Building an XQuery Interpreter in a Compiler Construction Course

Sara Miner More Tim Pevzner Alin Deutsch
Scott Baden Paul Kube

{more, tpevzner, deutsch, baden, kube}@cs.ucsd.edu
Department of Computer Science & Engineering
University of California, San Diego
La Jolla, CA 92093-0114

ABSTRACT

For two years, we have been teaching a quarter-long compiler construction course where students implement an interpreter for a variant of the XML query language XQuery. Our goal is to motivate students' interest in the course by exposing them to an interesting and powerful new language which they see as relevant to potential future experiences.

In this paper, we first explain the workings of the course itself, and then describe some pedagogically interesting variants of the XQuery language. We close with a discussion of challenges faced and conclusions.

Categories and Subject Descriptors

K.3 [**Computers & Education**]: Computer & Information Science Education—*Computer Science Education*

General Terms

Languages, Theory, Design

Keywords

Compiler construction, XQuery, XML, Capstone courses

1. INTRODUCTION

For years, our department taught a typical two-quarter sequence of courses on compiler construction. This sequence represented an integrated, capstone experience which spanned the two quarters. The assigned projects afforded students experience following a detailed specification, writing modular software and testing code. However, faculty described several problems with the existing situation. First, it was difficult to motivate students to learn about compilers. Most students were aware that it was unlikely that they would write a compiler or interpreter for a source language in their later careers, and seemed to devote less time and energy to the compiler sequence than to subjects which they perceived as more "relevant". Secondly, projects were completed in teams, and the second quarter project relied on work completed in the first quarter. This situation led to difficulties for students whose teammates did not remain enrolled in the course for the second quarter (either changed major, or took the second course during a later quarter). Furthermore, at the beginning of the second quarter, students were not starting off on even footing - those who had performed very well in the first quarter had a stronger base of code from which to begin the second half of the project.

Therefore, two years ago, with support from the department and university, we undertook a major change in our compiler sequence. First, we separated the courses so that the project in the second quarter did not depend on work completed in the first quarter. Then, we changed the focus language in the first quarter of the sequence in an attempt to improve student motivation for the course. Additionally, we tried to retain the integrative, capstone nature of the student experience, albeit only over a single quarter. To accomplish these goals, we selected the the World Wide Web Consortium's (W3C [6]) XML query language XQuery [9] as our new focus language, and asked students to build an interpreter for a subset of XQuery. In this paper, we describe our experiences using XQuery as the focus language in the first quarter of our compiler construction sequence.

2. WHY XQUERY?

As mentioned above, the selection of XQuery as a new focus language for this course was, in part, a response to faculty perception that the existing sequence of compiler construction courses was difficult to motivate. As Debray [11] noted, most computer science majors do not write a source language compiler or interpreter after graduation, so students often feel that the sequence of compiler courses is less relevant to contemporary practice. To mitigate this problem, we sought to relate the study of translators to emerging technologies that would be relevant to students' likely postgraduate experiences. Many of our undergraduates are interested in databases, and have expressed a desire to learn about XML. Since understanding the syntax and semantics of XQuery would require first obtaining a basic understanding of XML, we reasoned that students might become more enthusiastic about the subject matter from the start.

Secondly, we wanted students to come away from the compiler sequence with the sense that, with minimal effort, they would be capable of writing a compiler for a fairly complex language. Requiring them to work with XQuery, a rich language currently undergoing standardization, would

SIGCSE'05, February 23–27, 2005, St. Louis, Missouri, USA.

leave them with confidence about their ability to apply their knowledge to different languages. Admittedly, there is not enough time in a ten-week quarter to expect students to implement an interpreter for the entire language, so we ended up restricting the language somewhat.[1] However, we use a language subset that retains the flavor of the full language.

Furthermore, an XQuery interpreter lends itself nicely to a modular XML-centric design, as we will describe below. This design helps to reinforce students' newly gained knowledge about XML, allows for automated grading, and gives the instructor flexibility in terms of requirements.

3. DESCRIPTION OF THE COURSE

3.1 Concepts Covered

The course in question traditionally focused on lexical analysis, syntactic analysis, error analysis, type-checking and, to some degree, syntax-directed translation. (The second course in the sequence covers code generation and optimization, which are not included in the first quarter.) With the change to XQuery as our focus language, we did not want to sacrifice any of this conceptual core. However, we did now need to devote some time to teaching students about XML and the Document Object Model [2], as well as about XQuery itself. As a result, we reduced the amount of time spent on the theory of lexing and parsing. However, we do provide students with useful exposure to real-world language specifications and the process of dealing with an evolving standard. Furthermore, we introduce students to tools for automating the lexer and parser generation process, which they may encounter again after graduation.

In addition, in order to foster interest in the language and to help students relate it to potential real-world experiences, we invite an XQuery expert to give a guest lecture about how XQuery is used in practice and where it is headed.

3.2 Student Teams

We have a large number of CS majors in our department, and, as a result, even upper-level major courses have large enrollments. In the past two years, we have taught this class with sizes ranging from 75 to 250 students. Students are permitted to work in teams of two on the task of constructing the interpreter. (Teams typically remain together for the entire quarter, though this is not strictly required, and some students elect to work individually for some or all of the quarter.) Working in pairs ameliorates the problem of large enrollments by reducing the amount of time spent grading, and it carries with it the added benefits of encouraging student interaction, and allowing us to assign a more complex project. In fact, for most students, the project completed in this course is by far the largest programming project they have undertaken. They are also given much more flexibility and independence during the quarter than they have encountered in previous courses. Unfortunately, this freedom causes some students to struggle; we will discuss this problem further in Section 5.1.

[1] An alternative would be to create a simpler language such as MinimL [10], which was built to contain just enough features to be interesting, but not so many that students are overwhelmed. We opt for the subset approach because we feel that students are more excited about a language which they might imagine using in future employment. We discuss various subsets of the language in Section 4.

3.3 Phases of Interpreter Construction

The task of building an interpreter is often broken into three phases. We follow this traditional sequence of phases as follows. First, the lexer translates source code into a stream of tokens. Secondly, while enforcing syntax rules, the parser translates this stream of tokens into an abstract syntax tree (AST) or some other intermediate representation. Finally, the semantic engine evaluates the AST, performing type checking, and executes the code it represents.

In our XML-centric project design, XML output is used during intermediate phases. Specifically, in the first phase, students write driver programs for their lexers which, when given XQuery input, output the stream of tokens as an XML document. In the second phase, their driver programs output an XML representation of the abstract syntax tree, a natural fit with the tree-based structure of XML documents.

3.3.1 Lexer

At the beginning of the lexer portion of the course, students are introduced to the Java-centric JFlex [3] scanner-generator tool. They are then provided with a Lexical Specification documenting the requirements of their lexer. The goal of the lexer itself is to generate a stream of tokens found in the given input, including line and column information from the input file. A lexer driver extracts this token stream and outputs it in XML format. If any errors are found, they are reported in XML as well.

An interesting issue in the lexer phase (which also arises in Pascal compilers) is that XQuery comments are delimited by matching `(:` and `:)`. That is, comments may be nested. In the traditional division of labor within an interpreter, the lexer identifies comments and strips them from the input rather than tokenizing them, so that the parser is not burdened with them. However, a lexer which is a finite automaton is not sufficient for recognizing nested comments. Without the power of a stack, an XQuery lexer cannot reliably determine how to tokenize the text occurring after `:)`. This feature of XQuery breaks the traditional clean modular structure of an interpreter, but exposes students to real-world language issues. In addition, some XQuery language constructs require the use of lexical states, so we provide students with a Lexical State Specification.

3.3.2 Parser

We begin the parser portion of the course with an introduction to the Java-centric CUP [1] parser-generator. Students are given a Syntax Specification which contains the grammar rules for the language. (As mentioned above, the language is a subset of the full XQuery language, with minor modifications.) Their task is to generate an abstract syntax tree (AST) representing any given source input which conforms to the grammar. We also provide a specification for the representation of the AST, which we call the XQueryX Specification. (It is loosely based on an early version of the W3C's XQueryX specification document [8], which specifies a standard XML representation of an XQuery program.) In the case that the program contains syntax errors, the parser generates an XML file describing the location of the error.

There is one practical issue to mention here. When CUP is used to generate a parser, it creates a file called *sym.java* which contains list of token names and integers representing those tokens. The symbolic constants contained in this file are actually necessary for the scanner produced by run-

ning JFlex. Since the lexer phase of the project occurs prior to the parser phase, students have not created their own *sym.java* file yet. To rectify this problem, we provide students with the *sym.java* file generated from our reference parser to use during the lexer phase. Furthermore, parsers generated from different CUP files will generate different mappings in the resulting *sym.java* file. Providing a single file from which all teams can work avoids the problem of grading lexers which output different integers corresponding to a particular token. In the parser phase, however, students abandon the provided *sym.java* file from the earlier phase and run CUP to create their own. Since the output of the parser does not include tokens, the internal integer representation for a particular token does not affect the output.

3.3.3 Semantic Engine

In the final phase of the interpreter construction, the goal is evaluation of ASTs generated by the parser. Students are given a full Semantic Specification detailing the meaning of language constructs and its type system.

Evaluation of some XQuery constructs is quite straightforward for students, e.g., literals or the *if-then-else* statement. A central construct in an XQuery program, however, is a *FLWOR expression*, which supports iteration and binding of variables to values. Students are unfamiliar with the semantics of this expression, so implementing its evaluation takes some time. In particular, the *order-by* clause of a FLWOR is most troublesome for students. Furthermore, *constructors* and *path expressions* are constructs which allow for the creation and traversal, respectively, of nodes within XML document trees, and seem to be the most troubling for students to implement. In addition, the preexisting XML and XPath[7] schema on which XQuery was based introduce some unusual constructs into the language, adding to the complexity of the assignment. Luckily, the Java standard packages org.w3c.dom, org.xml.sax, and javax.xml.parsers provide much functionality for manipulating XML documents and Document Object Model interfaces. To assist students further, we supply them with our own utility package for manipulating XML in Java that provides much of the components of path expression functionality. Students are left with the task of combining these different given components of a path expression in the appropriate way.

Type checking is also performed in this phase. XQuery has a detailed hierarchical structure of predefined datatypes. We include a subset of these, consisting of what we believe is a manageable number of types (approximately 15, about half of which are the various XML node types). In addition, we simplify the language further by disallowing user-defined types. It turns out that implementing an XQuery interpreter in Java is convenient, as there is a nice mapping of most XQuery types to Java types. Furthermore, we encourage students to make use of Java inheritance when implementing the hierarchical type system of XQuery.

Because some students begin the course without prior knowledge of XML and nearly all students begin without prior knowledge of XQuery, we spend some time introducing these languages during lecture. In addition, we have found it useful to include an initial programming assignment that forces students to use these languages. Students write several fairly short XQuery programs which are intended to acquaint them with unfamiliar language constructions such as FLWOR expressions, constructors, path expressions and predicates. Although it delays the beginning of the lexer assignment, we find that familiarity with XQuery is very helpful for students during the semantic analysis phase.

3.4 The Grading Process

The XML-centric nature of the interpreter project allows fairly clean automation of the grading process. By running all test inputs through our reference implementation, we generate reference XML output files. Students use a script to turn in their source code electronically, and we use our own script to collect the turn-in files, compile them, and run them. We then use *xmldiff* [4], a freely available *diff*-like tool for comparing XML files, to compare student output to our reference output. We take an all-or-nothing approach to assigning points. For a 100-point assignment, we test each team's code on 100 different input files designed to cover all language features. The team earns one point for each output file which matches the corresponding reference output according to xmldiff. Generally, we do not award partial credit for partial matches, as this would require too much time for such a large class. This means that even minor spacing errors can cost a team significant points. To make this process more fair, we do provide a small set of public test inputs and outputs to the students while they work on a particular phase, and they have access to the xmldiff tool. This gives them the opportunity to check their output against the posted output, to help eliminate these types of errors. Grades are returned to students via email, along with the xmldiff output from the test runs and some individual comments on their progress, so that they can determine which types of tests they did not pass. Because the next phase of the interpreter builds on the current phase, timely feedback about mistakes is important for student progress.

In addition to the programming projects, we give midterm and final examinations to test the students' knowledge of course concepts. Students take these exams individually. Inevitably, there are some project teams in which one student performs a disproportionate amount of the work, and these individual assessments help us to assign appropriate grades. To this end, it is important to test understanding of the XQuery language in addition to compiler theory on the exams. This allows us to identify students who did not learn XQuery well because they did not put in their share of effort on the projects.

4. OUR VARIANTS OF XQUERY

In this section, we describe a language based on XQuery which we believe is interesting from a pedagogical standpoint. First, however, we begin with a very elementary overview of some of the features of full XQuery.

4.1 Some Basic Features of XQuery

XQuery is a functional language. The basic building blocks in XQuery are expressions, whose values consist of sequences of items. Designed as a query language for the widely-used markup language XML, XQuery allows concise yet powerful queries over XML documents. As such, built-in types in XQuery are based on the XML type system. A large number of standard functions are specified in XQuery, for tasks such as arithmetic, logical and set operations. Both user-defined types and functions are also allowed.

A central construct in the language is the FLWOR expression, where the letters in the acronym stand for the clauses

available in the expression, namely, *for*, *let*, *where*, *order by* and *return*. This type of expression allows iteration over and binding of values to variables, along with selection of particular values. Specifically, *for* clauses permit iteration, *let* clauses permit binding, and an optional *where* clause "gates" the execution of the *return* clause. That is, for a particular iteration, the boolean expression associated with the *where* clause is evaluated. If the expression is true, the *return* clause is executed for that iteration. The value of a FLWOR expression is the concatenation of all values returned by each execution of the *return* clause of the FLWOR into a single sequence. The optional *order by* clause may be used to sort the items in the returned sequence.

FLWOR expressions are useful for computing joins between two or more documents and for restructuring data.

Data in a well-formed XML document is logically arranged in a tree specified by the Document Object Model (DOM), and different nodes in the tree are of different types (e.g., *element*, *attribute*, *text*, *comment*, etc.). XQuery's path expression construct allows the selection of nodes in an XML document based on their position in the document's DOM tree, their types, and the names given to the nodes. For example, using a path expression within a FLWOR, one can iterate over a sequence of nodes in a particular document, such that, at every node n in the sequence, all logical children of n which are *element* type nodes that have name foo are selected. The full syntax of such an expression is as follows, where we assume that the variable `$sequence` is already bound to a sequence of XML nodes:

```
for $n in $sequence return $n/child::element(foo)
```

Because the path expression appearing in the above return clause is a common type of query, XQuery permits a shorthand version of it. Thus, the following expression is equivalent to the one above:

```
for $n in $sequence return $n/foo
```

That is, expressions of the form *a/b* indicate that all *element* children of node *a* with name *b* are to be returned.

A related path expression allows queries for all element nodes which are descendants (on any level in the tree) of a node. There are many other sorts of path expressions, allowing queries based on different position specifiers within the DOM tree and different node types. Some of these path expressions can be expressed using full path expression syntax as well as specialized shorthand notation.

XML documents on which path expressions queries are made can be read in from external files, or, alternatively, XML nodes may be constructed within the XQuery program themselves. *Direct constructors* allow literal XML to be written verbatim within an XQuery program. *Computed constructors* allow XML nodes to be specified using keywords to identify the type of node to be created.

4.2 Our Variations on XQuery

Because XQuery has many additional features (which we lack sufficient space to mention here), students in a quarter-long course can implement only a subset of the actual XQuery standard language. Here we discuss several variants of XQuery that include language features which we found to be interesting in our compiler construction course. In some cases, our variants are not strictly subsets of the official XQuery language; we do deviate from the official specification somewhat. However, we believe that the flavor of the official language is retained.

First, we restrict the type system of the language, so that only approximately 15 types are included. (Note that nine of these are XML node types or groups of of XML nodes, and much of the functionality required to implement them is available in standard Java classes.) We do not allow user-defined types. We also restrict the number of built-in functions, the different types of path expressions and the types of constructors allowed. We give detailed instructions about implicit type conversions used in function calls.

We are experimenting with the idea of treating path expressions as built-in functions, rather than using their official syntax. We also intend to eliminate the shorthand versions of path expressions, which students found confusing and which do not provide significant pedagogical interest.

We also found it useful to rework the XQuery grammar so that, for example, literals are derived in a small number of steps. This allows testing of parser code to be performed incrementally, instead of requiring that students have a large section of the grammar in place before testing can begin.

To combat plagiarism, we modify our project requirements from quarter to quarter in an attempt to avoid the "code migration" problem, where students in the course during a previous quarter share their code with students currently enrolled in the course. Of course, these change in requirements necessitate the preparation of a new reference implementation each quarter, but we feel the gains are worth the additional instructor effort.[2]

Changing the AST specification from quarter to quarter is one way to drastically change the code required for the project. For example, we changed the AST representation of a FLWOR expression dramatically one quarter. This not only helped combat plagiarism, it was also a representation that simplified the interpretation of the expression. In other quarters, we removed the *order by* clause from the language completely, allowing only FLWR expressions instead.

More simple changes include using different subsets of official XQuery's many built-in functions, or specifying new built-in functions which allow any number of parameters. Modifications to the error handling requirements are another option. Furthermore, we have incorporated different language features from the official XQuery specification at different times, such as the explicit specification of parameter and return types in user-defined functions and quantified expressions.

5. DISCUSSION

In the past two years, we have offered this course five different times. We now mention some general challenges that we faced along with the XQuery-specific issues we mentioned earlier. We close with some summary remarks.

5.1 Challenges

One of the goals of our compilers course is to expose students to large-scale programming projects. Since this is a new experience for a majority of the enrolled students, they have initial difficulties in grasping the big picture of the assignment and budgeting their time.[3] (These problems are typical of capstone courses, regardless of topic.) We address

[2]Because we use a different language each quarter, students sometimes find bugs in the project specifications. For some, this may be a good learning experience, but other students understandably find this frustrating.

[3]As last-minute desperation engenders plagiarism, we also

these challenges as follows. In order to encourage students to begin working on the phases of the project as soon as they are assigned, we have instituted voluntary checkpoints midway through the longer phases. In the future, these may include students turning in part of the assignment for feedback. This checkpoint would not be counted in their grade for the course, but might help them determine what areas need attention before the final project is handed in. Although it increases our administrative burden, we hope that this potential boost to their final score will motivate students to work steadily on the project from the beginning, increasing their chances for success. Adding software engineering topics [12] to the course may also be helpful, though there is not much time to spare during a ten-week quarter. On a departmental level, we would like to increase the effectiveness of prerequisite courses at preparing students to handle assignments of this size, so the long-term nature of the interpreter project does not come as a shock.

Because students work in pairs on the projects, we face the usual team-related issues. For example, is each partner is pulling his or her own weight? As mentioned above, we try to address this problem by including project-related material on the individual exams. When assigning final letter grades, students who have high project scores but low exam grades raise a red flag. Another minor problem arises when one student from a team withdraws during the quarter. We try to assist the remaining partner by finding another individual in the same situation, and in our large classes, we are usually successful. Another team-related challenge is that students become accustomed to asking a peer for assistance. We wonder if working in teams increases the likelihood of different pairs working together, which is not allowed.

Finally, we find that some topics covered in lectures and exams seem disconnected from the interpreter project. For example, the textbook and lectures include descriptions of top-down and bottom-up parsing algorithms, but students do not feel this knowledge is helpful, since CUP automates the parser generation process. Throughout the quarter, student interest level during lecture seems fairly low except when we are discussing project details. Students have repeatedly expressed their feelings that the book is "useless until it is time to study for the exams".

5.2 Conclusions

We conclude with some observations about using XQuery as the focus language in our compiler construction course. First, we note that none of the major challenges we faced were introduced when we began using XQuery. These challenges could arise during any team-based large project.

Compared with prior offerings of the course, we now spend more time discussing the base language and XML, as well as interpreter design. These come at the expense of some parsing theory, and add to the list of new ideas that students must juggle. An argument can be made that reducing the time spent on parsing theory is warranted, and we feel that the exposure to languages and tools which are more "real-world relevant" is useful for the students.

Looking toward the future, we can imagine several other versions of this course which use XQuery as a focus language. The XML-based modular design of the system allows students to concentrate on any particular part of the compiler. For example, we could provide students who write a lexer and parser with a web interface to an existing semantic engine, so that they could more easily verify the correctness of the AST output by their parser. (A similar approach where a parser is provided would allow students to verify the correctness of the token stream output by their lexer.) Alternatively, we may want to design a course emphasizing software design that would concentrate more on database issues, and less on lexing and parsing. To acheive this, we could provide a web interface to a parser that would generate an appropriate AST. Interpreters written by students would then take the generated AST as input, freeing them from the burden of writing the lexer and parser. Furthermore, the Java DOM implementation could be used as the AST representation, providing a simple interface. The convenience of the XML-based architecture of an XQuery interpreter makes other variations possible as well.

run student submissions through MOSS[5, 13], a tool which checks software similarity. MOSS is most useful on the semantic engine portion of the assignment, because it does not work on the JFlex and CUP files which constitute the majority of the earlier assigments.

6. REFERENCES

[1] CUP Parser Generator For Java. `http://www.cs.princeton.edu/appel/modern/java/CUP/`, August 2004.

[2] Document Object Model (DOM). `http://www.w3.org/DOM/`, August 2004.

[3] JFlex - The Fast Scanner Generator For Java. `http://jflex.de/`, August 2004.

[4] Logilab's xmldiff. `http://www.logilab.org/projects/xmldiff/`, August 2004.

[5] MOSS: A System for Detecting Software Plagiarism. `http://www.cs.berkeley.edu/aiken/moss.html`, August 2004.

[6] W3C World Wide Web Consortium. `http://www.w3.org/`, August 2004.

[7] XML Path Language (XPath) Version 1.0. `http://www.w3.org/TR/xpath`, August 2004.

[8] XML Syntax for XQuery 1.0 (XQueryX). `http://www.w3.org/TR/2001/WD-xqueryx-20010607`, August 2004.

[9] XQuery 1.0: An XML Query Language. `http://www.w3.org/TR/xquery/`, August 2004.

[10] D. Baldwin. A compiler for teaching about compilers. In *Proceedings of the 34th SIGCSE Technical Symposium on Computer Science Education*, pages 220–223. ACM SIGCSE, February 2003.

[11] S. Debray. Making compiler design relevant for students who will (most likely) never design a compiler. In *Proceedings of the 33rd SIGCSE Technical Symposium on Computer Science Education*, pages 341–345. ACM SIGCSE, February 2002.

[12] W. G. Griswold. Teaching software engineering in a compiler project course. *ACM Journal of Educational Resources in Computing*, 2(4):1–18, December 2002.

[13] S. Schleimer, D. S. Wilkerson, and A. Aiken. Winnowing: Local algorithms for document fingerprinting. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, pages 76–85. ACM SIGMOD, June 2003.