Extracted from:

The Definitive ANTLR 4 Reference

The Pragmatic Bookshelf

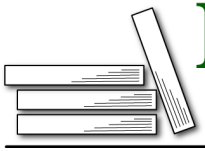
Dallas, Texas • Raleigh, North Carolina

The Pragmatic Programmers

The Definitive ANTLR 4 Reference

Terence Parr

Edited by Susannah Davidson Pfalzer



Pragmatic Bookshelf

Our Pragmatic courses, workshops, and other products can help you and your team create better software and have more fun. For more information, as well as the latest Pragmatic titles, please visit us at *http://pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-93435-699-9
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—September 19, 2012

11.3 Islands in the Stream

The input files we've discussed so far all contain a single language. For example, DOT, CSV, Python and Java files contain nothing but text conforming to those languages. But, there are file formats that contain random text surrounding structured regions or *islands*. We call such formats *island languages* and describe them with *island grammars*. Examples include template engine languages such as StringTemplate and the LaTeX document preparation language, but XML is the quintessential island language. XML files contain structured tags and &-entities surrounded by a sea of stuff we don't care about. (Because there is some structure between the tags themselves, we might call XML an *archipelago language*.)

Classifying something as an island language often depends on our perspective. If we're building a C preprocessor, the preprocessor commands form an island language where the C code is the sea. On the other hand, if we're building a C parser suitable for an IDE, the parser must ignore the sea of preprocessor commands.

Our goal in this section is to learn how to ignore the sea and tokenize the islands so the parser can verify syntax within those islands. We'll need both of those techniques to build a real XML parser in the next section. Let's start by learning how to distinguish XML islands from the sea.

Separating XML Islands from a Sea of Text

To separate XML tags from text, our first thought might be to build an input character stream filter that strips everything between tags. This might make it easy for the lexer to identify the islands, but the filter would throw out all of the text data, which is not what we want. For example, given input `<name>John</name>`, we don't want to throw out `John`.

Instead, let's build a baby XML grammar that lumps the text inside of tags together as one token and the text outside of tags as another token. Since we're focusing on the lexer here, we'll use a single syntactic rule that matches a bunch of tags, &-entities, `CDATA` sections, and text (the sea):

lexmagic/Tags.g4

```
grammar Tags;
file : (TAG|ENTITY|TEXT|CDATA)* ;
```

Rule `file` makes no attempt to ensure the document is well formed—it just indicates the kinds of tokens found in an XML file.

To split up an XML file with lexer rules, we can just give rules for the islands and then a catchall rule called TEXT at the end, to match everything else:

lexmagic/Tags.g4

```
COMMENT : '<!--' .* '-->' -> skip ;
CDATA : '<![CDATA[' .* ']]>' ;
TAG : '<' .* '>' ; // must come after other tag-like structures
ENTITY : '&' .* ';' ;
TEXT : ~[<&]+ ;    // any sequence of chars except < and & chars
```

Those rules make heavy use of the nongreedy .* operation (see *Matching String Literals*, on page ?) that scans until it sees what follows that operation in the rule.

Rule TEXT matches one or more characters, as long as the character isn't the start of a tag or entity. It's tempting to put .+ instead of ~[<&]+, but that would consume until the end of the input once it got into the loop. There's no string to match following .+ in TEXT that would tell the loop when to stop.

An important but subtle ambiguity-resolving mechanism is in play here. In Section 2.3, *You Can't Put Too Much Water into a Nuclear Reactor*, on page ?, we learned that ANTLR lexers resolve ambiguities in favor of the rule specified first in the grammar file. For example, rule TAG matches anything in angle brackets, which includes comments and CDATA sections. Because we specified COMMENT and CDATA first, rule TAG only matches tags that failed to match the other tag rules.

As a side note, XML technically doesn't allow comments that end with ---> or comments that contain --. Using what we learned in Section 8.4, *Error Alternatives*, on page ?, we could add lexical rules to look for bad comments and give specific and informative error messages:

```
BAD_COMMENT1:   '<!--' .* '--->'
                {System.err.println("Can't have ---> end comment");} -> skip ;
BAD_COMMENT2:   '<!--' ('--'|.)* '-->'
                {System.err.println("Can't have -- in comment");}    -> skip ;
```

I've left them out of grammar Tags for simplicity.

Now let's see what our baby XML grammar does with the following input:

lexmagic/XML-inputs/cat.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<?do not care?>
<CATALOG>
<PLANT id="45">Orchid</PLANT>
</CATALOG>
```

Here's the build and test sequence, using grun to print out the tokens:

```
$ antlr4 Tags.g4
$ javac Tags*.java
$ grun Tags file -tokens XML-inputs/cat.xml
[@0,0:37='<?xml version="1.0" encoding="UTF-8"?>',<3>,1:0]
[@1,38:38='\n',<5>,1:38]
[@2,39:53='<?do not care?>',<3>,2:0]
[@3,54:54='\n',<5>,2:15]
[@4,55:63='<CATALOG>',<3>,3:0]
[@5,64:64='\n',<5>,3:9]
[@6,65:79='<PLANT id="45">',<3>,4:0]
[@7,80:85='Orchid',<5>,4:15]
[@8,86:93='</PLANT>',<3>,4:21]
[@9,94:94='\n',<5>,4:29]
[@10,95:104='</CATALOG>',<3>,5:0]
[@11,105:105='\n',<5>,5:10]
[@12,106:105='<EOF>',<-1>,6:11]
```

This baby XML grammar properly reads in XML files and matches a sequence of the various islands and text. What it doesn't do is pull apart the tags and pass the pieces to a parser so it can check the syntax.

Issuing Context-Sensitive Tokens with Lexical Modes

The text inside and outside of tags conform to different languages. For example, `id="45"` is just lump of text outside of a tag but it's three tokens inside of a tag. In a sense, we want an XML lexer to match different sets of rules depending on the context. ANTLR provides *lexical modes* that let lexers switch between contexts (modes). In this section, we'll learn to use lexical modes by improving the baby XML grammar from the previous section so that it passes tag components to the parser.

Lexical modes allow us to split a single lexer grammar into multiple sublexers. The lexer can only return tokens matched by entering a rule in the current mode. One of the most important requirements for mode switching is that the language have clear lexical sentinels that can trigger switching back and forth, such as left and right angle brackets. To be clear, modes rely on the fact that the lexer doesn't need syntactic context to distinguish between different regions in the input.

To keep things simple, let's build a grammar for an XML subset where tags contain an identifier but no attributes. We'll use the default mode to match the sea outside of tags and another mode to match the inside of tags. When the lexer matches < in default mode, it should switch to island mode (inside tag mode) and return a tag start token to the parser. When the inside mode sees >, it should switch back to default mode and return a tag stop token.

The inside mode also needs rules to match identifiers and /. The following lexer encodes that strategy.

lexmagic/ModeTagsLexer.g4

```
lexer grammar ModeTagsLexer;

// Default mode rules (the SEA)
OPEN  : '<'     -> mode(ISLAND) ;       // switch to ISLAND mode
TEXT  : ~'<'+ ;                         // clump all text together

mode ISLAND;
CLOSE : '>'     -> mode(DEFAULT_MODE) ; // back to SEA mode
SLASH : '/' ;
ID    : [a-zA-Z]+ ;                     // match/send ID in tag to parser
```

Rules OPEN and TEXT are in the default mode. OPEN matches a single < and uses lexer command mode(ISLAND) to switch modes. Upon the next token request from the parser, the lexer will only consider rules in ISLAND mode. TEXT matches any sequence of characters that doesn't start a tag. Because none of the lexical rules in this grammar use lexical command skip, all of them return a token to the parser when they match.

In ISLAND mode, the lexer matches closing >, /, and ID tokens. When the lexer sees >, it will execute the lexer command to switch back to the default mode, identified by constant DEFAULT_MODE in class Lexer This is how the lexer ping-pongs back and forth between modes.

The parser for our slightly-augmented XML subset matches tags and text chunks as in grammar Tags, but now we're using rule tag to match the individual tag elements instead of a single lumped token:

lexmagic/ModeTagsParser.g4

```
parser grammar ModeTagsParser;

options { tokenVocab=ModeTagsLexer; } // use tokens from ModeTagsLexer.g4

file: (tag | TEXT)* ;

tag : '<' ID '>'
    | '<' '/' ID '>'
    ;
```

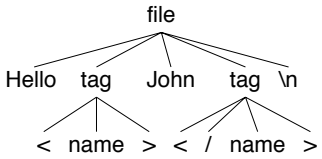
The only unfamiliar syntax in the parser is the tokenVocab option. When we have the parser and lexer in separate files, we need to make sure that the token types and token names from the two files are synchronized. For example, lexer token OPEN must have the same token type in the parser as it does in the lexer.

Let's build the grammar and try it out on some simple XML input:

```
⇛ $ antlr4 ModeTagsLexer.g4   # must be done first to get ModeTagsLexer.tokens
⇛ $ antlr4 ModeTagsParser.g4
⇛ $ javac ModeTags*.java
⇛ $ grun ModeTags file -tokens
⇛ Hello <name>John</name>
⇛ EOF
❮ [@0,0:5='Hello ',<2>,1:0]
  [@1,6:6='<',<1>,1:6]
  [@2,7:10='name',<5>,1:7]
  [@3,11:11='>',<3>,1:11]
  [@4,12:15='John',<2>,1:12]
  [@5,16:16='<',<1>,1:16]
  [@6,17:17='/',<4>,1:17]
  [@7,18:21='name',<5>,1:18]
  [@8,22:22='>',<3>,1:22]
  [@9,23:23='\n',<2>,1:23]
  [@10,24:23='<EOF>',<-1>,2:24]
```

The lexer sends `<name>` to the parser as the three tokens at indexes 1, 2, and 3. Also notice that `Hello`, which lives in the sea, would match rule `ID` but only in `ISLAND` mode. Since the lexer starts out in default mode, `Hello` matches as token `TEXT`. You can see the difference in the token types between tokens at index 0 and 2 where `name` matches as token `ID` (token type 5).

Another reason that we want to match tag syntax in the parser instead of the lexer is that the parser has much more flexibility to execute actions. Furthermore, the parser automatically builds a parse tree for us:



To use our grammar for an application, we could either use the usual listener or visitor mechanism or add actions to the grammar. For example, to implement an XML SAX event mechanism, we could shut off the automatic tree construction and embed grammar actions to trigger SAX method calls.

Now that we know how to separate the XML islands from the sea and how to send tag components to a parser, let's build a real XML parser.