(5*3-2)+y
```



> **NOTE**: Notice that there's a right-associative directive on the alternative that handles exponents. The reason for this is that it properly handles precedence when multiple exponents are chained together. See the Right-to-Left Recursion gotcha section for more information.

Comments

When you're parsing a language, you likely want to handle comments. There are 3 different ways to handle this…

1. read comments as a single token and use -> skip on that token
2. read comments as a single token and use -> channel(HIDDEN) on that token
3. use a different lexer mode for the comments

Options 1 and 2 are the easiest…

```
LINE_COMMENT: '//' .*? '\r'? '\n' -> skip;
LINE_COMMENT: '//' .*? '\r'? '\n' -> channel(HIDDEN);

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

> **NOTE**: Remember that dumping into a channel makes the comment still available in the token stream but ignored by the parser (assuming the parser isn't targeting that channel), while skipping will remove it from the token stream.

Option 3 is also available if you foresee doing any processing on the comments...

```
COMMENT_OPEN: '/*' -> pushMode(COMMENT);

MODE COMMENT;

COMMENT_DATA: ~'*' ~'/';
COMMENT_CLOSE: '*/' -> popMode();
```

Programming Patterns

ANTLR's runtime isn't documented very well and some of it isn't intuitive to use. The following subsections contain common ANTLR programming patterns.

Outputting Matched Rule

You can output the original input (including any whitespace / discarded characters) for any rule that matched by grabbing the starting char index of the starting token and the ending char index of the ending token. For example...

```
@Override
public void enterImportDeclaration(ImportDeclarationContext ctx) {
    Token startToken = ctx.getStart();
    Token stopToken = ctx.getStop();

    int startCharIdx = startToken.getStartIndex();
    int stopCharIdx = stopToken.getStopIndex();

    Interval charInterval = Interval.of(startCharIdx, stopCharIdx);
    CharStream charStream = startToken.getInputStream();

    String txt = charStream.getText(charInterval);
    System.out.println(txt);
}
```

> **NOTE**: The above works in every case, even if you use the -> skip lexer directive to skip whitespace. A better way to get proper whitespace is to feed the whitespace to the hidden channel via the -> channel(HIDDEN) directive. See the Retaining Whitespace gotcha section.

Outputting LISP-style AST

You can output the AST in LISP-style by using parseTree.toStringTree()...

```
You can output a LISP-style tree of the AST. For example...

CharStream charStream = CharStreams.fromString(
        "import java.util.List;\n"
        + "import java.util.Map;\n"
        + "public class Demo {\n"
        + " void f(int x, String y) { }\n"
```

```
        + " int[ ] g(/*no args*/) { return null; }\n"
        + " List<Map<String, Integer>>[] h() { return null; }\n"
        + "}");

Java9Lexer lexer = new Java9Lexer(charStream);
TokenStream tokenStream = new CommonTokenStream(lexer);
Java9Parser parser = new Java9Parser(tokenStream);

ParseTree tree = parser.compilationUnit(); // parse

System.out.println(tree.toStringTree(parser));
```

```
(compilationUnit (ordinaryCompilation (importDeclaration
(singleTypeImportDeclaration import (typeName (packageOrTypeName
(packageOrTypeName java) . util) . List) ;)) (importDeclaration
(singleTypeImportDeclaration import ( ...
```

Adding/Deleting/Replacing from AST

You can add/remove/delete content from a specific region of an AST using a TokenStreamRewriter. For example…

```
public class InsertSerialIDListener extends Java9BaseListener {
    public TokenStreamRewriter rewriter;

    public InsertSerialIDListener(TokenStream tokens) {
        rewriter = new TokenStreamRewriter(tokens);
    }

    @Override
    public void enterClassBody(Java9Parser.ClassBodyContext ctx) {
        rewriter.insertAfter(
            ctx.start,
            "\n\tpublic static final long serialVersionUID = 1L;");
    }
}
```

Once the AST has been walked, you can output the re-written stream using TokenStreamRewriter.getText(). For example...

```
ParseTreeWalker walker = new ParseTreeWalker();
InsertSerialIDListener extractor = new InsertSerialIDListener(tokenStream);
```

```
walker.walk(extractor, tree);

String output = extractor.rewriter.getText();
```

> **NOTE**: If you want getText() to return everything in the original input (e.g. including skipped whitespace), you need to use channels. See the See the Retaining Whitespace gotcha section.

Capturing and Reporting Errors

To handle errors, implement a custom ANTLRErrorStrategy and pass it to parser.setErrorHandler(). See the last section of chapter 9 in the ANTLR4 book to figure out what exactly it is to do, but an implementation is available to let you bail out on an error called BailErrorStrategy...

```
CharStream charStream = CharStreams.fromFile("file.java");

Java9Lexer lexer = new Java9Lexer(charStream);
TokenStream tokenStream = new CommonTokenStream(lexer);
Java9Parser parser = new Java9Parser(tokenStream);

parser.addErrorListener(new BailErrorStrategy());
```

To report errors, implement a custom ANTLRErrorListener and pass it to parser.addErrorListener(). The following example extends BaseErrorListener, which is the default base class for the ANTLRErrorListener interface...

```
CharStream charStream = CharStreams.fromFile("file.java");

Java9Lexer lexer = new Java9Lexer(charStream);
TokenStream tokenStream = new CommonTokenStream(lexer);
Java9Parser parser = new Java9Parser(tokenStream);

parser.removeErrorListeners();                  // remove default listener
parser.addErrorListener(new BaseErrorListener() {// add new listener
    @Override
    public void syntaxError(
            Recognizer<?, ?> recognizer,
            Object offendingSymbol,
            int line,
            int charPositionInLine,
            String msg,
            RecognitionException e) {
```

```
        // ...
    }
});
```

> **NOTE**: If you want an example, look at ConsoleErrorListener in the ANTLR runtime.

Creating Custom Error Messages

You can intentionally parse error cases and generate error messages for them by calling notifyErrorListeners() directly from the grammar. This is done on the choices of an alternative…

```
grammar Hello;

expr: expr '+' expr
    | expr '-' expr
    | expr '*' expr { notifyErrorListeners("mult not supported"); }
    | expr '/' expr
    | ID;

ID: [a-zA-Z]+;
WS: [ \t\r\n]+ -> skip
```

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello tag -gui
a+b*c
line 2:0 mult not supported
```

Best Practices

The following subsections detail common patterns (conceptual) for ANTLR.

Use English to create grammars for existing languages

> **NOTE**: This is described in chapter 5 of the ANTLR book. For a more thorough explanation, read that.

Grammars for existing languages should be designed in a top-down manner. There's a top-level/main rule that's coarse and acts as the entry-point for the grammar, and as that rule gets fleshed out it should introduce new finer-level rules.

When you initially write the start rule, it should be basically an english phrase that describes what's happening. For example, if I wanted to create a grammar to parse simplified XML, my top-level rule could be…

tag: an opening tag, followed by nested tags and/or text, followed by a closing tag

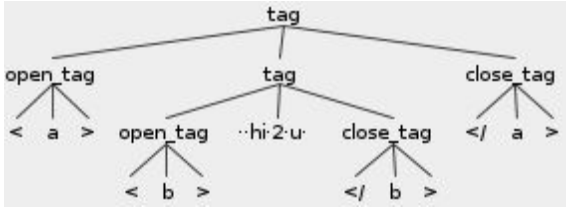
We can then break this up into lower-level rules. For example..

tag: open_tag, followed by nested tags and/or text, close_tag
open_tag: a <, followed by some alphanumeric string, followed by a >
close_tag: a </, followed by some alphanumeric string, followed by a >

Once we have something like this, we can start implementing actual grammar rules…

```
grammar Hello;

tag:  open_tag (tag | TEXT)? close_tag;
open_tag: '<' ID '>';
close_tag: '</' ID '>';

ID: [A-Za-z0-9]+;
TEXT: ~[<>]+; // any chars that aren't < and >
WS: [ \t\r\n]+ -> skip;
```

```
$ antlr4 Hello.g4
$ javac *.java
$ grun Hello tag -gui
<a><b>  hi 2 u </b></a>
```



When you write out the rules in English, here our how things generally break down...

- nouns break down to either rules or tokens.
- "x and y" break down to sequences.
- "x or y" break down to alternatives.
- "maybe x" / "at most one x" break down to a ? quantifier.
- "one or more x" / "at least one x" / "many" break down to a + quantifier.
- "zero or more x" / "any number of x" / "many" break down to a * quantifier.

For most cases, it doesn't make sense to begin writing lexing rules scratch. Most languages have common lexing (and parsing) patterns between them. For example, …

- an integer/string/float literal is specified the same way in almost every language.
- an identifier is specified the same way in almost every language.
- whitespace (not in a string literal) is almost always ignored.

See the Common Tokens section for more information.

Lex but skip whitespace and comments

> **NOTE**: This is described in the last section of chapter 6 of the ANTLR book. For a more thorough explanation, read that.

If you're just dealing with the core language, use lexing rules to read in but skip over whitespace and comments. If you include them, you have to write your parser to constantly check for whitespace and/or comments.

For example, compare…

```
row: WS? (ID WS? ',' WS?)* ID WS?;
file: row*;

ID: [a-zA-Z]+;
WS: [ \t\r\n]+;
```

to...

```
row: (ID ',')* ID;
file: row*;

ID: [a-zA-Z]+;
WS: [ \t\r\n]+ -> skip
```

The row rule in the bottom one is much simpler.

> **NOTE**: Remember that you can push into the HIDDEN channel instead of skipping. This makes whatever you skip available in the token stream, but you can still hide it from the parser.

Lex common tokens

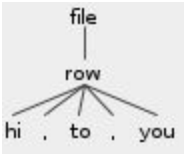
> **NOTE**: This is described in the last section of chapter 6 of the ANTLR book. For a more thorough explanation, read that.

There's a class of parsers called scannerless parsers that don't do any lexing before parsing. That is, only characters are used as tokens by the parser.

Scannerless parsers should be avoided (in ANTLR or in general?) because lexing has less overhead than the parsing. Common tokens such as keywords, literals (e.g. ints, floats, strings, etc..), identifiers, comments, etc.. should be identified by lexing rules.

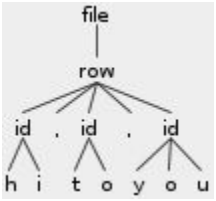
For example, compare passing the string "hi, to, you" into the following grammar…

```
row: (ID ',')* ID;
file: row*;

ID: [a-zA-Z]+;
WS: [ \t\r\n]+ -> skip;
```



vs...

```
id: LETTER+;
row: (id ',')* id;
file: row*;

LETTER: [a-zA-Z];
WS: [ \t\r\n]+ -> skip;
```



The top one is less computationally expensive because it's using the lexer to grab higher-level tokens for the parser.

Lex based on parser requirements

> **NOTE**: This is described in the last section of chapter 6 of the ANTLR book. For a more thorough explanation, read that.

This is common sense: make sure your lexer only tokenizes the minimum of what your parser needs.

For example, imagine you're parsing a language that supports string literals. At the syntax level, you don't really care about the individual components of the string literal or the contents of the string literal. All you really care about is knowing that it's there and that it's syntactically valid.

If our lexer took a string literal and tokenized it into its individual components…

- beginning quote (string start)
- word
- whitespace
- word
- whitespace
- word
- whitespace
- ...
- ending quote (string end)

it would be a ton of useless for the parser. As such, the best thing to do would be to lex the string literal as a whole such that it gets returned as a single token.