Extending ExtendJ with CUP2 Parser Support

Project in Computer Science – EDAN70
January 28, 2016

Karl Rikte
D10, Lund University, Sweden
atf07kri@student.lu.se

Abstract

JastAdd[9] is a project developed at the CS departement of Lund University to add easy AOP (Aspect Oriented Programming) to the Java language, which is useful for AST generation in compilers. Many projects that use JastAdd also use the Beaver parser.

This paper aims to address how to use alternate parsers with JastAdd projects. It focuses - in particular - on adding CUP2[7] parser support to ExtendJ[6], a compiler for the Java language, written in Java, using JastAdd.

Different methods to implement additional parsers support were investigated, and the approach involving the least refactoring was taken. A parser grammar translator program, for easy CUP2 integration with JastAdd projects that use Beaver, has been developed.

Performance and features evaluation has been conducted comparing the Beaver parser to CUP2.

1. Introduction

JastAdd is a framework for building an AST (Abstract Syntax Tree) from a grammar specification. JastAdd adds the possibility of AOP (Aspect Oriented Programming) to Java, [4] meaning a file can contain all code for a specific functionality, and add methods to many different Java AST classes. JastAdd also allows inherited attributes (inheriting from ancestor nodes, not Java inheritance), circular attributes, etc. JastAdd will be described in more detail in section 3.1.

JastAdd is often used together with Beaver, a parser generator. Beaver takes a parser grammar in the form of a .beaver file. It produces a LALR(1) parser with token definitions written in the Java language. [8] Beaver is quite mature and works well, however, Beaver has it's minor imperfections, as discussed in 3.2,

CUP2 is a very new parser generator that is actively developed. It is the successor of the commonly used parser generator CUP. It has many exciting features that few other parsers have, as it uses Java to express grammar. [7] CUP2 will be discussed in section 4.

As Beaver has some minor imperfections that could possibly be addressed by other parsers, and it may be interesting to compare the speed of Beaver to the speed of other parsers, it was decided to evaluate other parsers for use with JastAdd projects.

Thus, the goal of this paper is to show how multiple parser generators can be integrated with JastAdd, and to evaluate CUP2 as an alternative to Beaver in this context.

The JastAdd project of choice was ExtendJ[6], a compiler for the Java language, written in Java, using JastAdd. As ExtendJ is a quite large and advanced project, and it has many tests, it should be ideal for use for parser evaluation purposes. ExtendJ is built to be extendable, using the principles discussed in [5], which was another reason to choose it as a target project.

To achieve the goal, I have developed a grammar translation program, BeaverToCUP2, to aid in migrating from Beaver to CUP2. It also showcases a modular parser architecture, supporting use of either Beaver or CUP2 at runtime. This will be discussed in section 5.

The results from the evaluation conducted in this project can be summarized as follows: CUP2 is currently approaching beta state, and seems to be fully working, but is a bit lacking in start up performance. The parsing speed is about 20% slower than Beaver, however, parsing time is a small part of the total compile time in ExtendJ, as seen in section 7.3.

Future work could tell if the added features are worth the performance penalty. Also, as CUP2 matures, it may become faster than it is today.

2. Additional goals

In addition to the main goal discussed above, further goals of the project were set up as follows:

- Ease of integration:
  - Minimize modifications to ExtendJ build system and source code
  - Minimize modifications to JastAddParser (used by ExtendJ)
- Ease of use and maintainability:
  - Good, simple, extendable design, reducing the need for comments and documentation.
  - Produce well formatted and human readable output, with comments where necessary.
- Evaluation of CUP2 compared to Beaver:
  - Features and stability
  - Performance
- Provide feedback to developers of ExtendJ and CUP2:
  - Bug reports
  - Potential future refactoring that could be beneficial for multiple parsers support for JastAdd projects

3. Background

ExtendJ makes use or a couple of different projects. These will now be discussed. How they interact will be explained in figures and in text.

3.1 JastAdd

JastAdd, as mentioned earlier, allows AOP (Aspect Oriented Programming). Development using JastAdd is normally done as follows:

- Describe the AST node classes in an .ast file:
  - What AST class has which children AST classes, and are the children optional or lists.
  - What AST classes extend other AST classes.
- Declare methods and fields in .jrag files:
  - A single .jrag file can declare functions and fields in multiple generated AST classes, using ClassName.name notation.
  - Declare how attributes (methods or values) are inherited from parent nodes (not by use of java inheritance).

JastAdd supports many more features and use cases, for more in-depth analysis, please consult [4] and [5].

3.2 Beaver

Beaver is an LALR(1) [1] parser generator. It takes a grammar expressed in EBNF [3], with semantic actions (code that runs each time the parser decides to reduce). Many reference grammars fail with LALR(1) due to being ambiguous. [2] Beaver allows precedences to be set, which allow otherwise ambiguous grammars to be unambiguous.

Beaver works very well with JastAdd but there are a couple of minor imperfections that need to be handled.

Beaver does not support multiple grammar files. Working with say, a compiler with a very large grammar, it could be beneficial if one file could contain all grammar for expressions, and another file all grammar for statements.

Beaver forces the AST nodes to extend one of the classes in the Beaver system (beaver.Symbol). This is not ideal in all situations. The purpose beaver.Symbol is to keep track of line number information, and seems to be mostly intended for use with tokens. In many cases, none of the functionality of beaver.Symbol is used. One may also ask why the AST should extend a class from the parser at all, it could be considered a non modular design.

Specific line orders mandate types to be declared far from where they are used. Scanner interface etc. may bloat the grammar. Beaver is not actively maintained, it has not seen an update since 2012.

3.3 JastAddParser

Some projects that use JastAdd use Beaver directly, especially small projects. Larger projects can use a preprocessor called JastAddParser to address some of the limitations of Beaver such as:

- Allow splitting grammar into several files
- Declare types close to the productions, where they are used
- Automatic generation of JastAdd's representation for lists and optional nodes
- Get rid of some boilerplate code, such as scanner interface, result cast etc.

JastAddParser will produce a .beaver file for processing by Beaver as shown in figure 1.

3.4 Early decisions

After investigating how to add multiple parsers support to JastAdd projects, two possible options were considered:

1. Implement support for additional parser(s) in JastAddParser. Instead of generating a .beaver file, generate a parser specification for some other parser generator. JastAddParser is shown in figure 1.
2. Replace the Beaver step with a translator, that translates the .beaver file, produced by JastAddParser, to a specification for some other parser generator. This is demonstrated in figure 2.

Reasons for choosing option 2:

- Overwhelming refactoring required to implement additional parsers in JastAddParser. JastAddParser very dependent on Beaver, a complete rewrite may be in order.
- More control over the project, less dependent on others.
- Even smaller projects, which do not use JastAddParser, but rather Beaver directly, can use CUP2. This would not be the case if implemented in JastAddParser.
- What if it turns out that Beaver is the best parser generator? "If it ain't broken, don't fix it!" (unknown origin) all refactoring could be for nothing.
- Simplicity of integration. JastAddParser produces beaver dependent code, that is needed by ExtendJ. Use this code without modification.

Figure 1. How JastAddParser is used in ExtendJ. Several .parser files are concatenated to an .all file which is fed to JastAddParser, producing a .beaver file for later processing by Beaver.

Figure 2. The way additional parser support was implemented. Instead of sending the .beaver file produced by JastAddParser to Beaver, it is sent to the grammar translator, BeaverToCUP2, producing a CUP2 parser that can be used with ExtendJ.

4. CUP2

CUP2 uses only ordinary Java code to express a parsing grammar.

4.1 Minimal specification

A minimal example is shown below:

```
import static MyCUP2Spec.Terminals;
import static MyCUP2Spec.NonTerminals;

class MyCUP2Spec extends CUP2Specification{

    enum Terminals extends Terminal{
        NUMERAL
    };

    enum NonTerminals extends Terminal{
        num
    };

    class NUMERAL extends SymbolValue<String>;
    class num extends SymbolValue<String>;
```

```
    public MyCUP2Spec(){
        grammar(
            //grammar goes here
        );
        precedences(
            //optional precedences go here
        );
    }
}
```

First, the Terminals and NonTerminals enums were imported statically to be able to reference Terminal.NUMERAL as simply NUMERAL later in the grammar. CUP2 imports were omitted for readability. Then our specification extends CUP2Specification. Listing all terminals and non terminals is done using enums extending Terminal and NonTerminal receptively. Declaring the type of a terminal or non terminal is done by declaring a class with the name of the symbol and extending SymbolValue with appropriate type using Java generics. In the constructor, grammar and optional precedences should be declared.

4.2 Minimal Grammar

As an example, consider the following (ambiguous) Beaver grammar:

```
expr =
    expr.l ADD expr.r   {return new Add(l, r);}    |
    NUMERAL.n           {return new Num(n);}       ;

stmt =
    ID.s EQ expr.e SEMI {return new Assign(s, e);} ;
```

In CUP2, this would be expressed as:

```
grammar(
    prod(expr,
        rhs(expr, ADD, expr), new Action(){
            public Expr a(Expr l, Expr r){
                return new Add(l, r);
            }
        },

        rhs(NUMERAL), new Action(){
            public Expr a(String n){
                return new Num(n);
            }
        }
    ),

    prod(stmt,
        rhs(ID, EQ, expr, SEMI), new Action(){
            public Stmt a(String s, Expr e){
                return new Assign(s, e);
            }
        }
    )
)
```

Note that calls to prod() are listed in grammar(). Each production goes in a prod() call. The first parameter is the result of the production, the second parameter is the RHS (Right Hand Side), which goes in a call to rhs(). Alternative right hand sides, with corresponding semantic actions, can listed in each prod(). Also note that only some of the symbols in each RHS are present as parameters in the semantic action method a(), only those symbols that have been declared as carrying a value.

CUP2 also provides a simple way to assign preferences. For information on this, or for a full description of the CUP2 parser generator, please consult [7].

5. BeaverToCUP2

As explained in section 3.4, it was concluded that the route most worth pursuing was to write a .beaver file to CUP2 specification translator. It was decided to name it BeaverToCUP2.

The full source code for BeaverToCUP2, aswell as all files referenced in this document are available at:

`https://bitbucket.org/edan70/2015-cup-karl`

5.1 Build process

The build script will first build BeaverToCUP2.jar with only Beaver support. Then the resulting BeaverToCUP2.jar will run with its' own .beaver file as input, and produce a CUP2 specification for itself. Then the BeaverToCUP2.jar will be re-built with CUP2 support as well.

5.2 Implementation architecture

BeaverToCUP2 showcases how a modular parser architecture may be implemented. The scanner is incomplete, and a header file must be concatenated with the scanner for it to be processed by the scanner generator. Which header file is used depends on the parser in use. This way, the scanner can be kept independent of the parser in use.

The ParserFactory can instantiate either CUP2Parser or BeaverParser. Both do not necessarily need to be present. The code compiles anyway, as the classes are loaded dynamically, on demand. (may throw ClassNotFoundException)

5.3 Semantic actions

Beaver automatically generates semantic action code in some cases. This functionality had to be implemented in BeaverToCUP2 in order to work as a replacement for Beaver. Consider the production:

```
expr = add;
```

This production, in Beaver, actually implies:

```
expr = add.a {return a;};
```

CUP2 does not provide any similar functionality. Semantic action code, equivalent to the Beaver counterpart, had to be generated for the following cases:

- Production symbol type matches the type of one symbol in RHS. Return the symbol with matching type.
- Production symbol has type String, and one symbol in RHS has type beaver.Symbol. Cast the value of the symbol to String and return it.
- RHS has one symbol only, and the type does not match the production symbol type. Return and hope for inheritance. (this may fail, but it is hard to check without parsing the .ast file aswell)

6. ExtendJ integration

6.1 Build process

First, unzip extendj-cup2.zip in the root of ExtendJ. Run additional ant task "cup2".

6.2 Running ExtendJ with CUP2

Running ExtendJ is done as usual, but will use CUP2 by default. Example:

```
$ java -jar ExtendJ.jar Test.java
```

6.3 Added files

The following files are added by extracting the patch zip: BeaverToCUP2.jar, full CUP2 source code, ParserFactory, JavaParserBeaver and JavaParserCUP2 classes.

The ParserFactory can construct different parsers (JavaParserBeaver and JavaParserCUP2) that extend JavaParser by calling ParserFactory.instantiate(). Which is returned depends on the value of ParserFactory.which.

6.4 Modifications

The main method source file was modified to call ParserFactory.instantiate() instead of new JavaParser().

The ExtendJ build system was left largely untouched. The only required modification is to add a cup2 task, that calls BeaverToCUP2.jar.

7. Evaluation

7.1 Testing

The ExtendJ Test suite contains over 1000 tests for java version 8. After using BeaverToCUP2 to translate the parser to a CUP2 counterpart, a couple of bugs were uncovered, in ExtendJ (thanks Jesper Öqvist for fixing promptly), in CUP2 (thanks Michael Petter for promising a future fix), and lasty but not least, in BeaverToCUP2.

After ironing out bugs in BeaverToCUP2, the ExtendJ Test suite was run both with an unmodified ExtendJ and a copy with integrated CUP2 support using BeaverToCUP2.

The difference in test results was about 40 tests, all related to:

1. There is something wrong with documentation comments.
2. Error handling in BeaverToCUP2 is not implemented, thus, tests that expect specific Beaver error messages do not work.

Other than that, the same test cases pass for BeaverToCUP2+CUP2 as for Beaver. Over a 1000 tests pass!

7.2 Start up performance

Starting ExtendJ and compiling a single source code file incurs an additional 1s to 5s performance hit with CUP2, compared to Beaver. 1s was observed with an Intel® Core® i5 3750k CPU from 2012 (used for testing). 5s was observed on an old Intel® Pentium® Processor E2180 from 2007. In both cases, Windows 10 was used with Java Development Kit version 8u65.

This long start up time occurs due to CUP2 generating the parser tables on each start up. Serializing the parser tables should - in theory - be possible, but the deserialization is currently broken. Reading parsing tables from a file and deserializing takes orders of magnitude longer than just generating them from scratch. Because of this, serialization is disabled in the code release.

7.3 Parsing speed

Evaluation was primarily conducted by compiling Apache Commons IO 2.4 repeatedly. The performance of the parser was isolated from the rest of ExtendJ by using a software stopwatch approach. The stopwatch was added in ExtendJ and is initialized in paused state. Whenever parsing of a file starts, the stopwatch is resumed (starts counting up the time), and whenever the parsing finishes, the stop watch is paused. When compiling is finished, the time is printed to standard output. The result is that CUP2 is about 20% slower than Beaver, as can be seen in figure 4. However, this is a small part of the total compile time, as can be seen by comparing the parse time to the total time, as seen in figure 3.

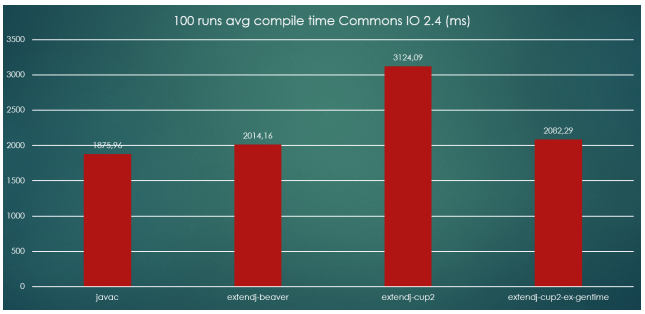

Figure 3. ExtendJ performance using CUP2 vs using Bevaer compiling Apache Commons IO 2.4. Javac is included as a reference. The rightmost result is a best case prediction when the deserialization bug of CUP2 has been resolved. (the time it takes to generate the parser tables has been subtracted)

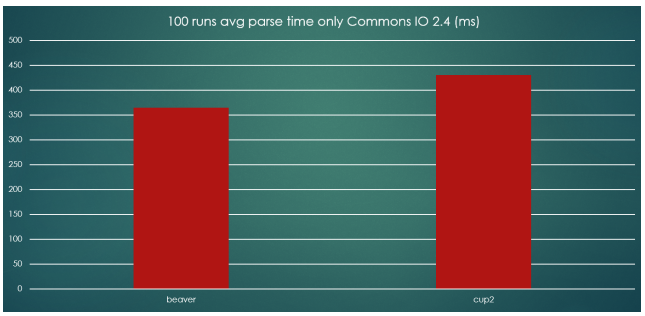

Figure 4. CUP2 vs Beaver parsing Apache Commons IO 2.4. (the lexer time is also included in this, but not start up time or compile time.

7.4 Features, stability etc.

CUP2 is the successor of CUP, a commonly used parser generator. CUP2 is in active development, features are added often and bugs are resolved, based on the commit log. [7] The developer listens to, and appreciates, change requests, suggestions and bug reports.

CUP2 supports all features of the Java language such as:

```
class Java8Parser extends Java7Parser
```

This could open up interesting possibilities, provided grammar can be written so that only productions have to be added in Java8Parser, not removed or changed. Also, debug productions could be added based on java if statements, etc.

CUP2 has proven stable enough for use with ExtendJ to compile Apache Commons IO 2.4. No stability problems have been observed.

CUP2 has working parser generators for several parsing algorithms. Supported algorithms include: LL(k), LR(0), LALR(1) and LR(1). Provided, that the grammar in question supports the parsing algorithm of choice. These can be selected at runtime, by passing the specification class, containing the grammar, to a generator class.

8. Related Work

Testing the original CUP parser generator with JastAdd projects could also be of interest, this has been studied by another group of the course Project in Computer Science – EDAN70. The report is, however, not yet released.

9. Future work and suggestions

To keep modifications to a minimum, some hacky code is present in the file JavaParserCUP2.java, that is part of the ExtendJ patch. Specifically, the scanners return Beaver tokens. These are converted into CUP2 tokens (by putting the Beaver tokens as values inside CUP2 tokens). Thus, semantic action code expecting beaver symbols will still work.

It would be much better if when the scanners were constructed, they were given a parser instance, which has a method constructToken, or similar. constructToken could then construct appropriate tokens based on which parser is being used.

One other suggestion is that scanners could package tokens inside some container object with line and column information. This line and column information could then be used by JastAddParser instead of relying on beaver.Symbol for this purpose. If this was done, swapping parsers would allow getting rid of all beaver.Symbol dependencies.

It could be of interest to investigate how grammar and precedences interacts with Java inheritance in CUP2, as mentioned in section 7.4. However, this is beyond the scope of this project.

10. Conclusion

CUP2 has many exciting features. However, CUP2 is not quite mature enough for production use. The test results suggest it is about 20% slower than Beaver. ExtendJ patched with CUP2 support using BeaverToCUP2 passes as many ExtendJ tests as the unchanged ExtendJ (except for comments, and expected beaver specific error messages).

A translation program, BeaverToCUP2, is provided at:

`https://bitbucket.org/edan70/2015-cup-karl`

This repository also contains a zip file, cup2-extendj.zip, that when extracted into the extendj root folder, adds support for the CUP2 parser. The cup2 support is built with "ant cup2", and the resulting extendj.jar can be run with the flag to use CUP2.

Acknowledgments

The following people have been helpful in the development of this project. Thanks to:

- My supervisor, Christoff Bürger, for advice and feedback.
- Jesper Öqvist, developer of ExtendJ, for fixing ExtendJ bugs uncovered.
- Michael Petter, developer of CUP2, for swift correspondence and bug verification, with promise of a fix.

References

[1] Practical Translators for LR(k) languages. DeRemer, Franklin L. (1969) Cambridge, Mass., M.I.T. Project MAC

[2] LR Parsing: Theory and Practice, Nigel P. Chapman, (1987) Cambridge University Press. p. 86-87

[3] Extended BNF - A generic base standard. Roger S. Scowen. (1993) Software Engineering Standards Symposium

[4] JastAdd - an aspect-oriented compiler construction system. Görel Hedin, Eva Magnusson. (2003) Science of Computer Programming. Volume 47, Issue 1, p. 37-58

[5] The JastAdd system - modular extensible compiler construction. Torbjörn Ekman, Görel Hedin. (2007) Science of Computer Programming. Volume 69, Issues 1-3, p. 14-26

[6] ExtendJ project web page, Hedin et al. (2015) http://jastadd.org/web/extendj/

[7] CUP2 User Manual. Michael Petter. (2015) http://www2.in.tum.de/ petter/cup2/

[8] Beaver Grammar Specification. Alexander Demenchuk. (2012) http://beaver.sourceforge.net/spec.html

[9] Reference Manual for JastAdd2. Hedin et al. (2015) http://jastadd.org/web/documentation/reference-manual.php