2012 International Conference on Networks and Information (ICNI 2012)
IPCSIT vol. 57 (2012) © (2012) IACSIT Press, Singapore
DOI: 10.7763/IPCSIT.2012.V57.16

Parser Development with an Internal DSL in Ruby

Kazuaki Maeda [1]

Department of Business Administration and Information Science, Chubu University, 1200 Matsumoto, Kasugai, Aichi 487-8501, Japan

Abstract. This paper describes an internal DSL (domain-specific language) to write syntax rules based on Ruby. Traditional parser generators provide their custom languages to define syntax rules. The parser generators read mixture of the syntax rules and action code written in one of general purpose languages. If we write action code containing trivial syntax errors, the parser generators do not analyze the code and they generate source code containing the syntax errors. To resolve this situation, an internal DSL, called RibLR, was carefully designed on a subset of Ruby. We can define syntax rules in RibLR and action code in a general purpose language. Ruby gives a terse description to define the syntax rules and we can embed action code in the generated parser with a flexible way. In the author's preliminary experience, productivity was improved in the design and implementation of parsers.

Keywords: Parser Development, Domain-Specific Language, Syntax Analysis, Ruby

1. Introduction

Most compilers read plain text, analyze it, and build a tree to represent hierarchical structure of source code. The first phase is a lexical analyzer (called a scanner), while the second phase is a syntax analyzer (called a parser). The scanner reads a sequence of characters, recognizes chunks of characters (called tokens), and passes them to the parser. In the next phase, the parser reads a sequence of tokens, recognizes syntactic structure, and builds an abstract syntax tree[1].

Development of parsers was complex before the 1970s. In the 1970s, the parser generator Yacc[2] 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. Since Yacc was introduced to parser development, many parser generators have been built[3,4,5,6,7,8]. The parser generators read syntax rules with action code, and generate source code pertaining to a target programming language. But the syntax rules are written in their custom languages. They are different from general purpose languages. This kind of languages is called DSL, which stands for Domain-Specific Language.

A DSL is a language dedicated to a particular problem domain. Martin Fowler wrote, in his book, two types of DSLs: an external DSL and an internal DSL[9]. All custom languages to represent syntax rules are different from general purpose languages so that they are the external DSLs. We write them in context free grammar style. The parser generators read mixture of the syntax rules written in the external DSL and action code written in one of general purpose languages, such as Java or C#.

BYACC/J[8] has developed for Java programmers as a variant of Yacc. If we develop the parser with BYACC/J and action code is written in Java, the parser generator reads them, and generates source code in Java. It is a useful tool to develop parsers in Java. However, if we write action code in Java containing trivial syntax errors such as omitting semicolons, BYACC/J generates source code still containing the syntax errors because it does not analyze the action code. The Java compiler finds the syntax errors after BYACC/J

[+] Corresponding author. Tel.: +81-568-51-1111; fax: +81-568-52-1505.
E-mail address: kaz@acm.org.

generates it, but the Java compiler does not know about syntax rules with action code which are defined before parser generation.

The author believes that an internal DSL is one of the approaches to resolve this situation. The internal DSL is a language represented with subset of a general purpose language. If we define syntax rules with action code in a same programming language, syntax errors are detected by language processors before generating source code. Moreover, we can use software development tools for the general purpose language, so that we can improve the productivity to develop programs with parsing. This paper describes an internal DSL, called RibLR, to write syntax rules based on Ruby. Ruby gives a flexible description to define the syntax rules and we can embed action code in the generated parser with a flexible way.

Section 2 explains related works about parser development. Section 3 explains syntax definitions in RibLR and parser generation. Section 4 summarizes this paper.

2. Parser Development Using Parser Generator

The parser generators read user-defined syntax rules with action code and generate parsers written in the target programming language. When a syntax rule is accepted during parsing, the generated parser invokes action code attached to the syntax rule.

The Parser generators are tightly coupled to their target programming languages. Berkeley Yacc (called BYACC[6]) was built as a Yacc variant. BYACC accepts any inputs that conform to the Yacc specification, and generates a parser written in the C programming language. It has been ported for other programming languages; the variants are Jay[7] and BYACC/J[8].

Fig. 1 shows a snippet of the syntax rules and the action code in Java for a simple arithmetic expression. It is written for BYACC/J which is an extension of the Berkeley YACC-compatible parser generator. Action code in Java is attached to each syntax rule located in the same file. The parser requirements are implemented using the action code. When the syntax rules are invoked, the action code attached to the rules is also invoked. For example, if the generated parser reads `2 * 3` as an input, the syntax rule

```
expr : expr PLUS term
    { System.out.println("plus expression");
      $$ = Util.add($1,$3); }
    | term
    { $$ = $1; }
    ;
term : term MULT NUMBER
    { System.out.println("mult expression");
      $$ = Util.mult($1,$3); }
    | NUMBER
    { System.out.println("integer constant");
      $$ = $1; }
    ;
```

Fig. 1: Syntax rules of arithmetic expression with action code

```
expr : expr PLUS term
    { System.out.println("plus expression");
      $$ = Util.add($1,$3); }
    | term
    { $$ = $1; }
    ;
term : term MULT NUMBER
    { System.out.println("mult expression")
      $$ = Util.mult($1,$3) }
    | NUMBER
    { System.out.println("integer constant");
      $$ = $1; }
    ;
% byaccj -J -Jclass=CalcParser parse-error.y

% javac CalcParser.java
CalcParser.java:458: ';' expected
{ System.out.println("mult expression")
                                        ^
CalcParser.java:459: ';' expected
    yyval.obj = Util.mult(val_peek(2).obj,val...

2 errors
```

Fig. 2: Syntax rules of arithmetic expression containing syntax errors in action cod

term : term MULT NUMBER

is matched and the action code

System.out.println("mult expression");

$$ = Util.mult($1,$3);

is executed. 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 of the right-hand side, and `$3` represents an attribute of the third symbol. These symbols are available on all syntax rules, and they are handled with a stack inside the generated parser.

BYACC/J does not find syntax errors in action code. Let's think we write action code in Java containing trivial syntax errors such as omitting semicolons shown in Fig. 2. The figure shows that BYACC/J does not analyze action code and it generates source code containing the syntax errors in Java. After the code generation, the Java compiler finds the syntax errors at line number 458 and 459, but the Java compiler does not know which action code contains the errors.

A domain-specific language (DSL) is a computer language with limited expressiveness focused on a particular domain. Martin Fowler wrote two types of DSLs: an external DSL and an internal DSL[9].

- An external DSL is a domain-specific language 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.

External DSLs have their own custom syntax. The examples include Yacc grammar files, XML configuration files and SQL. The syntax is completely different from general purpose languages such as Java and C# so that we need parsers using text parsing techniques in the applications with the external DSLs.

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

If we develop parsers to analyze source code in modern programming languages (e.g., C# or Java), it is very difficult to define complete syntax rules without reduce/reduce conflicts. Let us consider writing syntax rules for C#. The author's experience shows that, according to the specification written in the C# book[10], more than 700 syntax rules for C# were defined. A similar situation exists for Java; according to the Java specification[11], more than 500 syntax rules for Java were defined.

After finishing development of a parser in Java using BYACC/J, if we develop another parser in C using Yacc, we do not need to change the syntax rules themselves because the external DSL has same syntax. However, we need to modify action code written in the target programming language from Java to C. This is not easy task if we need to define hundreds of syntax rules. Moreover, maintenance problems may be happened in the case that we need two versions of parsers in Java and C with same syntax rules.

The author believes that an internal DSL is one of the approaches to resolve these situations. If we define syntax rules and action code in an internal DSL, we can use attractive software tools for parser development.

3. RibLR for Language Recognition

From discussions in the previous section, an internal DSL to define syntax rules, called RibLR, was designed. The syntax rules are written in context free grammar. Fig. 3 shows an example of syntax rules for a simple arithmetic expression. The operator **.>** means derivation. For example, a 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 RibLagr, was developed to generate a bottom-up parser. RibLagr reads user defined syntax rules in RibLR and action code written in Ruby. After that, it generates a bottom-up parser shown in Fig. 5. The generated parser is composed of a driver for parsing, action code in Ruby, and parsing data

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

Fig. 3: Syntax rules and token definitions for a simple arithmetic expression in RibLR

```
action(:term,:term,:MULT,:NUMBER) do
  $params[0] = $params[1] * $params[3]
end
action(:term,:NUMBER) do
  $params[0] = $params[1]
end
```

Fig. 4: User defined actions for a simple arithmetic expression

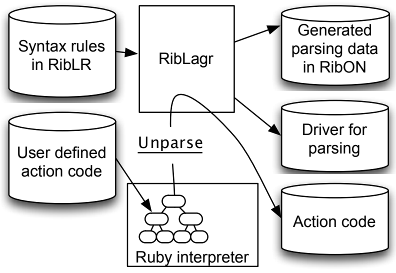

Fig. 5: Program generation by RibLagr and user defined action code

written in RibON which the author has already developed[12]. Before generating source code, action code defined by users is loaded into the Ruby interpreter, but RibLagr reconstructs it to source code and embeds the code to the generated parser. The generated parser reads tokens and decides whether the sequence of tokens conforms to the syntax rules.

RibLagr reads syntax rules and user defined actions, and then generates parsing data represented in RibON. Fig. 4 shows action code for the third rule

:term .> :term,:MULT,:NUMBER and action code for the fourth rule

:term .> :NUMBER

in Ruby. Special symbols $params[0], $params[1], and $params[3] are attributes in the symbols :term, :term and :NUMBER of the third syntax rule respectively. The important point in Fig. 4 is that action is a method call in Ruby. RibLagr unparses the do...end block, reconstructs the source code, and generates action code in Ruby. If there are any syntax errors in action code as the following:

```
action(:term,:NUMBER) do
  $params[0] = $params[1
end
```

the Ruby interpreter detects lack of ']' before generating the parser.

After generating the parser, we can execute syntax analyzer shown in Fig. 6. At that timing, we can switch action code to another code. The syntax of action code is same shown in Fig. 7 as action code during parser generation.

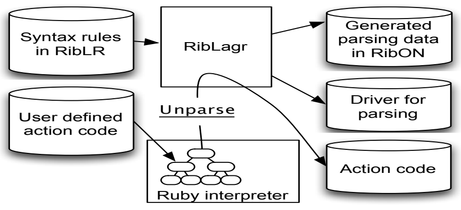

Fig. 6: Defined action switched to another code

```
action(:term,:term,:MULT,:NUMBER) do
   puts 'mult expression'
    $params[0] = $params[1] * $params[3]
end
```

Fig. 7: User defined action code at runtime

4. Summary

This paper described an internal DSL, called RibLR, to write syntax rules based on Ruby. Traditional parser generators provide their custom languages to define syntax rules. The parser generators read mixture of the syntax rules and action code written in one of general purpose languages. If we write action code containing trivial syntax errors, the parser generators do not analyze the code and they generate source code containing the errors. To resolve this situation, RibLR was carefully designed on a subset of Ruby. If we define syntax rules in RibLR and action code in Ruby, syntax errors are detected by the Ruby interpreter before generating source code. A generator, called RibLagr, was developed to generate a bottom-up parser. RibLR gives a terse description to define the syntax rules and we can embed action code in the generated parser using RibLagr with a flexible way. Presently RibLR and its related tools are now still under development. The results will be published in a future paper.

5. References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers : Principles, Techniques, and Tools*, 2nd edition, Pearson Education, 2006.

[2] Steven C. Johnson. Yacc: Yet Another Compiler Compiler, *UNIX Programmer's Manual*, vol.2, 353-387, 1979.

[3] Bison - Gnu Parser Generator, http://www.gnu.org/s/bison/ (accessed at Sep. 26, 2012).

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

[5] T. J. Parr and R. W. Quong. ANTLR: A Predicated-LL(k) Parser Generator, *Software Practice & Experience*, pp.789-810, Vol.25, No.7,1995.

[6] BYACC - BERKELEY YACC, http://invisible-island.net/byacc/byacc.html (accessed at Sep. 26, 2012).

[7] jay (Language Processing), http://www.cs.rit.edu/~ats/projects/lp/jay/package.html (accessed at Sep. 26, 2012).

[8] BYACC/J Home Page, http://byaccj.sourceforge.net/ (accessed at Sep. 26, 2012).

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

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

[11] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*, 3rd ed., 2005, http://docs.oracle.com/javase/specs/ (accessed at Sep. 26, 2012).

[12] Kazuaki Maeda. Ruby-based Data Representation and the Performance in Java Programs, *Second International Conference on the Applications of Digital Information and Web Technologies*, pp.814-819, 2009.

Kazuaki Maeda graduated in Department of Administration Engineering from Keio University in Japan, and graduated in Graduate School of Science and Technology from Keio University. After leaving Keio University, he got an academic position at Chubu University in Japan. Now he is a professor at Department of Business Administration and Information Science at Chubu University in Japan. He is a member of ACM, IEEE, IPSJ and IEICE. His research interests are Compiler Construction, Domain-Specific Languages, Object-Oriented Programming, Software Engineering and Open Source Software.