Quick Parser Development Using Modified Compilers and Generated Syntax Rules

KAZUAKI MAEDA
Department of Business Administration
and Information Science, Chubu University
1200 Matsumoto, Kasugai, Aichi, JAPAN
kaz@acm.org

Abstract: This paper describes some ideas about quick parser development from source code of popular free and libre open source (called FLOSS) compilers using modified parser generator. Parser development is time-consuming and laborious according to traditional approaches. In this paper's approach, a specialized scanner is built using FLOSS compilers. The scanner reads source code, analyzes it, and writes a sequence of tokens in XML. The parser generator is modified so that it generates a specialized parser and syntax rules with code to read the tokens in XML for quick parser development. In the author's experiences, it takes within one hour to implement a simple recognizer which reads C# source code, writes a sequence of tokens in XML and analyzes it.

Key–Words: Parser Development , Parser Generator, FLOSS Compiler, Syntax Rule

1 Introduction

For centuries past, innovations (e.g. in industrial revolution) had drastically changed industrial environments. Vinod Khosla said " Science has the potential to do something not 10 percent better or 20 percent better but 100 times better, and that power is what is so exciting to me[1]." Free and Libre Open Source Software (called FLOSS) has the potential to improve something 100 times better, and that power is expected to excite software developers.

FLOSS has been adopted by many companies, organizations and governments. An important reason for the proliferation of FLOSS is lower costs. Typical consumers can work on computers if operating systems and office suites are installed on their computers. Nowadays, we can freely download Linux as an operating system and OpenOffice as office suites. As a result of adapting FLOSS, the initial cost to obtain software is becoming drastically lower. In addition, many FLOSS products are already available on multi-platforms, e.g. Linux, Windows and Mac OS X. If we develop applications on top of FLOSS, tasks for porting the applications from a FLOSS platform to another one are easier than traditional commercial platforms.

There are many FLOSS projects for improving the software productivity. Those are middleware, Web frameworks, programming languages, integrated development environments and so on. If developers master how to use the software, they can improve productivity of software development. Developers, however, use the software as just only a black box.

The origin of the term "Free and Libre Open Source Software" is that source code is freely available. The source code is, however, not effectively used to improve the productivity of developing another software.

This paper describes an effective method to reuse source code of popular FLOSS compilers and to develop parsers quickly. The method has two features.

- Implementation of a scanner to write a sequence of tokens in XML

  Most of compilers does not produce lexical information to developers. If compilers write a sequence of tokens, we can easily implement token based software tools.

- Customization of parser generators for producing syntax rules and source code to read tokens written by the scanner

  We can use the syntax rules without modification so that we can develop our own parser quickly. Source code of FLOSS compilers is checked by world wide software developers to improve the quality and functionality. Therefore if the FLOSS compilers are mature, the parser becomes reliable.

In section 2, parser and scanner development and FLOSS compilers are explained. In section 3, specialized scanners and customized parser generators are explained. The final section provides a summary of this paper.

2 Scanners and Parsers in Compilers

Most of compilers read plain text, analyze it and build a tree to represent the hierarchical structure of source code. The first phase is a lexical analyzer (called scanner), and the second phase is a syntax analyzer (called parser). This section describes the scanner, the parser, and FLOSS compilers.

2.1 Scanners

The scanner reads a sequence of characters, recognizes chunks of the characters (called tokens), passes them to the parser. For example, Figure 1 shows that the scanner reads a sequence of characters “1 + 2 * 3”, discards white spaces, produces 5 tokens: three numbers and two operators. Developers usually define the token patterns using regular expressions to design the scanner. The regular expressions are used to represent just only patterns of a sequence of characters so that we sometimes need syntactic and semantic information to decide what kind of token it is.

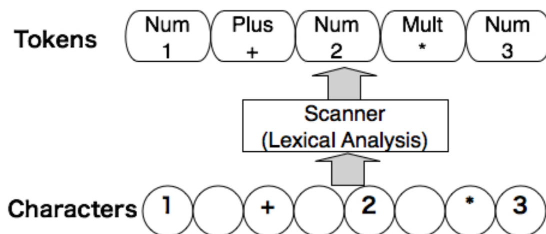

Figure 1: Tokens produced by the scanner

The patterns using regular expressions are used for scanner generators. Lex[2] is a traditional scanner generator, which reads descriptions of possible tokens and generates a scanner written in the C programming language. Scanner development using Lex is usually easier than scanners by hand. But in my best knowledge, scanners for typical FLOSS compilers and interpreters (e.g. GCC[3], Mono C# compiler[4] and Ruby interpreter[5]) are developed not by the scanner generator, but by hand. Because the scanners need fine grained customization.

These scanners are developed only for compilers. It is difficult to extract only the scanners to reuse for other purposes, for example, to build token based software tools such as pretty printers or token based clone code detectors[6]. Because they are tightly coupled with parsers and symbol table handlers in the compilers.

2.2 Development of Parsers Using Parser Generators

The parser reads a sequence of tokens, recognizes a syntactic structure, and build an abstract syntax tree[7]. For example, Figure 2 shows that the parser reads 5 tokens, analyzes them in according to the syntactic structure, and build the abstract syntax tree to represent the arithmetic expression.

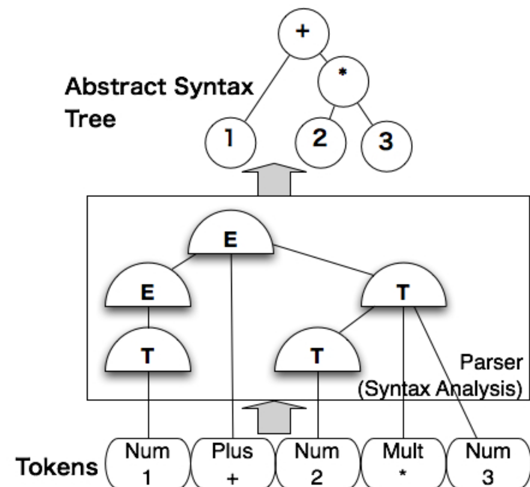

Figure 2: Abstract syntax tree produced by parsers

In the 1970s, the parser generator Yacc[8] was developed, which made parser development much easier. Yacc reads user-defined syntax rules with action codes to be invoked when the syntax rules are recognized, and it generates an LALR (lookahead LR) parser[7] written in the C programming language.

At invoking the syntax rules, the action codes related with the rules are also invoked. The requirements to the parser are implemented using the action codes. For example, if we calculate the simple arithmetic expression, we can define the syntax rules with the action codes shown in Figure 3. The action codes are surrounding by curly braces. $$ is a symbol to represent an attribute with the left-hand side of each rule. $1 is a symbol to represent an attribute with the first symbol of the right-hand side, and $3 is a symbol to represent an attribute with the third symbol.

It is easy to define simple syntax rules as shown in Figure 3. But if we develop parsers to analyze source code in modern programming languages (i.e. C# or Java), it is very difficult to define complete syntax rules without reduce/reduce conflicts[7]. Let us consider writing syntax rules for the C# programming language. The author’s experience shows that, straight according to the specification written in the C# book[9], 718 syntax rules for C# were defined but they include 149 reduce/reduce conflicts. Let us con-

```
E : E Plus T
  { $$ = $1 + $3; }
  | T
  { $$ = $1; }
  ;
T : T Mult Num
  { $$ = $1 + $3; }
  | Num
  { $$ = $1; }
  ;
```

Figure 3: Syntax rules with actions for a simple arithmetic expression

sider writing syntax rules for the Java programming language. In the same way, according to the Java specification[10], 526 syntax rules for Java were written but they include 372 reduce/reduce conflicts.

It is desirable to modify the conflicted rules. If a parser reads one symbol and more than one syntax rule can be reduced, reduce/reduce conflict occurs among the syntax rules. We need to keep all rules consistent without reduce/reduce conflicts, but it is time-consuming and laborious to do so. Because the specification of modern popular programming language is very complicated. For example, in my experience, it took about one week to modify the syntax rules for Java and complete the parser without reduce/reduce conflicts. In addition, we need to spend much time for writing code to build an abstract syntax tree.

Based on the author's experience of constructing many parsers, there are two approaches to develop parsers quickly.

1. To get syntax rules from major web sites

   There are some web sites including collections of syntax rules for major programming languages[11]. The collections in the web sites are very useful to improve the productivity of parser development. On the other hand, many syntax rules contain some errors and there is no guarantee that they are strictly correct. As a result, we must laboriously test the syntax rules to improve the quality.

2. To extract source code of the parser from FLOSS compilers

   There are many high-quality FLOSS compilers available such as GCC and Mono C# compiler. These compilers, however, have been developed only to generate object code from source code. It is difficult to extract only the parser to reuse for other purposes because it is tightly coupled with other modules in the compiler (e.g. lexical analyzer and symbol table handler).

This paper describes another approach, that is, to modify scanners and parser generators for writing reusable information. Functionality for writing tokens is embedded in scanners and functionality for writing syntax rules and lexical information is embedded in parser generators. If we use the modified scanner and parser, we can develop parsers quickly.

2.3 FLOSS Compilers

Mono and GCC are the most important compilers in FLOSS.

Mono is a UNIX implementation of the Microsoft .NET development environment and development tools[4]. It is a FLOSS project commercially supported by Novell. The objective of the Mono project is to enable UNIX developers to build and deploy cross-platform .NET applications. Mono is currently available on major operating systems of Linux, Mac OS X, and Microsoft Windows. It implements various technologies developed by Microsoft that have been submitted to the ECMA International. The Mono distribution includes C# compilers, a VB.NET compiler, the class libraries, and runtime environments including a class loader, just-in-time compiler and a garbage collector.

In the core distribution, the total source code exceeds 3M lines of code, while that for the Mono C# compiler (called Mcs in this paper) exceeds 70K lines of code. The conceptual structure of the Mcs is shown in Figure 4. The parser for Mcs is developed using a parser generator Jay[12]. Jay reads syntax rules for the C# programming language and generates a parser written in C# to analyze C# source code.

GCC is a GNU Compiler Collection[3]. It basically supports C, C++, Objective-C, Fortran, Java and Ada. To compiler Java source code, we use GCJ written in the C programming language. One of the advantages is that GCJ can generate both native code and bytecode from Java source code. Many applications have already been developed with GCJ. For example, all of the major GNU/Linux distributions use GCJ to distribute programs like OpenOffice, Eclipse and Tomcat.

The parser for GCJ was developed using a parser generator Bison[13]. In the case of GCJ, Bison reads syntax rules for the Java programming language and generates a parser written in the C programming language to analyze Java source code. Bison is used to develop many parsers (e.g. the parser for Ruby).

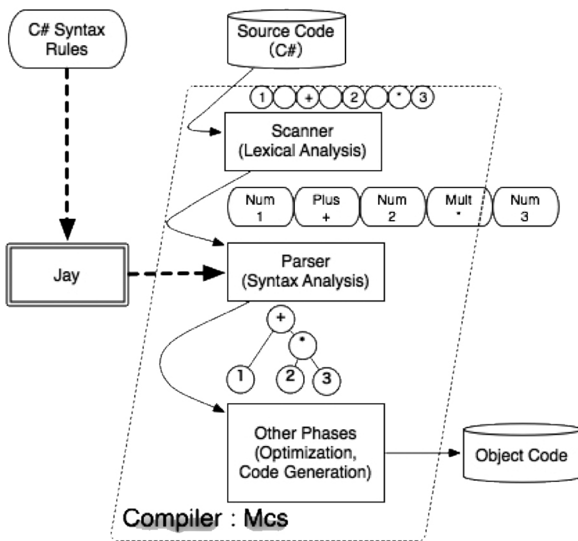

Figure 4: Conceptual structure of Mcs

3 Modification of Compilers and Parser Generators

FLOSS compilers and parser generators are modified so that functionality for writing tokens is embedded in the compilers and functionality for writing syntax rules and lexical information is embedded in the parser generators.

3.1 Modified Compilers as Scanners

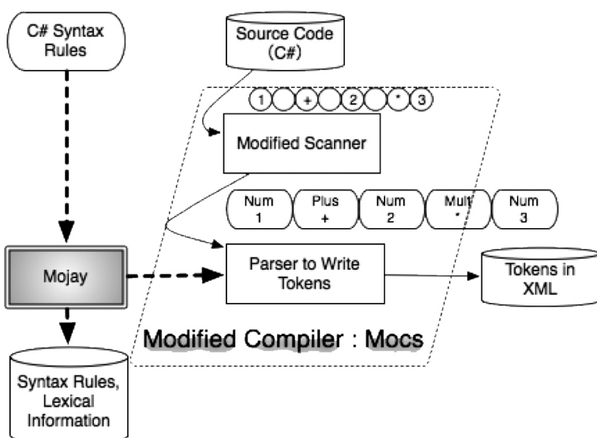

Figure 5: Modified version of Mcs

Source code of the C# compiler Mcs was obtained and modified for an implementation to produce a sequence of tokens from source code. The modified version of Mcs is called Mocs in this paper. Mocs reads C# source code and produces a sequence of tokens as shown in Figure 5.

Jay was modified to implement Mocs and it is called Mojay. Mojay reads syntax rules for C# and generates a special parser for Mocs which contains functionality to produce the tokens. For example, if Mocs reads C# source code shown in Figure 6, it produces a sequence of tokens in XML shown in Figure 7. Table 1 shows the meanings of the attributes in XML.

```
using System;
namespace Com.Xyz
{
    public class Hello
    {
    }
}
```

Figure 6: C# source code

```
<lex tk="USING" va="using" li="1" co="1" />
<lex tk="IDENTIFIER" va="System" li="1" co="7" />
<lex tk="SEMICOLON" va=";" li="1" co="13" />
<lex tk="NAMESPACE" va="namespace" li="2" co="1" />
<lex tk="IDENTIFIER" va="Com" li="2" co="11" />
<lex tk="DOT" va="." li="2" co="14" />
<lex tk="IDENTIFIER" va="Xyz" li="2" co="15" />
<lex tk="OPEN_BRACE" va="{" li="3" co="1" />
<lex tk="PUBLIC" va="public" li="4" co="5" />
<lex tk="CLASS" va="class" li="4" co="12" />
<lex tk="IDENTIFIER" va="Hello" li="4" co="18" />
<lex tk="OPEN_BRACE" va="{" li="5" co="5" />
<lex tk="CLOSE_BRACE" va="}" li="6" co="5" />
<lex tk="CLOSE_BRACE" va="}" li="7" co="1" />
```

Figure 7: Fragment of tokens in XML

Table 1: Attributes in tokens

Name	Meaning of the attributes
tk	kind of a token
va	string image of a token
li	line number
co	column number

Mocs plays a role of a scanner. Mocs reads C# source code, analyzes it lexically and syntactically, and it writes a sequence of tokens. The parser generated by Mojay is included in Mocs and Mocs can generate object code in the same way as Mcs, but the main role of Mocs is to write a sequence of tokens.

Source code of GCJ was obtained and modified for an implementation to produce a sequence of tokens from Java source code. The parser for GCJ is developed using a parser generator Bison[13]. In the

case of GCJ, Bison reads syntax rules for the Java programming language and generates a parser written in the C programming language to analyze Java source code. Bison was modified to implement modified version of GCJ. The modified Bison reads syntax rules for the Java programming language and generates a special parser which contains functionality to produce the tokens.

The parser of Ruby interpreter is also developed using Bison. Source code of Ruby interpreter was obtained and modified in the same way as GCJ to produce a sequence of tokens from Ruby source code.

As previous described, scanners for typical compilers and interpreters are developed by hand. Moreover, there are no standard way to pass lexical information from the scanner to the parser. Therefore, the author have read all source code of the scanners for three language processors (i.e. C#, Java and Ruby) one by one, and modified source code to get lexical information of each token.

3.2 Parsers Generated from Extracted Syntax Rules

Mocs plays a role of a scanner, the next phase to develop is a parser. Using syntax rules and code generated by Mojay, we can develop our own parser quickly.

Mojay reads the specification for C# and generates the parser. In addition, it generates syntax rules with source code to read tokens. Figure 8 shows fragment of the syntax rules. The rules do not have any action code as shown in Figure 3. All action code and other description are eliminated from the original input for jay.

```
outer_declarations : outer_declaration
  | outer_declarations outer_declaration
  ;
outer_declaration : extern_alias_directive
  | using_directive
  | namespace_member_declaration
  ;
using_directives : using_directive
  | using_directives using_directive
  ;
using_directive : using_alias_directive
  | using_namespace_directive
  ;
using_namespace_directive :
    USING namespace_name SEMICOLON
  ;
```

Figure 8: Fragment of the syntax rules generated by Mojay

If we build an abstract syntax tree to pass it to another phase, we need to embed the action codes in

```
public static void RegisterToken(HashMap hashToken) {
  hashToken.put("EOF",Token.EOF);
  hashToken.put("NONE",Token.NONE);
  hashToken.put("ERROR",Token.ERROR);
  hashToken.put("FIRST_KEYWORD",Token.FIRST_KEYWORD);
  hashToken.put("ABSTRACT",Token.ABSTRACT);
  hashToken.put("AS",Token.AS);
  hashToken.put("ADD",Token.ADD);
  hashToken.put("ASSEMBLY",Token.ASSEMBLY);
  hashToken.put("BASE",Token.BASE);
  hashToken.put("BOOL",Token.BOOL);
  hashToken.put("BREAK",Token.BREAK);
  hashToken.put("BYTE",Token.BYTE);
  hashToken.put("CASE",Token.CASE);
```

Figure 9: Fragment of generated code by Mojay

appropriate syntax rules in the same manner as traditional parser development using the parser generator.

3.3 Software Tool Development Using Mocs and Generated Syntax Rules

In the author's experiences, it takes within one hour to implement a simple recognizer which reads C# source code, writes a sequence of tokens in XML and analyzes it as shown in Figure 10. If we develop a simple reverse engineering tool for C#, we should implement the simple recognizer at first and then build some functions into it to analyze relationship between classes and to generate class diagrams.

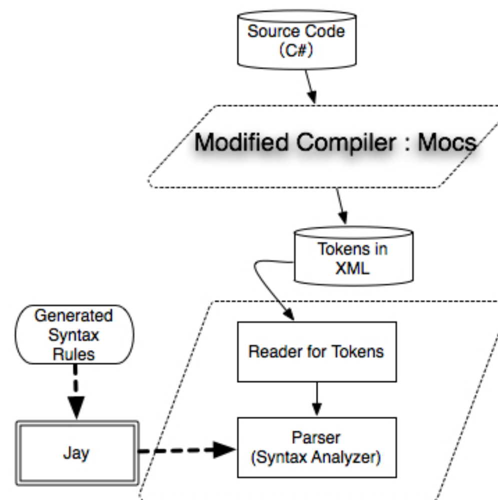

Figure 10: Our own C# parser

The author have already developed a commercial based reverse engineering tool. It reads C# source code and generates the class information. The reverse engineering tool consists of 6,750 lines of C# source code and the syntax rules consists of 2,115 lines. Some classes were implemented to build an ab-

stract syntax tree. The tool was developed on Mac OS X using Mono. After completing the development of the production quality version, the source code was transferred to another PC (running on Windows XP), and the author attempted to build the executable file using Cygwin and Visual Studio 2005. This building work was very simple, and it was carried out without any problems. This is because XML and C# function independently of operating systems and computers.

4 Summary

This paper described some ideas about quick parser development from source code FLOSS using modified parser generator. Development of parsers is time-consuming and laborious according to traditional parser development. We can develop parsers quickly using the specialized scanner and the modified parser generator.

An advantage of this method is that source code of FLOSS compilers is publicly opened to distributed software developers and checked by them to improve the quality and functionality. We can get the most up-to-date source code of FLOSS compilers with high quality so that the parsers developed using this method have high quality. However, we should pay attention to the fact that everything is inherited from the original FLOSS compilers. If the FLOSS compilers are under GPL (GNU General Public License), the developed parsers are also under GPL. We must separate the parser from other modules, if we need to escape from GPL.

References:

[1] Vinod Khosla, On " science " and climate " meltdown, " http://venturebeat.com/2006/10/15/vinod-khosla-on-science-and-climate-meltdown/.

[2] John R Levine, Tony Mason and Doug Brown, LEX & YACC, Second Edition, O'Reilly, 1992.

[3] Free Software Foundation. *GCC, the GNU Compiler Collection – GNU Project,.* http://gcc.gnu.org/.

[4] *Main Page - Mono.* http://www.mono-project.com/.

[5] Ruby Programming Language, http://www.ruby-lang.org/en/

[6] Toshihiro Kamiya, Shinji Kusumoto and Katsuro Inoue, CCFinder: A Multilinguistic Token-Based Code Clone Detection System for Large Scale Source Code, IEEE transaction of Software Engineering, vol.28, no.7, pp.654–670, 2002.

[7] Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman, *Compilers : principles, techniques, and tools, 2nd Ed.*, Pearson Education, 2006.

[8] Steven C. Johnson, Yacc: Yet another compiler compiler, In *UNIX Programmer's Manual*, volume 2, 353–387, 1979.

[9] Anders Hejlsberg, Mads Torgersen and others, The C# Programming Language, Third Edition, Addison-Wesley, 2009.

[10] The Java Language Specification, Third Edition, http://java.sun.com/docs/books/jls/index.html.

[11] *Grammar List*, http://www.antlr.org/grammar/list.

[12] jay (Language Processing), http://www.cs.rit.edu/ ats/projects/lp/doc/jay/.

[13] Bison – GNU parser generator, http://www.gnu.org/software/bison/