

Parsing XML with ANTLR 3

By Oliver Zeigermann¹

Parsing XML using traditional parsing techniques is tricky most of all because XML is a sick language. Depending on the context, different characters are either keywords or simple text. In XML tags like `<tag attribute="value">` the `=` has a special meaning as well as the `>` while this is of course not the case in plain text.

Additionally, newlines and other white space inside tags have to be ignored, but are part of the text outside of them. You could say, this is the case in most if not all common non natural languages. Considering Java, everything inside quotes is text. That is correct, but this can be handled very simple as a quote is just a single character which has the same meaning in every context. It is thus very easy to write a rule for literal strings. This is different with XML. The character that starts the tag mode is the `'<'`, but only is not in a CDATA section. Even worse the character ending tag mode, `'>'`, can be used in attribute values as well.

For further investigations we restrict ourselves to a simple subset of XML. It will have the same complexity of XML seen from the perspective of parsing, but will make the examples much simpler. It is restricted to tags and parsable text (PCDATA) – no XML declaration, not doctype, no DTD, no processing instructions, no CDATA sections.

In ANTLR 3 there are at least three possible solutions to this problem:

1. faking lexer modes using semantic predicates
2. using island grammars or multiple lexers, one for each context
3. reflecting context in lexer rules and fragments

The rest of the paper requires at least basic knowledge of ANTLR 2, but none of ANTLR 3. It will give you a brief survey of the special new LL(*) parsing features of ANTLR 3 with an emphasis on tree processing guided by the XML parsing example.

Context in lexer rules and fragments

The idea here is to have no parser at all and actually not to produce any token, but rather execute actions inside the lexer. Such a lexer-only solution, has been chosen in the XML parser that is part of the ANTLR 2.x examples. The lexer consists of a single token rule that serves as the entry point of the whole parse. All other rules are mere fragments that do not produce tokens. The only token produced is the one by the entry rule which we simply ignore.

The real trick is that the lexer is not restricted to regular expressions and you can write rules that keep track of closing tags matching opening tags – something impossible with regular expressions in ordinary lexers. This means a complete rule is possible in the lexer (note that all sub rules are lexer rule fragments as well):

```
fragment ELEMENT
: ( START_TAG
  ( ELEMENT
```

¹ Oliver be reached at oliver@zeigermann.de

```

        | PCDATA
        ) *
        END_TAG
    | EMPTY_ELEMENT
    )
;

```

This rule simply says that an element can be an empty element (`EMPTY_ELEMENT`) with nothing inside or an element started with a `START_TAG` and ended with an `END_TAG`. In between there can be a mixture of text (`PCDATA`) and other nested elements.

When looking at the rule for a start tag

```
fragment START_TAG : '<' WS? GENERIC_ID WS? ( ATTRIBUTE WS? )* '>' ;
```

you can instantly notice one drawback of such an approach: It is not possible to filter out ignorable whitespace (rule `WS`) as this would require a stage before the lexer which obviously does not exist. This takes a little bit of elegance from our grammar. Maybe even worse, further processing of the parsed XML has to be done completely outside of ANTLR and its means. This is because a lexer-only solution has no second parser stage and does not produce tokens of information. Because of this, actions to process the XML have to be inserted directly in the lexer grammar. Especially when recording of the tree structure becomes necessary this is quite inconvenient.

Island grammars / Multiple Lexers

One of ANTLR 3's examples uses two grammars for two syntactically very different sections of the input to be parsed. Using a separate grammar for each context as in option (2) really is overkill as the one for text really is a single rule like

```
PCDATA : (~'<')+ ;
```

That alone would rather not recommend this approach, but additionally, feeding a single parser with two lexers turned out to require custom code.

Lexer using Semantic predicates¹

This makes solution (1) look like the most promising. It also is the classic approach where you have a single lexer and a parser that is fed by it. We add a tree generation step to this approach and develop a tree parser for the generated AST. This can be augmented with appropriate actions just like the template tree grammars for the well known ANTLR Java parser.

As we already mentioned, we are faking lexer modes using semantic predicates. We introduce a flag that tells us if we are currently parsing tags or consuming simple text:

```
lexer grammar XMLLexer;
{
    boolean tagMode = false;
}
```

¹ Full source of the XML parser can be found at <http://www.zeigermann.de/xmlSubset.zip>

Now we switch this flag to tag mode every time we see a '<' and switch it back at a '>'. This is obviously correct as each tag is enclosed in these characters. Next we need a semantic predicate for each non-fragment rule that checks the mode and only lets the rule apply in the right one. For the tokens that start tags this may look like:

```
TAG_START_OPEN : { !tagMode }? '<' { tagMode = true; } ;
TAG_END_OPEN  : { !tagMode }? "</" { tagMode = true; } ;
```

The rules that end a tag look like:

```
TAG_CLOSE      : { tagMode }? '>' { tagMode = false; } ;
TAG_EMPTY_CLOSE : { tagMode }? ">" { tagMode = false; } ;
```

Like the above rules these apply to the inside of a tag only:

```
EQ : { tagMode }? '=' ;

VALUE : { tagMode }?
      ( '!' (~'!')* '!'
      | '\!' (~'\!')* '\!'
      )
      ;

GENERIC_ID
  : { tagMode }?
    ( LETTER | '_' | ':' ) ( options {greedy=true;} : NAMECHAR )*
    ;

WS : { tagMode }?
    ( ' '
    | '\t'
    | ( '\n'
      | "\r\n"
      | '\r'
      )
    )+
    { channel=99; }
    ;
```

As already noticed in the island grammar section, the only rule for non tags is simple. In this case we just have to extend it with the checking semantic predicate:

```
PCDATA : { !tagMode }? (~'<')+ ;
```

Finally, we have the fragments used by the other rules to complete our grammar:

```
fragment NAMECHAR : LETTER | DIGIT | '.' | '-' | '_' | ':' ;
fragment DIGIT    : '0'..'9' ;
fragment LETTER   : 'a'..'z' | 'A'..'Z' ;
```

Unfortunately, this approach did not work with ANTLR's early access release 5 release. The problem is that the semantic predicate in the DFA prediction code for PCDATA does not get checked. In effect no PCDATA can be parsed. This bug has been identified and will be fixed in future releases, though.

Anyway, let's turn to the parser grammar now and let's start with the rule for an XML element:

```
element
  : ( startTag
      (element
       | PCDATA
      )*
      endTag
    | emptyElement
  )
  ;
```

Looks familiar? Right! It is the same as the element rule for the lexer-only solution, except written for a parser. Same with the rule for a start tag:

```
startTag : TAG_START_OPEN GENERIC_ID attribute* TAG_CLOSE ;
```

Opposed to the lexer-only solution this looks more elegant as there are no insets for whitespace consumption. This has been done in the lexer already.

One of the cool features of ANTLR 3 is that it can handle arbitrary amounts of common left prefixes. This allows for a rule for the empty element that looks exactly like the one above except for the different closing token:

```
emptyElement : TAG_START_OPEN GENERIC_ID attribute* TAG_EMPTY_CLOSE ;
```

Of course, you could have left factored these two rules into one, but that wouldn't be half as elegant. As you may know, ANTLR predicts the right alternative by traversing the input until it either sees the `TAG_CLOSE` or the `TAG_EMPTY_CLOSE` token using a DFA.

For the sake of completion here is the rest of the rules which do not add anything exciting:

```
attribute : GENERIC_ID EQ VALUE ;
endTag : TAG_END_OPEN GENERIC_ID TAG_CLOSE ;
```

To make clear that this parser uses the token vocabulary from our lexer, the head must look like this:

```
parser grammar XMLParser;
options {
  tokenVocab=XMLLexer;
}
```

Adding a tree parser stage

Part of the critique of the lexer-only approach was that we could not use ANTLR's features to further process the input beyond the lexer stage. With the above grammar we already augmented processing to a convenient parser stage. While this may be enough for most practical examples, there still is quite a lot of noise, i.e. redundant data, to be processed. On the token level this certainly is tag start/end information as well as the `EQ` in the *attribute* rule. On the structural level there is even more noise. When looking at the *element* rule what is an end tag good for? And: Is there a structural difference between an empty element denoted with a start tag directly

followed by an end tag and the special syntax for an empty element? Hardly! So, why not introduce a tree parser stage that only sees the information that it really needs, so that you can concentrate on what to do with this information?!

Even though this is not a common approach, let us start with the tree grammar that looks most natural and simple. Later we can find out how tree construction must look like to produce a tree matching this grammar. To achieve this let us remember that an XML document is a representation of a tree structure. Leaf nodes are either empty elements or text data. Non-leaf nodes are always elements that enclose other elements or text. Additionally, there are attributes and finally names that are associated to elements. It turns out that we can easily write all this in a single rule:

```
element
  : ^( ELEMENT GENERIC_ID
      ( ^(ATTRIBUTE GENERIC_ID VALUE) ) *
      (element
       | PCDATA
      ) *
    )
  ;
```

Cute, isn't it? This rule expresses the whole stammering above. The ^ symbol before an opening bracket indicates a hierarchical tree structure with the first item after the bracket being the root of the tree and the rest being the children.

To complete the tree grammar we just need to add the header that tells the tree parser to use the same token vocabulary as the lexer and the parser and we are done:

```
tree grammar XMLTreeParser;
options {
    tokenVocab=XMLLexer;
}
```

For real end user processing of the tree, you certainly would want to access the text of the tokens. In ANTLR 3 you do this by

```
((Tree)name).toString()
```

where *name* is the label of a tree element like `name=GENERIC_ID`.

We now have to augment the parser we have already seen above with tree construction statements. In ANTLR 3 tree creation and matching syntax is very similar. You can actually declare how the constructed tree is supposed to look like by duplicating the grammar fragment that matches the tree. E.g. for the tree construction of the attribute part we could write:

```
attribute : GENERIC_ID EQ VALUE -> ^(ATTRIBUTE GENERIC_ID VALUE) ;
```

where the tree construction part that comes after the `->` is exactly the same as the fragment seen above. Isn't that cool? It, however, turns out that tree construction declaration by example isn't sufficient in all scenarios. For example we want our end tag to be omitted completely. That's what the ! operator is for. You can either apply it to a full rule by placing it behind a rule name like in

```
endTag! : TAG_END_OPEN GENERIC_ID TAG_CLOSE;
```

or to a single item (*endTag* again) in a rule like in

```
element
  : ( startTag^^
      (element
       | PCDATA
      )*
      endTag!
    | emptyElement
  )
  ;
```

You will notice that this is the element rule we already are familiar with. Next to the ! we can see the double ^^ here. This operator declares *startTag* to be the root of the generated tree. As you can see, this does not only work for a single token, but also for complete sub trees! Additionally, we make use of the automatic tree construction feature already known from ANTLR 2. Left hand items without declaration will simply result in a flat tree node.

Finally, we have the two rules for start tags and empty elements

```
startTag : TAG_START_OPEN GENERIC_ID attribute* TAG_CLOSE
         -> ^(ELEMENT GENERIC_ID attribute*)
         ;
```

```
emptyElement : TAG_START_OPEN GENERIC_ID attribute* TAG_EMPTY_CLOSE
             -> ^(ELEMENT GENERIC_ID attribute*)
             ;
```

which generate a unified tree. As you might have noticed we are using new token names which we have to declare. This results in this grammar head:

```
parser grammar XMLParser;
options {
  tokenVocab=XMLLexer;
  output=AST;
}

tokens {
  ELEMENT;
  ATTRIBUTE;
}
```

That's our whole grammar. Never has tree construction been easier!

Summary

This paper has shown how to parse and process an interesting subset of XML using ANTLR 3. Guided by this example we have seen how a lexer, parser and tree parser work together nicely and how easy tree processing has become. It has been shown how the LL(*) parsing algorithm can significantly help us to create more intuitive and elegant grammars by allowing common left prefixes in multiple rules.