APPLYING OBJECT-ORIENTATION AND ASPECT-ORIENTATION IN TEACHING DOMAIN-SPECIFIC LANGUAGE IMPLEMENTATION*

Xiaoqing Wu, Barrett Bryant and Jeff Gray
Department of Computer and Information Sciences
The University of Alabama at Birmingham
Birmingham, AL, USA 35294-1170
Phone: 1-205-934-2213
{wuxi, bryant, gray}@cis.uab.edu

Marjan Mernik
Faculty of Electrical Engineering and Computer Science
University of Maribor
2000 Maribor, Slovenia
Phone: 386-2-220-7455
marjan.mernik@uni-mb.si

ABSTRACT

In traditional compiler design and programming language courses, the complexity required for a successful implementation of the course project is often a major obstacle for many students. This is especially true for courses focused on the design and implementation of domain-specific languages, where the language evolves constantly. This paper describes an approach that allows students to modularize the language constructs of a compiler using object-orientation (OO) and aspect-orientation (AO). Compared to traditional methods used in compiler projects, such a modular design can help students to improve the comprehensibility and changeability of their implementation, leading to a decrease in the overall complexity.

INTRODUCTION

Project development is one of the greatest challenges in teaching compiler design and programming language implementation courses. Many students often have difficulty in completing a real compiler project from the beginning to the end. This is due to the fact that the implementation is hard to modularize, specifically because the different phases of compiler construction (e.g., lexical analysis, syntax analysis, tree generation, static checking, and code generation) are always tangled together, making the implementation complex to evolve. The problem is more evident when the target language is continually evolving, as in the case of Domain-Specific Languages (DSLs) [1], which change more frequently than general-purpose programming languages. The problem cannot be solved solely by formal specification-based implementations (as introduced in classical compiler design textbooks [2]) because most mathematics-based formal specifications do not cleanly separate the different phases of the compiler implementation and do not provide a strong library mechanism and I/O capabilities. Consequently, it can be very complicated to implement low-level semantics because of the difficulty in comprehending the details and associations among language constructs. Despite the potential applicability of various formal specifications, students generally implement their project by a manual implementation or through use of a parser generator such as YACC (Yet Another Compiler-Compiler - http://dinosaur.compilertools.net/yacc).

To address the problem of complexity in compiler projects, students should be encouraged to apply modern software engineering principles and concepts in their implementation. This paper describes an approach that assists a student in constructing a compiler with greater extensibility and changeability. A side benefit of the approach is that students have a context for mastering state-of-the-art software engineering methodologies.

The paper is organized as follows. The next section introduces DSLs and aspect-oriented programming, followed by a sample DSL that will be used as a case study. The case study is used to illustrate the application of object-orientation and aspect-orientation in compiler implementation. The pedagogical benefits in adopting the approach are summarized in the final section.

BACKGROUND: DOMAIN-SPECIFIC LANGUAGES AND ASPECTS

To understand the remainder of the paper, there are two research areas that are presented briefly in this section.

Domain-Specific Languages (DSLs). A DSL is a computer language targeted to a particular kind of problem, rather than a general purpose language aimed at any kind of software problem. DSLs usually allow solutions to be expressed at the level of abstraction of the problem domain such that low-level implementation details are hidden and implemented by the compiler. A typical example of a DSL is SQL, which enables database users to manipulate data without concern for data storage issues.

Aspect-Oriented Programming (AOP). Aspect-Oriented Programming [3] provides special language constructs called aspects that modularize crosscutting concerns in conventional program structures (e.g., a concern that is spread across class hierarchies of object-oriented programs). In AOP, a translator called a weaver is responsible for

merging the additional code specified in an aspect language into the traditional language. A general aspect-oriented language for supporting separation of crosscutting concerns is AspectJ [4], which is an extension of Java.

SAMPLE LANGUAGE DESIGN: GQL

Although Google™ has already provided dozens of query forms to interface its advanced search features, the inflexible and untraceable nature of these forms offsets the popularity of using Google advanced search. The Google Query Language (GQL) is a DSL that we have developed to provide a user-friendly facility to support advanced Google search. The language enables query results to be constrained by domain names, language preference, file format, file date, and keyword location. Moreover, in addition to traditional text search, it also supports image search, online groups search, news search and shopping search. A key feature of GQL is automatic syntax and static checking to insure the user-supplied query is valid. Additionally, each GQL query is represented by a named program, which makes it possible to search within previous query results (i.e., previous queries can be reused to build new queries and consequently make the new query more precise). The left part of Figure 1 is a sample GQL program called *CSResume*, which is used to search sample resumes that contain the exact keyword "computer science" in the form of a PDF or PS file. As can be seen, *CSResume* is based on an existing GQL program named *SampleResume* (shown in the right of Figure 1).

```
CSResume
search "computer science"
from SampleResume
where type=pdf, ps
```

```
sampleResume
search CV|resume,
sample|example, !letters
```

Figure 1. Sample GQL programs

Figure 2 shows the initial syntax for GQL. The students are asked to extend the grammar several times during language evolution. This initial version will be used as an example to illustrate the benefits ascribed in the next section.

```
query ::= searchtype keylist fromstmt
      constraints
searchtype ::= WEBSEARCH | IMGSEARCH
keylist ::= key | keylist COMMA key
key ::= word | noword | orwordlist | exactword
word ::= STRING
noword ::= NOT word
orwordlist ::= orword OR orword |
        orwordlist OR orword
```

```
orword ::= word | exactword
exactword ::= QSTRING
constraints ::= WHERE constraintlist |
constraintlist ::= filetype|constraintlist filetype
filetype ::= acceptfiletype | rejectfiletype
acceptfiletype ::= TYPE EQ TYPEVALUE
rejectfiletype ::= TYPE NE TYPEVALUE
fromstmt ::= FROM query | FROM filename |
filename ::= STRING
```

Figure 2. GQL syntax

METHODOLOGY

The goal of the GQL translator is to compile the GQL programs into Google recognizable search tokens, which are a sequence of user-supplied search keywords associated with reserved keywords (e.g., *filetype:*). in addition to the parsing phase, two more phases need to be built after the abstract syntax tree (AST) is generated. The static

checking phase is utilized to ensure the program represents a valid query and the code generation phase implements the translation.

Figure 3 provides the control flow of DSL implementation. Tools are shown in ellipses. Shaded boxes contain generated code. To describe the language GQL, the student needs to first specify the lexical and syntactic rules for each grammar symbol (terminal or non-terminal) of GQL in a Context-Free Grammar (CFG). The CFG will serve as input to a specification compiler that extracts lexical rules and syntax rules, which can be processed by the lexer generator JLex (Java Lexical Analyzer Generator - http://www.cs.princeton.edu/~appel/modern/java/JLex/) and parser generator CUP (Parser Generator for Java - http://www.cs.princeton.edu/~appel/modern/java/CUP/) to generate the corresponding lexer and parser in Java. Additionally, AST nodes are generated as Java classes and interfaces. The parser uses the embedded action code in CUP to call the construction methods of these AST classes to build an AST object structure during the parsing phase. Students can later add AspectJ code for semantic analysis without any change to the automatically generated Java code. After the lexer, parser and AST nodes in Java are compiled together and the semantics in AspectJ are weaved into those Java classes, a GQL compiler is produced. The use of object-orientation (OO) and aspect-orientation (AO) can improve the ability to evolve GQL, as described below.

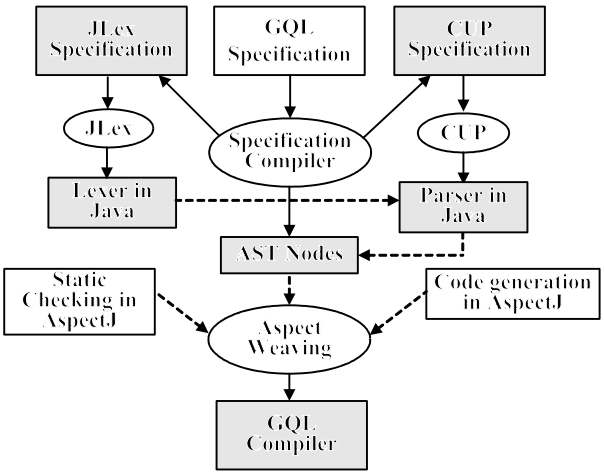

Figure 3. The compiler implementation framework

A benefit can be realized by applying the Interpreter pattern [5] to treat each GQL grammar symbol as a class (terminals are treated as a special class *String*) in order to generate Java code for AST nodes. For each production rule in the form of $R ::= R_1 R_2 ... R_n$, the left-hand-side (LHS) non-terminal R is generated as a Java class, and the right-hand-side (RHS) of the production $R_1 R_2 ... R_n$ is generated as a set of attributes of the class, which are assigned values when the constructor is called. For example, the CFG production in GQL "*query ::= searchtype keylist fromstmt constraints*" will generate the following Java class for non-terminal *query*:

```
class Query implements Node{
    public String searchtype, Keylist keylist, Fromstmt fromstmt, Constraints constraints;
    public Query (String searchtype, Keylist keylist , Fromstmt fromstmt, Constraints constraints ){
```

```
            this.searchtype = searchtype;        this.keylist = keylist;
            this.fromstmt = fromstmt; this.constraints = constraints;
        }
}
```

By using the Interpreter pattern, students can easily change and extend the grammar during compiler development. This is well-suited for GQL implementation because the language has to be extended as more Google functionalities are increasingly supported. The student can modify the grammar by class manipulation or extend the grammar using inheritance. Moreover, because each AST node represented by a Java class is automatically generated from the syntax grammar, the students do not need to have a separate phase to design AST nodes and generate the tree.

Each phase of semantics analysis has basic AOP characteristics: the structure and behavior characteristics are scattered throughout the AST nodes. In order to freely attach each phase of the semantics analysis to generated AST nodes, the aspect-oriented Visitor pattern [5, 6] can be utilized in the student compiler implementation. In the Visitor pattern, all the methods pertaining to one operation of the nodes are encapsulated into a single visitor aspect, which is independent of other node classes and can be freely added or deleted from the implementation. Below are the sample aspect specifications for static check of AST node *Acceptfiletype*. More details of the aspect-oriented semantics implementation can be found in [6].

```
aspect StaticCheck{
  ... ...// static check for other AST nodes
  public boolean Acceptfiletype.staticCheck(){
  if (typevalue.compareTo("pdf")==0 || typevalue.compareTo( "ps" )==0 ||
          typevalue.compareTo( "doc" )==0 || typevalue.compareTo( "xls" )==0 ||
          typevalue.compareTo( "ppt" )==0 || typevalue.compareTo( "rtf")==0)
     return true;
  else {
       ErrorReport.ErrorMessage("filetype must be pdf, ps, doc, xls, ppt, or rtf");
     return false; }
  }
}
```

By using an aspect-oriented semantics implementation, all the operations that belong to one phase can be encapsulated as a separated aspect, which allows additions to be added to the existing class hierarchy without "polluting" the parser or AST node structure. Therefore, students can always come back to the early phase during development of later phases. This feature eases the progress of development as well as facilitates teaching and grading of compiler projects, because the failure of one phase will not affect the success of previous phases. Each aspect can either run independently or glued together with other aspects as one phase by using the pointcut-advice [4] model. In AspectJ, a pointcut is used to provide an abstraction for one or more phases with before advice associating the abstraction with other phases. For example, the aspect code below invokes static checking of each AST node before compiling the language into Google recognizable tokens.

```
pointcut translate(Node node): target(node) && call(String *.translate());
before(Node node): translate(node){       node.staticCheck ();        }
```

Another benefit of using aspects in semantics implementation is the ability to accumulate states (e.g., the symbol table) during AST node traversal, which comes from the use of

the Visitor pattern. Without a visitor, the state would be passed as extra arguments to the semantics operations or they might appear as global variables.

RESULTS & CONCLUSIONS

This paper describes a pedagogical approach that allows students to construct a DSL compiler project rapidly using a modular technique. The concept is based on the application of object-orientation and aspect-orientation to compiler design. Compared to the traditional methods used in compiler course projects, students using OO and AO can improve the comprehensibility and changeability of their compiler project, which leads to a decrease in the overall complexity of their implementation. When the language definition is frequently changed (e.g., evolution of a domain-specific language), the benefits are even more evident.

A language called GQL is presented in the paper as an example target language for a compiler project. In the implementation of the GQL compiler, the lexical and syntax analysis are integrated as one phase, which automatically generates lexer, parser and AST classes for the language. This releases the students from spending particular effort on the phase of tree generation. The framework enables a student to build the GQL compiler through a series of phases with clear separation of each phase in the implementation. The separation of concerns among compiler phases eases the development task for students. There is initial evidence to support a claim that teaching compiler and language concepts is improved, as well as a reduction in the grading effort. The implementation strategy helps students understand and master state-of-the-art software engineering concepts in the context of a compiler course.

REFERENCE

[1] van Deursen, A., Klint, P., Visser, J., Domain-specific languages: an annotated bibliography, *ACM Sigplan Notices*, 35 (6), 26-36, 2000.

[2] Aho, A. V., Sethi, R., Ullman, J. D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C., Loingtier, J., Irwin, J., Aspect-oriented programming, *Proceedings of the 11th European Conf. Object-Oriented Programming (ECOOP)*, LNCS 1241, 220-242, 1997.

[4] Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W. G., An Overview of AspectJ. *Proceedings of the 15th European Conf. on Object-Oriented Programming (ECOOP)*, LNCS 2072, 327–355, 2001.

[5] Gamma, E., Helm, R., Johnson, Vlissides, R., J., *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[6] Wu, X., Roychoudhury, S., Bryant, B., Gray, J., Mernik, M., A two-dimensional separation of concerns for compiler construction, *Proceedings of the ACM Symposium on Applied Computing (SAC)*, 1365-1369, 2005.