Reading 18: Parser Generators

Software in 6.005

Safe from bugs	**Easy to understand**	**Ready for change**
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

Objectives

After today's class, you should:

- Be able to use a grammar in combination with a parser generator, to parse a character sequence into a parse tree
- Be able to convert a parse tree into a useful data type

Parser Generators

A *parser generator* is a good tool that you should make part of your toolbox. A parser generator takes a grammar as input and automatically generates source code that can parse streams of characters using the grammar.

The generated code is a *parser* , which takes a sequence of characters and tries to match the sequence against the grammar. The parser typically produces a *parse tree* , which shows how grammar productions are expanded into a sentence that matches the character sequence. The root of the parse tree is the starting nonterminal of the grammar. Each node of the parse tree expands into one production of the grammar. We'll see how a parse tree actually looks in the next section.

The final step of parsing is to do something useful with this parse tree. We're going to translate it into a value of a recursive data type. Recursive abstract data types are often used to represent an expression in a language, like HTML, or Markdown, or Java, or algebraic expressions. A recursive abstract data type that represents a language expression is called an *abstract syntax tree* (AST).

Antlr is a mature and widely-used parser generator for Java, and other languages as well. The remainder of this reading will get you started with Antlr. If you run into trouble and need a deeper reference, you can look at:

- Definitive Antlr 4 Reference . A book about Antlr, both tutorial and reference.
- Antlr 4 Documentation Wiki . Concise documentation of the grammar file syntax.
- Antlr 4 Runtime API . Reference documentation for Antlr's Java classes and interfaces.

An Antlr Grammar

The code for the examples that follow can be found on GitHub as sp16-ex18-parser-generators .

Here is what our HTML grammar looks like as an Antlr source file:

```
grammar Html;

root : html EOF;
html : ( italic | normal ) *;
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;  /* represents a string of one or more characters that are not < or >
```

Let's break it down.

Each Antlr rule consists of a name, followed by a colon, followed by its definition, terminated by a semicolon.

Nonterminals in Antlr have to be lowercase: `root` , `html` , `normal` , `italic` . Terminals are either quoted strings, like `'<i>'` , or capitalized names, like `EOF` and `TEXT` .

```
root : html EOF;
```

`root` is the entry point of the grammar. This is the nonterminal that the whole input needs to match. We don't have to call it `root` . The entry point can be any nonterminal.

`EOF` is a special terminal, defined by Antlr, that means the end of the input. It stands for *end of file* , though your input may also come from a string or a network connection rather than just a file.

```
html : ( normal | italic ) *;
```

This rule shows that Antlr rules can have the alternation operator `|` , the repetition operators `*` and `+` , and parentheses for grouping, in the same way we've been using in the grammars reading . Optional parts can be marked with `?` , just like we did earlier, but this particular grammar doesn't use `?` .

```
italic : '<i>' html '</i>';
normal : TEXT;
TEXT : ~[<>]+;
```

`TEXT` is a terminal matching sequences of characters that are neither `<` nor `>` . In the more conventional regular expression syntax used earlier in this reading, we would write `[^<>]` to represent all characters except `<` and `>` . Antlr uses a slightly different syntax – `~` means *not* , and it is put in front of the square brackets instead of inside them, so `~[<>]` matches any character except `<` and `>` .

In Antlr, terminals can be defined using regular expressions, not just fixed strings. For example, here are some other terminal patterns we used in the URL grammar earlier in the reading, now written in Antlr syntax and with Antlr's required naming convention:

```
IDENTIFIER : [a-z]+;
INTEGER : [0-9]+;
```

More about Antlr's grammar file syntax can be found in Chapter 5 of the Definitive ANTLR 4 Reference .

Generating the parser

The rest of this reading will focus on the *IntegerExpression* grammar used in the exercise above, which we'll store in a file called `IntegerExpression.g4` . Antlr 4 grammar files end with `.g4` by convention.

The Antlr parser generator tool converts a grammar source file like `IntegerExpression.g4` into Java classes that implement a parser. To do that, you need to go to a command prompt (Terminal or Command Prompt) and run a command like this:

```
cd <root of project>
cd src/intexpr/parser
java -jar ../../../lib/antlr.jar IntegerExpression.g4
```

You need to make sure you `cd` into right folder (where `IntegerExpression.g4` is) and you refer to `antlr.jar` where it is in your project folder structure, using a relative path like `../../../lib/antlr.jar`).

Assuming you don't have any syntax errors in your grammar file, the parser generator will produce new Java source files in the current folder. Nothing will be printed in the terminal. The generated code is divided into several cooperating modules:

- the **lexer** takes a stream of characters as input, and produces a stream of terminals (Antlr calls them *tokens*) as output, like `NUMBER` , `+` , and `(` . For `IntegerExpression.g4` , the generated lexer is called `IntegerExpressionLexer.java` .
- the **parser** takes the stream of terminals produced by the lexer and produces a parse tree. The generated parser is called `IntegerExpressionParser.java` .
- the **tree walker** lets you write code that walks over the parse tree produced by the parser, as explained below. The generated tree walker files are the interface `IntegerExpressionListener.java` , and an empty implementation of the interface, `IntegerExpressionBaseListener.java` .

Antlr also generates two text files, `IntegerExpression.tokens` and `IntegerExpressionLexer.tokens` , that list the terminals that Antlr found in your grammar. These aren't needed for a simple parser, but they're needed when you include grammars inside other grammars.

Make sure that you:

- **Never edit the files generated by Antlr.** The right way to change your parser is to edit the grammar source file, `IntegerExpression.g4` , and then regenerate the Java classes.
- **Regenerate the files whenever you edit the grammar file.** This is easy to forget when Eclipse is compiling all your Java source files automatically. Eclipse does not regenerate your parser automatically. Make sure you rerun the `java -jar ...` command whenever you change your .g4 file.
- **Refresh your project in Eclipse each time you regenerate the files.** You can do this by clicking on your project in Eclipse and pressing F5, or right-clicking and choosing Refresh. The reason is that Eclipse sometimes uses older versions of files already part of the source, even if they have been modified on your filesystem.

More about using the Antlr parser generator to produce a parser can be found in Chapter 3 of the Definitive ANTLR 4 Reference .

Calling the parser

Now that you've generated the Java classes for your parser, you'll want to use them from your own code.

First we need to make a stream of characters to feed to the lexer. Antlr has a class `ANTLRInputStream` that makes this easy. It can take a `String` , or a `Reader` , or an `InputStream` as input. Here we are using a string:

```
CharStream stream = new ANTLRInputStream("54+(2+89)");
```

Next, we create an instance of the lexer class that our grammar file generated, and pass it the character stream:

```
IntegerExpressionLexer lexer = new IntegerExpressionLexer(stream);
TokenStream tokens = new CommonTokenStream(lexer);
```

The result is a stream of terminals, which we can then feed to the parser:

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

To actually do the parsing, we call a particular nonterminal on the parser. The generated parser has one method for every nonterminal in our grammar, including `root()` , `sum()` , and `primitive()` . We want to call the nonterminal that represents the set of strings that we want to match – in this case, `root()` .

Calling it produces a parse tree:

```
ParseTree tree = parser.root();
```

For debugging, we can then print this tree out:

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

Or we can display it in a handy graphical form:

```
Trees.inspect(tree, parser);
```

which pops up a window with the parse tree shown on the right.

In the example code: `Main.java` lines 34-50 , which use lines 71-85 .

Traversing the parse tree

So we've used the parser to turn a stream of characters into a parse tree, which shows how the grammar matches the stream. Now we need to do something with this parse tree. We're going to translate it into a value of a recursive abstract data type.

The first step is to learn how to traverse the parse tree. To do this, we use a `ParseTreeWalker` , which is an Antlr class that walks over a parse tree, visiting every node in order, top-to-bottom, left-to-right. As it visits each node in the tree, the walker calls methods on a *listener* object that we provide, which implements `IntegerExpressionListener` interface.

Just to warm up, here's a simple implementation of `IntegerExpressionListener` that just prints a message every time the walker calls us, so we can see how it gets used:

```
class PrintEverything implements IntegerExpressionListener {

    @Override public void enterRoot(IntegerExpressionParser.RootContext context) {
        System.err.println("entering root");
    }
    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        System.err.println("exiting root");
    }

    @Override public void enterSum(IntegerExpressionParser.SumContext context) {
        System.err.println("entering sum");
    }
    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        System.err.println("exiting sum");
    }

    @Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext context) {
        System.err.println("entering primitive");
    }
    @Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext context) {
        System.err.println("exiting primitive");
    }

    @Override public void visitTerminal(TerminalNode terminal) {
        System.err.println("terminal " + terminal.getText());
    }

    // don't need these here, so just make them empty implementations
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}
```

Notice that every nonterminal `N` in the grammar has corresponding `enterN()` and `exitN()` methods in the listener interface, which are called when the tree walk enters and exits a parse tree node for nonterminal `N` , respectively. There is also a `visitTerminal()` that is called when the walk reaches a leaf of the parse tree. Each of these methods has a parameter that provides information about the nonterminal or terminal node that the walk is currently visiting.

The listener interface also has some methods that we don't need. The methods `enterEveryRule()` and `exitEveryRule()` are called on entering and exiting *any* nonterminal node, in case we want some generic behavior. The method `visitErrorNode()` is called if the input contained a syntax error that produced an error node in the parse tree. In the parser we're writing, however, a syntax error causes an exception to be thrown, so we won't see any parse trees with error nodes in them. The interface requires us to implement these methods, but we can just leave their method bodies empty.

```
ParseTreeWalker walker = new ParseTreeWalker();
IntegerExpressionListener listener = new PrintEverything();
walker.walk(listener, tree);
```

If we walk over the parse tree with this listener, then we see the following output:

```
entering root
entering sum
entering primitive
terminal 54
exiting primitive
terminal +
entering primitive
terminal (
entering sum
entering primitive
terminal 2
exiting primitive
terminal +
entering primitive
terminal 89
exiting primitive
exiting sum
terminal )
exiting primitive
exiting sum
terminal <EOF>
exiting root
```

Compare this printout with the parse tree shown at the right, and you'll see that the `ParseTreeWalker` is stepping through the nodes of the tree in order, from parents to children, and from left to right through the siblings.

Constructing an abstract syntax tree

We need to convert the parse tree into a recursive data type. Here's the definition of the recursive data type that we're going to use to represent integer arithmetic expressions:

```
IntegerExpression = Number(n:int)
                    + Plus(left:IntegerExpression, right:IntegerExpression)
```

If this syntax is mysterious, review recursive data type definitions .

When a recursive data type represents a language this way, it is often called an *abstract syntax tree* . A `IntegerExpression` value captures the important features of the expression – its grouping and the integers in it – while omitting unnecessary details of the sequence of characters that created it.

By contrast, the parse tree that we just generated with the *IntegerExpression* parser is a *concrete syntax tree* . It's called concrete, rather than abstract, because it contains more details about how the expression is represented in actual characters. For example, the strings `2+2` , `((2)+(2))` , and `0002+0002` would each produce a different concrete syntax tree, but these trees would all correspond to the same abstract `IntegerExpression` value: `Plus(Number(2), Number(2))` .

Now we create a listener that constructs a `IntegerExpression` tree while it's walking over the parse tree. Each parse tree node will correspond to a `IntegerExpression` variant: `sum` nodes will create `Plus` objects, and `primitive` nodes (that matched the `NUMBER` terminal) will create `Number` objects.

Some `primitive` nodes are parenthesized subexpressions, not numbers. For these nodes, our listener will construct no new `IntegerExpression` object at all. Parentheses are concrete syntax whose meaning is captured in the abstract syntax tree by the structure of `Plus` and `Number` objects, and we already have a `Plus` to represent the `sum` inside the parentheses.

Whenever the walker exits each node of the parse tree, we have walked over the entire subtree under that node, so we create the next `IntegerExpression` object at exit time. But we have to keep track of all the children that were created during the walk over that subtree. We use a stack to store them.

Here's the code:

```
/** Make a IntegerExpresion value from a parse tree. */
class MakeIntegerExpression implements IntegerExpressionListener {
    private Stack<IntegerExpression> stack = new Stack<>();
    // Invariant: stack contains the IntegerExpression value of each parse
    // subtree that has been fully-walked so far, but whose parent has not yet
    // been exited by the walk. The stack is ordered by recency of visit, so that
    // the top of the stack is the IntegerExpression for the most recently walked
    // subtree.
    //
    // At the start of the walk, the stack is empty, because no subtrees have
    // been fully walked.
    //
    // Whenever a node is exited by the walk, the IntegerExpression values of its
    // children are on top of the stack, in order with the last child on top. To
    // preserve the invariant, we must pop those child IntegerExpression values
    // from the stack, combine them with the appropriate IntegerExpression
    // producer, and push back an IntegerExpression value representing the entire
    // subtree under the node.
    //
    // At the end of the walk, after all subtrees have been walked and the the
    // root has been exited, only the entire tree satisfies the invariant's
    // "fully walked but parent not yet exited" property, so the top of the stack
    // is the IntegerExpression of the entire parse tree.

    /**
     * Returns the expression constructed by this listener object.
     * Requires that this listener has completely walked over an IntegerExpression
     * parse tree using ParseTreeWalker.
     * @return IntegerExpression for the parse tree that was walked
     */
    public IntegerExpression getExpression() {
        return stack.get(0);
    }

    @Override public void exitRoot(IntegerExpressionParser.RootContext context) {
        // do nothing, root has only one child so its value is
        // already on top of the stack
    }

    @Override public void exitSum(IntegerExpressionParser.SumContext context) {
        // matched the primitive ('+' primitive)* rule
        List<IntegerExpressionParser.PrimitiveContext> addends = context.primitive();
        assert stack.size() >= addends.size();

        // the pattern above always has at least 1 child;
        // pop the last child
        assert addends.size() > 0;
        IntegerExpression sum = stack.pop();

        // pop the older children, one by one, and add them on
        for (int i = 1; i < addends.size(); ++i) {
            sum = new Plus(stack.pop(), sum);
        }

        // the result is this subtree's IntegerExpression
        stack.push(sum);
    }

    @Override public void exitPrimitive(IntegerExpressionParser.PrimitiveContext context) {
        if (context.NUMBER() != null) {
            // matched the NUMBER alternative
            int n = Integer.valueOf(context.NUMBER().getText());
            IntegerExpression number = new Number(n);
            stack.push(number);
        } else {
            // matched the '(' sum ')' alternative
            // do nothing, because sum's value is already on the stack
        }
    }

    // don't need these here, so just make them empty implementations
    @Override public void enterRoot(IntegerExpressionParser.RootContext context) { }
    @Override public void enterSum(IntegerExpressionParser.SumContext context) { }
    @Override public void enterPrimitive(IntegerExpressionParser.PrimitiveContext context) { }

    @Override public void visitTerminal(TerminalNode terminal) { }
    @Override public void enterEveryRule(ParserRuleContext context) { }
    @Override public void exitEveryRule(ParserRuleContext context) { }
    @Override public void visitErrorNode(ErrorNode node) { }
}
```

More about Antlr's parse-tree listeners can be found in Section 7.2 of the Definitive ANTLR 4 Reference .

Handling errors

By default, Antlr parsers print errors to the console. In order to make the parser modular, however, we need to handle those errors differently. You can attach an `ErrorListener` to the lexer and parser in order to throw an exception when an error is encountered during parsing. The `Configuration.g4` file defines a method `reportErrorsAsExceptions()` which does this. So if you copy the technique used in this grammar file, you can call:

```
lexer.reportErrorsAsExceptions();
parser.reportErrorsAsExceptions();
```

right after you create the lexer and parser. Then when you call `parser.root()` , it will throw an exception as soon as it encounters something that it can't match.

This is a simplistic approach to handling errors. Antlr offers more sophisticated forms of error recovery as well. To learn more, see Chapter 9 in the Definitive Antlr 4 Reference .

Summary

The topics of today's reading connect to our three properties of good software as follows:

- **Safe from bugs.** A grammar is a declarative specification for strings and streams, which can be implemented automatically by a parser generator. These specifications are often simpler, more direct, and less likely to be buggy then parsing code written by hand.

- **Easy to understand.** A grammar captures the shape of a sequence in a form that is compact and easier to understand than hand-written parsing code.

- **Ready for change.** A grammar can be easily edited, then run through a parser generator to regenerate the parsing code.