Using XML in a Compiler Course

D. Robert Adams
Department of Computer Science and
Information Systems
Grand Valley State University
Allendale, MI 49401-9403
adams@csis.gvsu.edu

Christian Trefftz
Department of Computer Science and
Information Systems
Grand Valley State University
Allendale, MI 49401-9403
trefftz@gvsu.edu

ABSTRACT
In this paper we describe how XML can be introduced into a compiler construction course. We make the case that XML and compilers have much in common, and that introducing XML into a compiler course makes sense. We then go on to demonstrate how XML was used in two recent compiler courses. Finally, we discuss the tradeoffs of using an XML-based project rather than a traditional programming-language project.

Categories and Subject Descriptors
K.3.2 [Computers and Education]: Computer and Information Science Education

Keywords
compilers, XML

1. INTRODUCTION
XML is a technology that is being widely adopted in many different fields [6]. Given XML’s growing importance, it is reasonable to introduce students to XML in the curriculum of undergraduate computer science programs [3, 4]. Many courses seem appropriate in which to introduce XML, e.g., data structures, databases [7], compilers, etc. However, a compiler course has several objectives that closely mirror fundamental aspects of XML, and makes a compiler course especially appropriate for the introduction of XML:

• In a compiler course, a student gains an understanding of the lexical analysis and parsing phases of a source file, i.e., determining if the file is free from syntax errors and matches the rules of a grammar. XML has the concepts of a well-formed and valid document. “Well-formed” means it is free from XML syntax errors. “Valid” means the document matches a specific grammar.

• In a compiler course, students learn how a grammar specifies the syntax rules of a specific language. For an XML document to be valid, it must match a specific grammar called a Document Type Definition (DTDs).

• Parsing a programming language involves producing a derivation (parse) tree. Many XML API’s build a similar tree that corresponds to an XML document.

• After the derivation tree is produced, students are often expected to output intermediate code (e.g., pseudomachine-language) equivalent to the source. An XML document is often viewed as an intermediate language between applications. Thus the translation of an XML document to a format suitable for a specific application is common [2].

Conversely, an understanding of compiler techniques will help students create applications that use XML documents. For example, observing the derivation tree produced by the parsing process will enable students to choose the correct operations in available API’s in order to traverse an XML document tree. Also, when using the Document Object Model (DOM) for processing, an understanding of the fundamentals of parsing will allow students to appreciate the options that they have available when processing a document.

This paper has two goals. First, we argue that XML should be used as either a replacement for or as an addition to a tradition compiler project. Second, we describe our experiences of introducing XML in two different compiler courses we have taught recently.

The rest of this paper is structured as follows: Section 2 contains a description of how XML was actually used in the courses. The projects that were assigned to the students are described in Section 3. Our answers to potential arguments are provided in Section 4. Conclusions and future work are in Section 5.

2. PEDAGOGY
We have used XML in two recent compiler courses. The first was a short summer course and the second was a regular one semester course. This section describes how XML topics were introduced in the classroom. In the next section we describe how students put their XML knowledge to practical use.

The XML universe is vast [5], and it is impractical to teach everything about XML in a single course. Rather, it is necessary to select a subset of key XML elements to
introduce, allowing the student to build upon that subset in later courses, or perhaps in their jobs after graduation.

Our basic course outline was essentially the same as any traditional compiler course, but supplemented with lectures on XML. Lecture topics ranged from lexical analysis to code generation. Below, we summarize how and where XML was introduced.

- Basic notions about XML were introduced at the very beginning of the course. These ideas included the background of XML, why and where XML is used, etc.

- After introducing the concepts of regular expressions and grammars, DTDs were covered in detail. Lectures covered what DTDs are, and how a DTD represents the elements and attributes of an XML language.

- The difference between a “well-formed” and a “valid” document was explained after the syntactic analysis session. Lectures involved demonstrating how deciding if an XML document is well-formed and valid is the same as looking for syntax errors in a programming language.

- The Document Object Model (DOM) for processing was introduced after the sessions that introduced derivation trees. DOM processing is closely related to traversing a derivation tree.

Different books were used in the two courses we taught: In the short course the “Dragon Book” [1] was used and in the semester long course Appel’s book [8]. Neither book contains any material about XML. Students were directed to on-line tutorials on XML to supplement the material on XML covered in class.

3. PROJECTS

Students were required to implement an XML compiler. The project consisted of several phases that closely mirror the phases of development in a traditional compiler project. In this course, students were permitted to develop their own specification for the project. The idea was that students would be more motivated to work on a project of their own devising, than a project given to them by the instructor.

In the first phase, students were asked to write their own DTD. They were asked to use a minimum of 12 elements and to include at least one attribute. Along with the DTD, they submitted an instance of a document that conformed to the DTD they wrote. They used the open-source tool xmllint to validate their sample document.

This phase of the project is analogous to a student developing their own grammar in a traditional compiler project. Unfortunately, in many compiler courses students are asked to write a compiler for a grammar given to them, and rarely have the opportunity to develop their own.

The next phase consisted of writing a lexical analyzer for their sample document. Their lexical analyzer was required to properly tokenize their sample document, isolating tags, attributes, data, etc.

A syntactic analyzer was requested in the next phase. Their syntactic analyzer used the lexical analyzer from phase two, and determined whether or not a document conformed to the grammar originally described in the DTD. Once again, students used xmllint to verify that their compiler detected the same errors as a “real” XML parser.

In phase four, students were asked to create and traverse a parse tree. In the semester-long course, students were also asked to perform “type” checking on their document. Because they were not parsing a programming language, their compiler did not do true type checking. Rather, their XML compiler performed two kinds of checks. First, their compiler had to ensure that attribute values in the document matched what was specified by the original DTD. For example, an ID attribute’s value must be unique in a document. Second, the student’s compiler had to perform application-specific checks on element data. For example, an element that supposedly contained a phone number must be checked to see if it looks like a phone number.

In phase five, students where asked to traverse a parse tree, and this time to process the document. In the short-course, this involved writing a Java program that used the DOM API. In the semester-long course, because of the increased time, students did this twice: first by writing a Java program to traverse the derivation tree “by hand”, and second to use the DOM API to do the same thing. In all cases, the end result was an application that converted XML into some other format (e.g., HTML).

The two courses used two different compiler toolsets for building the XML compiler. In the short version of the course, students used flex, bison and C. This required that students explicitly build a parse tree in phase four. In the semester long version of the course, SableCC and Java were used, and because SableCC returns a parse tree, students were not required to explicitly build a tree.

4. DISCUSSION

The inclusion of XML into a compiler course is not without cost. In a traditional compiler course, students often spend the entire semester developing a compiler for a programming language. The inclusion of XML in the curriculum means that will no longer be possible. Below, we attempt to predict and answer criticism of our XML-based approach.

A typical student compiler project involves four phases: lexical analysis, syntactic analysis, type checking, and code generation. As stated above, an XML-based project certainly involves the first two of those four.

Naturally, an XML document doesn’t need to be type-checked in the same way as a programming language does. However, the XML-based project emphasizes the same basic idea: traversing a parse tree and evaluating some property of the nodes. Thus, we feel that students gain an understanding of the type checking process, even though they’ve not actually implemented a type checker.

The XML project is obviously lacking a code generation phase. However, we feel that this isn’t a great loss. We certainly talk about code generation in lecture, but students don’t get “hands-on” experience in actually doing it. We would argue that most students are much more likely to use XML in their professional lives, either writing software to process XML documents or writing DTDs, than to write software to generate machine code. An XML project seems more relevant for the professional future of students.

The use of XML has benefits not found in a traditional compiler project. First, we chose to allow each student to develop their own grammar. This is rarely, if ever, done in a
compiler project. But with XML, a student gets experience in developing a grammar, fine-tuning it, and then trying to live with the limitations imposed by DTDs and/or LR(1) grammars.

Second, the use of `xmllint` gives students the ability to independently verify that their compiler is working correctly. Given an XML document with errors, they can test whether or not their compiler detects the same errors as `xmllint`.

5. CONCLUSIONS

The two compiler courses that we have taught using XML have been successful on two fronts. First, based on their performance on exam questions, students grasped the fundamental areas of compiler construction, even without building a compiler for a programming language.

Second, students gained an understanding of what XML is and how it can be used by an application. Some students had some previous exposure to XML before taking these courses, and many of them had misconceptions about what XML is. For example, the order of elements in the document does not matter. This, and other misconceptions disappeared after taking the course.

Third, students were able to integrate their knowledge of compilers and XML processing via the project. Concepts and ideas about a programming language compiler learned in lecture were applied in a different context (XML processing).

Two fundamental areas of XML were not included in either course. First, XML schemas, as an alternative grammar, where not discussed. However, given that students know how a DTD works, then learning to use schemas should not be difficult.

Second, XML has two well-known API’s, DOM and the Simple API for XML (SAX). In class we only discussed DOM. Again, given a thorough understanding of DOM, student should be able to learn about SAX with little difficulty.

The use of XML in the compiler class has been a success, and we foresee using XML in the course in the future.

Acknowledgments: We would like to thank Larry Kotman for his comments and suggestions.

6. REFERENCES


