Using XML as an Intermediate Form for Compiler Development

Roy **Germon**

Abstract

Many applications of XML require transformations between XML and other formal grammars. This case study, which uses XML in the implementation of a programming language compiler, illustrates techniques for translating other formal grammars to XML, and XML to other formal grammars.

1. Introduction

The paper describes a small programming language prototype project, and the background theory that underlies the implementation techniques. The project involves compiling a simple block-structure imperative programming language similar to Pascal or C. XML is used as the encoding format between the parser and the code-generator. The parser compiler phase is implemented as a stand alone OmniMark program which parses the source language and outputs an XML document representing the program. The code generation phase (or back end) is a second OmniMark program which translates the XML encoded parse tree into an abstract machine language similar to assembler languages for real machines.

Although strictly speaking, the paper is about compiler implementation, the techniques are far more broadly useful. The first pass (parser) is all about overlaying a structure on a serial encoding of data. Conversion of non-XML to XML requires the same techniques. The last pass of a compiler involves mapping XML to a non-XML format. This also occurs more broadly across many problem domains. Content publishing in particular requires markup-to-presentation format conversions.

XML was used as the intermediate form for this project for a number of reasons. First, since XML is a text based encoding, the tree encodings would be human-readable. This would allow experimenting with optimization techniques, and one would be able to read the resulting trees at all stages in the process. Second, I was interested in exploring parallels between the traditional data conversion problem domain and the programming language implementation problem domain. Finally, XML is designed to encode trees, and XML tools are designed to manipulate those trees. In other words, XML and XML tools are well suited to implementing the back end portions of a programming language compiler.

2. Formal Grammars

A grammar is a notation for describing the syntax of a language. Typically a grammar is represented as a series of productions. A production consists of a nonterminal called the "left side"

of the production, and a sequence of tokens (terminal symbols, or simply "terminals") and/or nonterminals called the right side of the production. One nonterminal is designated as the start symbol. The start symbol is the top level symbol of the grammar. Any legal instance of a language must satisfy the right hand side of a production which has the start symbol as its left hand side. Terminals are the tokens that occur directly in the language itself...numbers, words and symbols. Nonterminals represent productions in the grammar, and thus levels in the hierarchical structure of the information.

Example:

Terminals appear inside square brackets.

```
term   ::= term ["*"] factor
term   ::= term ["/"] factor
term   ::= factor
factor ::= [number]
factor ::= ["("] term [")"]
```

Grammars are very similar in their function to DTDs or XML Schemas. Nonterminals can be thought of as elements. Terminals are analogous to data content. Productions are like element definitions where the element name is the left side and the content model is the right side. Finally, the start symbol is like the document element in a DTD.

Parsing involves determining whether or not an instance is described by a grammar, and if so, what hierarchy of productions from the grammar describe the instance. One of the most obvious ways of parsing a grammar is to start with the start symbol, and look ahead at tokens from the instance to see if you can figure out which production of the start symbol you might be able to match. Once you have selected a production, read the right hand side from left to right, making sure any terminal symbols in the production match tokens in the instance, and recursively apply this process to any nonterminals. This is known as LL parsing. Grammars which can be parsed using this technique are known as LL grammars. The maximum number of tokens you need to look ahead at to choose a production can vary from grammar to grammar. An LL grammar is called an LL(K) grammar when it requires K tokens of lookahead

LL(1) Example:

Invariant terminals are quoted inside square brackets.

```
phrase ::= ["eat"] stuff
stuff  ::= ["lunch"]
stuff  ::= ["my"] ["shorts"]
```

This example is LL(1) because right off the bat, you can tell from the first symbol in the input whether you are in the production for phrase or not. Either you have the word "eat", or you are in error. Move ahead to the next symbol, and you again can make the next descision immediately. If it is "lunch" you are in production 2. If it is "my" you are in production 3. Anything else is an error, and you never need to look ahead for the symbol "shorts" to make the decisions.

LL(3) Example:

```
phrase ::= ["eat"] ["my"] ["shorts"]
phrase ::= ["eat"] ["my"] ["lunch"]
```

Right off the bat you need to see three tokens in the input to know which production to choose, making this LL(3).

Some grammars, like the term and factor arithmetic grammar given earlier, can not be parsed using this top down approach. This is due to left-recursion in the grammar. Notice that the first symbol on the right hand side for producing a term can be a term. This recursion can cause an infinite loop when you try to parse instances of the language using the top down method.

A parsing technique which doesn't suffer from infinite recursion when parsing left recursive grammars is the LR or bottom up parsing method. Using tokens from the instance, decide which production is being satisfied currently. Productions recognized are pieced together into higher productions up the grammar until finally a production for the start symbol is recognized. Grammars which can be parsed this way are called LR grammars. The grammar given earlier for infix multiplication and division is a typical LR grammar.

The content model for an element is a good analog for a grammar right hand side. The element itself is the left side nonterminal. Actually, the start and end tags are nonterminals in the right hand side of the equivalent grammar. Since a start tag uniquely identifies the element, and since any element has only a single element definition and content model, it follows that XML documents — for the purposes of content model validation — fit an LL(1) parsing model.

So if XML is easy to parse, why not just start with XML in the first place?

The structure of information itself carries a lot of information. That structural information needs to be encoded in some way. XML uses tags and the nesting relationship of elements to indicate structure. Traditional languages rely more heavily on context to encode the structural information. This is an extremely compact notation. As such it makes the data easier to find in the metadata, and makes authoring and reading simpler for humans. Context is the most compact markup of all.

Example:

Typical programing language:

```
x := 1 + 2
```

XML:

```
<assign>
   <LValue><variable>x</variable></LValue>
   <RValue>
      <add><constant>1</constant>
           <constant>2</constant>
      </add>
   </RValue>
<assign>
```

3. Into XML

Parsing is the essence of data conversion. Converting any structured information into XML involves two processes: Parsing the original format, and mapping the components of the original format into XML.

The most obvious way of implementing an LL parser makes use of the recursive descent technique. This is a functional technique where each non-terminal is basically represented by a function. The function uses lookahead to decide which production of that non-terminal matches the text in the instance, and then attempts to satisfy the right hand side of that production by consuming terminals and calling production functions to satisfy non-terminals in a left-to-right fashion. The parsing process is launched by calling the production function for the start symbol of the grammar. The examples using OmniMark use find rules to emulate a functional style because nested pattern matching allows for combining the lexical layer (recognition of tokens) into the parsing layer.

Example:

```
Block       ::= ["do"] Declaration* Action* ["done"]
Declaration ::= ["declare"] Variable
Action      ::= ["output"] Variable
```

In OmniMark:

```
find "do" white-space+
   using group declarations do
      submit #current-input
   catch done-nesting
   done
   using group actions do
      submit #current-input
   catch done-nesting
   done
   assert #current-input matches "done"

group declarations
find "declare" ...
   ; act on the declaration
find ""
   throw done-nested
```

The direct translation of a grammar to an implementation is a tremendous advantage of LL grammar parsing via recursive descent. As was noted earlier, some grammars require LR techniques. Programming language compiler implementors in particular need to be aware of LR parsing techniques. The sophistication of LR techniques is generally not needed outside of programming language parsing. Details on LR parsing can be found in compiler texts such as [Compilers: Principles, Techniques and Tools, by Aho, Sethi and Ullman].

The mapping of a formal grammar to XML is a fairly simple matter. The naive approach is to map each nonterminal to an XML element. The right hand side of each production for that nonterminal is an alternative content model (think "or" groups). Invariant terminals can usually

be discarded since they add no information. Variant terminals like numbers and variable names become #PCDATA content. Where a production contains more than one variant terminal, you can not map the grammar directly because you would have two distinct #PCDATA elements in the content model of a single XML element. Variant non-terminals can be wrapped in nonterminal productions to solve this problem. For example instead of:

```
set-action  ::= ["set"] [name-token] ["to"] [name-token]

set-action  ::= ["set"] destination ["to"] source
destination ::= [name-token]
source      ::= [name-token]
```

If all variant terminals are wrapped, then the translation is easy. Some nonterminals will have right sides which consist entirely of a single variant terminal. These nonterminals become elements with #PCDATA as the content model. Any other nonterminal will have on the right side some mix of invariant terminals and nonterminal symbols. The invariant terminals get stripped, and the content model of the XML element is the remaining terminals.

4. From XML

Translating from XML into other encodings is a much easier task than converting into XML because XML explictly expresses the structural information of the document. XML is a meta-language that allows the creation of distinct languages which use a common syntax to encode the structure of a document. XML parsers recognize the structure of any well formed XML document. The parsing problem is fully solved and handled on your behalf by the XML parser. Exploiting structure is an easier task than discovering it.

At the design level, the only thing that remains is to map components of the XML structure to the target language. This process is highly variable depending on the nature of the target language.

In the implementation, template-based programming can be a useful technique for translating out of XML into other formats. Each node in the tree is associated with a template with placeholders. Processes which act on the content of the node act essentially as data generators which produce fill-in values for the template.

An issue that compiler developers have to deal with that has a strong analog in other areas is cross referencing within the output of an XML to non-XML translation. In language compilation this happens when forward referencing of objects occurs, or when forward jumps in the code table need to be generated. Backpatching is a technique that is commonly used in the compiler world for accomplishing this. Backpatching involves writing placeholders into the output, and replacing them with real values during a later phase, or when the real data is processed. The referent mechanism in OmniMark implements this process for the programmer. Another possible solution is generating a template file in one pass, and resolving the placeholders explicitly in a second pass. A third solution involves retaining the "output" in a random access data structure such as an array or a table, and fixing the backpatch points directly when the data becomes available.

5. A Tree Interpreter

In addition to the compiler, I have also implemented an interpreter for the intermediate XML tree form. This of course introduces significant interpretive overhead for executing programs. On the other hand, the implementation was extremely straight-forward, requiring only a few hours to implement.

The implementation is in OmniMark, and uses a streaming XML approach. Streaming introduces certain challenges in this role. In particular, iterative constructs require revisiting certain sub-trees repeatedly. This was accomplished by capturing the sub-tree in XML form in a buffer, and repeatedly applying the execute processor to it until the loop exit conditions were met.

Another challenge of the streaming approach for tree execution is that the sub-elements of an element might not be in an order which lends itself to execution in a streaming pass. Careful design of the XML can help solve this problem. Where the XML language itself can not be modified to solve this problem, the capture-execute approach used for loops can also be applied to grab the out-of-order components, and then execute them in the correct order. DOM based processing avoids these issues, since you can choose to process any path through the tree in any order.

6. Conclusions

At the time of this writing performance statistics on my implementation are not available. Experience and observation lead me to believe that the XML based implementations will be considerably slower than conventional implementations. For compilers of small to medium sized languages, poor compiler performance is generally acceptable provided run time performance is adequate. Using XML does bring some benefits to offset poor performance. In particular tools for processing XML can reduce the time for compiler implementation. Other benefits listed in the first section of this paper are less tangible, but contribute to the viability of using XML as an intermediate form.

In spite of performance concerns, XML is a workable intermediate form for certain compiler applications. Certainly for prototyping, using XML tools rather than low or intermediate level programming languages is an advantage in developer time. For production compilers where compiler performance may be an issue, the cost/benefit is unclear, but there are scenarios where this approach can be beneficial overall.

The XML tree interpreter on the other hand is not likely appropriate as a production run time environment for executing programs. For scripting languages that are small, there may be a case to be made, but in general run time performance is a significant concern. The XML tree interpreter is however a very inexpensive prototyping technique.

The techniques used in this paper are compiler techniques, but are generally applicable in data conversion problems of all types. The simplest techniques (LL parsing and template-based output) alone are very powerful for general data conversion work.

Finally, all OmniMark source code and compiled binaries for Windows, Solaris and Linux are available for all the components from http://www.omnimark.com/etcetera/index.html

Biography

Roy **Germon**
OmniMark Technologies
Ottawa
Canada

Roy Germon is a Senior Software Developer at OmniMark Technologies. He has been a member of the XML Working Group. He is currently leads programming language implementation and maintenance at OmniMark.