Parser Development with an Internal Domain-Specific Language and an Aspect Weaver

Kazuaki Maeda
Department of Business Administration and
Information Science, Chubu University
1200 Matsumoto, Kasugai, Aichi 487-8501, Japan
kmaeda@gmail.com

ABSTRACT

This paper describes an internal domain-specific language (DSL) to write syntax rules and action code in Ruby. Parser generators read a mixture of the syntax rules and the action code written in one of the general-purpose languages. If the action code contains trivial syntax errors, the parser generators do not analyze the code; they generate source code containing the syntax errors. To resolve this situation, an internal DSL to define syntax rules was designed as a subset of Ruby. Moreover, action code is injected into the generated parser using an aspect weaver. In the author's preliminary experience, productivity was improved in the design and implementation of parsers.

Categories and Subject Descriptors

D.3.4 [**Processors**]: Translator writing systems and compiler generators; D.2.11 [**Software Architectures**]: Domain-specific architectures; D.3.3 [**Language Constructs and Features**]: Modules, packages

General Terms

Parser Development, Domain-Specific Language, Aspect-Oriented Programming, Ruby

1. INTRODUCTION

Parser development was complex before the 1970s. In the 1970s, the parser generator Yacc[3] was built, which made parser development much easier. Yacc reads user-defined syntax rules with action code and generates a parser written in the C programming language. If each input string is matched to a syntax rule, then the action code associated with the syntax rule is invoked. After Yacc was introduced to parser development, many parser generators have been built[4, 5, 6, 10, 11].

Parser generators provide their custom languages to define syntax rules. The custom language is called a DSL, which stands for Domain-Specific Language.

A DSL is a programming language tailored to a specific application domain[1] and designed precisely to describe problems in the specific domain[2]. It is a special-purpose, and not a general-purpose, programming language. We write the syntax rules in a context-free grammar style using a DSL. The parser generators read a mixture of the syntax rules written in the DSL and action code written in one of the general-purpose programming languages, such as C or Java.

Martin Fowler's book[12] describes two types of DSLs: an external DSL and an internal DSL. In the book, they are defined as follows:

- *An external DSL is a DSL represented in a separate language to the main programming language it's working with.*
- *An internal DSL is a DSL represented within the syntax of a general-purpose language.*

All custom languages used to represent syntax rules are different from general-purpose languages; therefore they are the external DSLs.

To develop a parser using Yacc, one must write the action code in C. Yacc reads the syntax rules with the action code and generates source code in C. It is a useful tool to develop parsers in C, but Yacc does not analyze the action code associated with the syntax rules. If the action code in C contains trivial syntax errors, such as omitting semicolons, Yacc generates source code that still contains the syntax errors. The C compiler finds the syntax errors after the source code was generated by Yacc, but the C compiler cannot determine which syntax rule contains action code with syntax errors.

Using an internal DSL is one of the approaches to resolve this situation. The internal DSL is a language represented with a subset of a general purpose language. If the syntax rules are defined with action code in the same programming language, syntax errors can then be detected by language processors.

To develop parsers to analyze source code in modern programming languages (e.g., C# or Java), many syntax rules must be written. For example, when writing syntax rules for C#, the author's experience shows that, according to the specification written in the C# book[13], more than 700 syntax rules for C# had to be defined. A similar situation exists for Java; according to the Java specification[14], more than 500 syntax rules for Java had to be defined.

BICT 2014, December 01-03, Boston, United States

DOI 10.4108/icst.bict.2014.257977

Typically for parser generators, action code is tightly coupled to the syntax rules, and the code is scattered across the syntax rules. It is hard to understand and maintain such a large number of syntax rules with action code.

One of the solutions to decouple the syntax rules and the action code is to use the Visitor design pattern[15]. Both the syntax rules and the related action code can exist separately using the Visitor pattern. For example, ANTLR Version 4[7] supports the generation of some classes for the Visitor pattern. ANTLR is a parser generator for processing or translating source code. The Web site[8] provides more than forty grammars for various languages — from simple languages like JSON to complex programming languages like Java 8. From the grammar, ANTLR generates a parser that can build and walk parse trees. The generated code walks parse trees for invoking some methods defined in the Visitor class.

Aspect-oriented programming (AOP) is a technique to modularize cross-cutting concerns[9]. Cross-cutting concerns are pieces of functionality that are used across a number of classes. If the pieces of code are scattered over a system, it becomes increasingly difficult to develop and modify them. In AOP, the code fragments can be separated with concerns (called aspects) for modularization. An aspect weaver is used to combine the code fragments for the aspects into a coherent program. The technological improvements realized when using AOP should be reflected in parser development. The AOP approach should be applied to separate the syntax rules and the action code.

This paper describes an internal DSL used to write syntax rules based on Ruby. Ruby is an object-oriented programming languages. It plays an important role as a host language for the internal DSL. One of Ruby's appealing features is the fact that using parentheses for arguments of methods are optional; therefore, statements are easier to read and understand than the ones in other programming languages.

Section 2 explains the motivation for parser development. Section 3 explains the syntax rules in the internal DSL and the injection of action code. Section 4 summarizes this paper.

2. PARSER DEVELOPMENT USING PARSER GENERATOR

2.1 DSL for Parser Development

All parser generators provide their custom languages that are used to define syntax rules. The syntax rules are written in different syntax from any general-purpose languages. Therefore, they are external DSLs. The parser generators read a mixture of the syntax rules written in the external DSL and action code written in a general-purpose language; they then generate source code in the target programming language.

Figure 1 shows a snippet of the syntax rules for Yacc, and the action code in C for a simple arithmetic expression. The generated parser reads the simple arithmetic expression, such as $1 + 2 * 3$, and it builds a lisp-like prefix expression, $(+\ 1\ (*\ 2\ \ 3)\ )$.

```
expr : expr PLUS term
     { $$ = newString("(+ %s %s)",$1,$3);}
     | term
     { $$ = $1; }
     ;
term : term MULT NUMBER
     { $$ = newString("(* %s %s)",$1,$3);}
     | NUMBER
     { $$ = $1; }
     ;
```

Figure 1: Syntax rules of an arithmetic expression with action code

```
expr : expr PLUS term
     { $$ = newString("(+ %s %s)",$1,$3) }
     | term
     { $$ = $1; }
     ;
term : term MULT NUMBER
     { $$ = newString("(* %s %s)",$1,$3);}
     | NUMBER
     { $$ = $1; }
     ;

% yacc -yvd p.y
% cc -c -o y.tab.o y.tab.c
p.y:34:95: error: expected ';' after expression
  ...%s %s)",(yyvsp[(1) - (3)].value),
                 (yyvsp[(3) - (3)].value)) }
                                           ^
                                           ;
1 error generated.
```

Figure 2: Syntax rules of an arithmetic expression containing a syntax error in the action code

When the generated parser reads input characters and a syntax rule is matched to the input, the action code associated with the rule is invoked. For example, if Yacc reads the syntax rules shown in Figure 1 and the generated parser reads $1 + 2$ as an input, the syntax rule

```
expr : expr PLUS term
```

is matched, and the action code

```
$$ = newString("(+ %s %s)",$1,$3);
```

is invoked. The action code is surrounded by curly braces. $$ is a symbol that represents an attribute of the left-hand side of each rule. $1 represents an attribute of the first symbol `expr` of the right-hand side, and $3 represents an attribute of the third symbol `term`.

Yacc does not find syntax errors in action code. For example, the action code written in C might contains a trivial syntax error, such as omitting a semicolon, as shown in Figure 2. Yacc reads a file *p.y*, but it does not analyze the action code. Yacc embeds the fragments of action code in a source file *y.tab.c*, as shown in Figure 3, and successfully generates the source code containing the syntax error in C. After the code generation, the C compiler finds the syntax errors at line 34, but the C compiler does not know which syntax rule contains the action code with the error. In this case, developers need to read the generated file *y.tab.c* and guess from the error messages to fix the syntax error.

Using an internal DSL to define syntax rules is one of the approaches to resolve these situations. If syntax rules are written in the internal DSL based on Ruby and the action

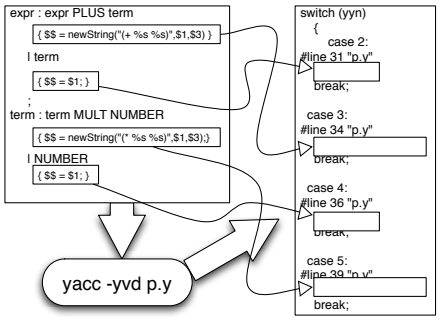

Figure 3: Yacc embeds fragments of code in y.tab.c

code is also written in Ruby, syntax errors are detected by the Ruby interpreter before it generates source code.

2.2 Decoupling Action Code from Syntax Rules

Building a program to parse source code requires a parser to invoke specific action code. The action code is associated with each syntax rule. Typical parser generators read the syntax rules with the action code, and generate parsers written in a target programming language. The action code is tightly coupled to the generated parser.

As previously mentioned, one of the solutions to decouple action code from the syntax rules is to use the Visitor pattern. Decoupling the syntax rules from the action code makes it reusable for different programs. ANTLR supports the generation of some classes for the Visitor pattern. We can implement some concrete visitor classes and the visitors walk parse trees by calling some methods defined in the Visitor classes.

However, there is no way to add code fragments in the generated parsers. ANTLR generates some classes to handle parse trees shown in Figure 4, such as ExprContext, AddOrSubContext, MulOrDivContext and NumberContext, for the example shown in Figure 1. If some code fragments need to be added in the generated classes, the only option is to modify the generated code by hand. It would be troublesome to maintain the parsers. In this case, all functionality must be written in the Visitor. If a lot of extra code is written, it makes implementation complicated and difficult to understand and maintain.

AOP made an advanced technological improvement in software engineering in twenty years. It is a technique used to separate concerns. The technological improvement can be reflected in parser development. The author believes that the separation of concerns improves modularity of the syntax rules and action code in parser development.

3. INTERNAL DSL AND ASPECT WEAVER

Based on the ideas mentioned in the previous section, the author designed an internal DSL to define syntax rules based on a subset of Ruby. The syntax rules are written in context-free grammar. Figure 5 shows an example of syntax rules for the simple arithmetic expression, which are the same syntax rules shown in Figure 1.

A symbol in Ruby means a terminal or a non-terminal in the

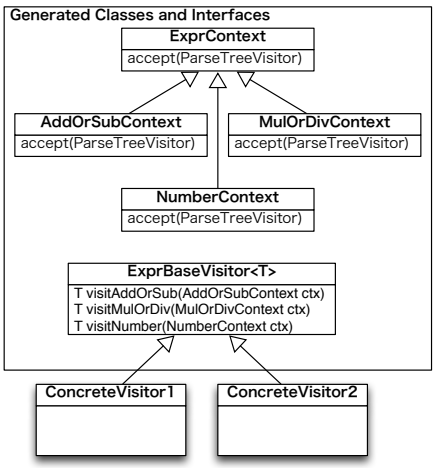

Figure 4: Generated classes and interfaces by ANTLR

```
:expr .> :term
:expr .> :expr, :PLUS, :term
:term .> :term,:MULT,:NUMBER
:term .> :NUMBER

:PLUS.is_token
:MULT.is_token
:NUMBER.is_token
```

Figure 5: Syntax rules and token definitions for a simple arithmetic expression in the internal DSL

grammar. The operator .> means derivation. For example, a syntax rule

```
:expr .> :expr, :PLUS, :term
```

means that `:expr`, `:PLUS`, and `:term` are derived from `:expr`. The method `is_token` defines a token symbol. In the figure, three token symbols `:PLUS`, `:MULT`, and `:NUMBER` are specified.

A generator, called RisoLR, was developed to generate a bottom-up parser. RisoLR reads user-defined syntax rules, as shown in Figure 5, and generates a bottom-up parser with some classes and methods written in Ruby. The generated parser is composed of a driver for parsing in Ruby, and parsing data.

RisoLR generates simple method definitions for the action code shown in Figure 6. For example, it reads a syntax rule

```
def doExpr_Term(p0,p1) p0 end
def doExpr_ExprPlusTerm(p0,p1,p2,p3) p0 end
def doTerm_TermMultNumber(p0,p1,p2,p3) p0 end
def doTerm_Number(p0,p1) p0 end

def inExpr_Term(&block) ... end
def inExpr_ExprPlusTerm(&block) ... end
def inTerm_TermMultNumber(&block) ... end
def inTerm_Number(&block) ... end
```

Figure 6: Generated stub for action code and methods to inject code

```
injection = Injection.new
injection.inExpr_ExprPlusTerm {
    |jp,obj,a0,a1,a2,a3|
    a0 = "(+ #{a1} #{a3})"
}
injection.inExpr_Term {
    |jp,obj,a0,a1|
    a0 = a1
}
```

Figure 7: Injected action code for a simple arithmetic expression

```
:expr.> :expr, :PLUS, :term
```

and generates a method definition

```
def doExpr_ExprPlusTerm(p0,p1,p2,p3) p0 end
```

The method name is made by concatenating the prefix `do` and four symbols `:expr`, `:expr`, `:PLUS`, and `:term`. The method has four arguments and returns only the value of p0. The arguments `p0`, `p1`, `p2`, and `p3` are attributes in correspondence of four symbols.

RisoLR also generates method definitions to inject action code. As shown in Figure 7, if the method call

```
injection.inExpr_ExprPlusTerm {
 |jp,obj,a0,a1,a2,a3|
  a0 = "(+ #{a1} #{a3})"
}
```

is invoked, the block

```
                                {
  |jp,obj,a0,a1,a2,a3|
  a0 = "(+ #{a1} #{a3})"
}
```

is injected into the body of the method `doExpr_ExprPlusTerm`. The argument `jp` is for a join point, `obj` is for a targeted object. The four arguments `a0`, `a1`, `a2` and `a3` are attributes of four symbols. Therefore, the method `doExpr_ExprPlusTerm` is modified and semantically same as

```
def doExpr_ExprPlusTerm(p0,p1,p2,p3)
  p0 = "(+ #{p1} #{p3})"
  p0
end
```

The injection is implemented by Aquarium[16] as the following:

```
def inExpr_ExprPlusTerm(&block)
  Aspect.new(:around,
             :calls_to=>:doExpr_ExprPlusTerm,
             :on_types=>[GeneratedCode], &block)
end
```

Aquarium was developed by Dean Wampler for an AOP library in Ruby. We can define join points and advices using Aquarium.

4. SUMMARY

This paper described an internal DSL to write syntax rules based on Ruby, and injection of action code into the generated parser using Aquarium. Traditional parser generators provide their custom languages to define syntax rules. The parser generators read a mixture of the syntax rules and action code written in a general-purpose languages. If the action code contains trivial syntax errors, the parser generators do not analyze the code and they generate source code containing the errors. To resolve this situation, the internal DSL was designed as a subset of Ruby. If we define syntax rules in the internal DSL and action code in Ruby, syntax errors are detected by the Ruby interpreter before generating the source code. A generator, called RisoLR, was developed to generate a bottom-up parser. The internal DSL gives a terse description to define the syntax rules and the action code can be injected into the generated parser. Currently, RisoLR and its related tools are still under development.

5. REFERENCES

[1] Marjan Mernik, Jan Heering, and Anthony M. Sloane. When and How to Develop Domain-Specific Languages, ACM Computing Surveys, Vol.37, No.4, pp.316–344., 2005.

[2] Paul Hudak. Building Domain-Specific Embedded Languages, *ACM Computing Surveys*, Vol.28, No.4es, p.196, 1996.

[3] Steven Johnson. Yacc: Yet Another Compiler Compiler, UNIX Programmer's Manual, Vol.2, pp.353–387, 1979.

[4] Bison – Gnu Parser Generator, http://www.gnu.org/s/bison/ (accessed on Sep. 20, 2014).

[5] E. M. Gagnon and L. J. Hendren. SableCC, an Object-Oriented Compiler Framework, TOOLS 26 Technology of Object-Oriented Languages, pp.140–154, 1998.

[6] Terence Parr and Russell Quong. ANTLR: A Predicated-LL(k) Parser Generator, Software Practice & Experience, Vol.25, No.7, pp.789–810, 1995.

[7] Terence Parr. The Definitive ANTLR 4 Reference, The Pragmatic Bookshelf, 2012.

[8] antlr/grammars-v4 - GitHub, https://github.com/antlr/grammars-v4 (accessed on Oct. 15, 2014).

[9] Gregor Kiczales, John Lamping, et al., Aspect-Oriented Programming, ECOOP '97 – Object-Oriented Programming: 11th European Conference, LNCS 1241, pp.220–242, Springer, 1997.

[10] BYACC - BERKELEY YACC, http://invisible-island.net/byacc/byacc.html (accessed on Sep. 20, 2014).

[11] BYACC/J Home Page, http://byaccj.sourceforge.net/ (accessed on Sep. 20, 2014).

[12] Martin Fowler. Domain-Specific Language, Addison-Wesley, 2011.

[13] Anders Hejlsberg, Mads Torgersen, Scott Wiltamuth and Peter Golde. The C# Programming Language, 3rd edition, Addison-Wesley, 2008.

[14] James Gosling, Bill Joy, Guy Steele and Gilad Bracha. The Java Language Specification, 3rd edition, 2005, http://docs.oracle.com/javase/specs/ (accessed on Sep. 20, 2014).

[15] Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides. Design Patterns, Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[16] deanwampler/Aquarium - GitHub, https://github.com/deanwampler/Aquarium (accessed on Sep. 20, 2014).