

Efficient parser generator for mixed languages

Shiva Prasad Kintali

*Computer Science Department
University of Southern California
kprasad@usc.edu*

Abstract

This paper proposes some extensions to ANTLR to generate an efficient parser for mixed languages.

1. Motivation

Chip designers reuse intellectual property (IP) from legacy designs and third-party developers. This IP can be written in various languages, such as Verilog, VHDL and C to use existing code written in another language or to gain performance of certain modules. Verification tools, therefore, must understand multiple design languages. One obvious choice is to integrate the existing Verilog and VHDL simulators to implement a mixed-HDL environment. This often results in a complicated mixed-simulator architecture and adds some limitations of its own. We propose some simple extensions to ANTLR in order to parse mixed languages in an elegant way. For the rest of the paper we use examples from mixed language programming using C++ and XQuery so that it will be understood by a wider audience.

Example :

The following example is taken as it is from [3]. Consider the following XML document called "books.xml" as an example:

```
<?xml version="1.0" encoding="ISO-8859-1"?>

<bib>

<book year="1994">
  <title>TCP/IP Illustrated</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

```
<book year="1992">
  <title>Advanced Programming in the Unix environment</title>
  <author><last>Stevens</last><first>W.</first></author>
  <publisher>Addison-Wesley</publisher>
  <price>65.95</price>
</book>
```

```
<book year="2000">
  <title>Data on the Web</title>
  <author><last>Abiteboul</last><first>Serge</first></author>
  <author><last>Buneman</last><first>Peter</first></author>
  <author><last>Suciu</last><first>Dan</first></author>
  <publisher>Morgan Kaufmann Publishers</publisher>
  <price>39.95</price>
</book>
```

```
<book year="1999">
  <title>The Technology and Content for Digital TV</title>
  <editor>
    <last>Gerbarg</last><first>Darcy</first>
    <affiliation>CITI</affiliation>
  </editor>
  <publisher>Kluwer Academic Publishers</publisher>
  <price>129.95</price>
</book>
```

```
</bib>
```

XQuery uses XPath node paths to extract data from XML documents.

For example, the following XQuery :

```
doc("books.xml")/bib/book/title
```

will extract the following data:

```
<title>TCP/IP Illustrated</title>
<title>Advanced Programming in the Unix environment</title>
<title>Data on the Web</title>
<title>The Technology and Content for Digital TV</title>
```

Let's say we want to use the extracted data in a C++ application. We can write a C++ file as follows :

```
#include <iostream>

int main(int argc, char *argv[])
{
    Char * titles[] = doc("books.xml")/bib/book/title;
    // some C++ code that uses titles[][]
    return 0;
}
```

Note that the above C++ file has the syntax of C++ and Xquery. In the following sections we will address the issue of efficiently parsing the files having syntax from different languages (in our example they are C++ and XQuery).

2. Definitions

We define *grammar segment* as a non-terminal in a grammar which can appear (as a non-terminal) inside another grammar. For obvious reasons, the start symbol of a grammar cannot be a grammar segment.

We define a *reference point* in a grammar as the position where a grammar segment (from a different language) can appear.

For example :

Consider the assignment statement `char * titles[] = doc("books.xml")/bib/book/title;` from the above example. Typical grammar for this statement is as follows :

$$\text{assign_statement} : \text{type IDENTIFIER EQUAL} \\ \text{xpath_expression SEMICOLON}$$

In the above example *xpath_expression* is a grammar segment from Xquery grammar and *\$4* is the reference point.

3. Syntactic extensions

We introduce two new keywords *# import* and *# export* (similar to *# token*) to specify the grammar segments imported to and exported from a grammar file.

Example :

Following is the C++ grammar file for the above example. For simplicity we have shown only the required part of the C++ grammar :

```
// File : cpp.g

# token INT          "int"
# token CHAR         "char"
# token SEMICOLON   ";"
# token EQUAL       "="
# token IDENTIFIER  "[a-z]+"
# token NUMBER      "[0-9]+"

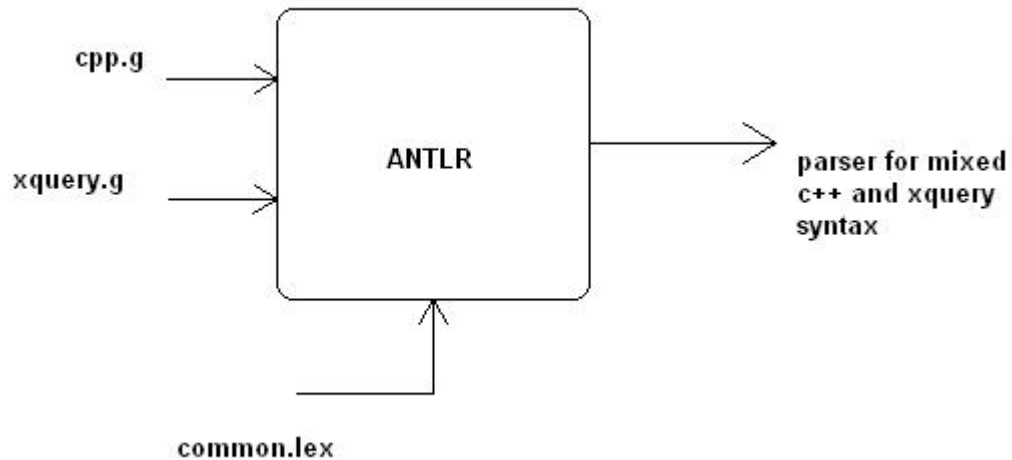
# import xpath_expression

assign_statement :
                type IDENTIFIER EQUAL
                  xpath_expression SEMICOLON
```

4. Handling tokens

Different tokens have different meanings in different languages. For example, *extern*, *cout*, *class* are keywords in C++ but not in XQuery. Similarly *order*, *where* are keywords in XQuery but not in C++. This implies that the token tables must be maintained separately in the individual C++ classes generated by ANTLR. Some common tokens used by all the grammar files (Eg., { , } , ; , + - = > < etc.,) can be kept in a global token table and shared among the C++ classes.

5. ANTLR in mixed mode



We enhance ANTLR to process multiple grammar files and a common lex file and generate single parser. Each file has grammar segments. The grammar segments are all referenced by the top-level grammar.

Common lex file (`common.lex`) specifies the tokens used by all the grammar files. For example, it can contain symbols like `{ , } ; , + - = > <` etc., Tokens belonging to a single grammar can be declared in the individual grammar files.

The non-terminals declared using `#import` and `#export` are the grammar segments imported to, exported from other grammar files. A separate parser (C++ class with parse functions for each grammar segment) is generated for each grammar file. Individual grammars are connected to each other only at the reference points. Undefined Grammar segments can be detected at the end of processing all the grammar files, and proper error messages can be generated.

To simplify things further, each input grammar file can specify just the grammar segments, the tokens and the import/export declarations. Generating the `main` function (for top-level grammar) and the individual C++ classes for each grammar can be done by ANTLR itself. We will see an example in the next section.

To run ANTLR in mixed mode we add the following command line options :

-mixed	To run ANTLR in mixed mode
-top	To specify the top level grammar file
-common	To specify the common lex file

Input files :

common.lex	Common tokens used by C++ and XQuery
cpp.g	C++ grammar file
xquery.g	XQuery grammar file

In mixed mode, ANTLR is run as follows :

```
antlr -mixed -top cpp.g xquery.g -common common.lex
```

6. Generated parser

Following is the *skeleton* of the generated file for cpp.g :

```
// File : cpp.cpp
#include <stdio.h>
#define ANTLR_VERSION 132
#include "tokens.h"
#include "AParser.h"
#include "DLexerBase.h"
#include "ATokPtr.h"
#include "DLGLexer.h"

#include "Cpp.h"
#include "XQuery.h"

class ANTLRToken : public ANTLRAbstractToken {
    // Body of class ANTLRToken
};
```

```

int main()
{
    // code to initialize CppParser
    CppParser.assign_statement(); // start parsing at rule 'assign_statement'
    return 0;
}

void CppParser::assign_statement(void)
{
    -- match and consume type
    -- match and consume IDENTIFIER
    -- match and consume EQUAL
    // at this point we know that xpath_expression is imported
    // we generate code to call the XqueryParser to parse xpath_expression

    -- instantiate the parser for xpath_expression

    XqueryParser->xpath_expression();
    delete xqueryParser;

    -- match and consume SEMICOLON

fail:
    syn(zzBadTok, (ANTLRChar *)"", zzMissSet, zzMissTok, zzErrk);
    resynch(setwd1, 0x1);
}

```

7. Conclusion

The above approach can be extended to process more than two grammars. ANTLR, with the extensions suggested above, finds its major application in electronic design automation (EDA) where there is a need to develop hardware simulators for mixed languages. Also, it can be used in mixed language programming with C++ and FORTRAN and C++ and XQuery.

References

- [1] *ANTLR : A predicated-LL(K) parser generator* Terence J. Parr, Russell Quong.
- [2] Source code of pccts1.33 written by Terence J. Parr, Will Cohen, Hank Dietz, Russell Quong.
- [3] *XQuery tutorial* from <http://www.w3schools.com/xquery/>