EDAN70 Compiler Project

CUP parser generator for JastAdd
January 20, 2016

Felix Åkerlund
D11, Faculty of Engineering (LTH), Lund University
felix.akerlund.978@student.lu.se

Ragnar Mellbin
D11, Faculty of Engineering (LTH), Lund University
ragnar.mellbin.498@student.lu.se

Abstract

We have extended a parser specification preprocessor called JastAddParser to support the Construction of Useful Parsers (CUP) parser generator as a backend in addition to the Beaver parser generator, which previously was the only available option.

In the process of implementing CUP support we also modularized the pre-processor to make it easier to support additional generators in the future. This will make JastAddParser less dependent on the continued support and development of Beaver. It is also interesting to see how a CUP-generated parser performs compared to one built by Beaver.

We encountered several difficulties during the processes, but produced a implementation with partial CUP support.

1. Introduction

Beaver and Construction of Useful Parsers (CUP) are two open source LALR(1) Java parser generators. A parser generator is a program that takes a parser specification as input and produces parser code that implements the specification. The parser code can then be linked with a scanner to build the first stages of a compiler that can transform source code into object code. The scanner reads the source code and divides the character sequence into tokens such as numbers, variable names, operators, etc. The parser's task is then to build a parse tree out of the tokens, which is later used by a code generator to produce the final code.[6, Chapter 1.2]

JastAddParser is a preprocessor for the Beaver parser specification, which adds some features to the Beaver specification. This project aims to add support for generating CUP specifications to JastAddParser. There are two main motivations for doing this. Firstly, to make JastAddParser less dependent on Beaver, which has not seen any updates for a while. Secondly, in the process modularize JastAddParser, to make future modifications of this kind easier to perform.

2. Background

Parser specifications define the production rules of a programming language grammar. Specifications are often written in Backus-Naur Form, a formal notation for context-free grammars.

A context-free grammar is a grammar in which all production rules consist of a nonterminal symbol breaking down into a number of nonterminals and/or terminals (tokens). Context-free means that any such rule can always be applied to a nonterminal symbol regardless of the symbols preceding the nonterminal.

Extended Backus-Naur Form (EBNF)[11] is an extended variant of the Backus-Naur Form, with added support for multi-line rules and shorthand symbols representing repetition, exceptions, etc. An example can be seen in figure 1. The example features four production rules with a nonterminal on the left side of the assignment, and a number of terminals or nonterminals on the right. Terminal strings are enclosed in quotation marks, a vertical bar represents 'or', and commas are for concatenation. Each rule is terminated by a semicolon. Symbols inside square brackets are optional, whereas curly braces indicate repetition.

```
bit                 = "0" | "1" ;
nibble              = bit , bit , bit , bit ;
bits                = bit , { bit } ;
signed-bits         = [ "-" ] , bits ;
```

Figure 1. A small example of a grammar in EBNF.

```
<bit>               = "0" | "1"
<nibble>            = <bit> , <bit> , <bit> , <bit>
<bits>              = <bit> | <bit> , <bits>
<signed-bits>       = <bits> | "-" , <bits>
```

Figure 2. The grammar from figure 1, expressed in BNF.

All grammars expressed in EBNF can be converted to BNF. Figure 2 shows how one can convert an EBNF representation with optionals and repetition to BNF. Nonterminals are enclosed in angle brackets. No semicolons are needed because a rule is always represented by a single line.

2.1 JastAddParser

JastAddParser[4] is a pre-processor for Beaver that allows parser specifications to be split into modules and also uses a slightly different syntax. The specification for JastAddParser is built with Beaver in mind, and thus shares many similarities. Moreover, the implementation of JastAddParser takes advantage of this by, for example, storing parts of the specification that do not need transformation as raw strings internally, and just prints it at the correct location. JastAddParser uses the meta-compilation system JastAdd2 to generate an AST class hierarchy.

A JastAddParser specification can be seen in figure 3. The example features one terminal of type 'TOKEN' and two nonterminals, 'list' and 'list_item'. While being very similar to Beaver specification, it is slightly less verbose. Another noteworthy feature is that JastAddParser supports both definitions with ":=" as well as those with "=". The former replaces previous definitions, while the latter adds on to them.

ExtendJ[8] (previously known as JastAddJ) is an extensible Java compiler built with JastAddParser. It is not as fast as the standard

```
TokenList list =
    list list_item {: list.add(list_item);
        return list; :}
    | list_item {: return new TokenList(list_item); :}
;
ListItem list_item =
    TOKEN {: return new ListItem(TOKEN); :}
;
```

Figure 3. A small JastAddParser specification.

javac compiler, but it can be extended to support custom languages based on Java.

JastAddParser uses a test framework for automated testing. Each test case for this framework has a separate directory that contains the test input data, the test parameters and the expected output of the test case. The test parameters include any flags that should be passed to JastAddParser and the goal test pass for the test case. The test passes are related as in figure 4 and works as follows:

JAP_PASS
Tests with this pass as goal will pass if JastAddParser successfully parses the input file.

JAP_ERR_OUTPUT
Tests with this goal will pass if JastAddParser fails to parse the input file.

JAP_OUTPUT_PASS
Tests with this goal passes if JastAddParser successfully parses the input file, and the output matches the expected output.

EXEC_PASS
Tests with this pass as goal passes if the output from the JAP_OUTPUT_PASS can be used to parse the test data without errors.

EXEC_OUTPUT_PASS
Tests with this goal pass if the output form the JAP_OUTPUT_PASS can be used to parse the test data, and the expected AST is generated.

3. The Beaver parser generator

Beaver[2] is a parser generator for generating LALR(1) parsers from an EBNF grammar specification. LALR stands for Look-Ahead Left-to-right, Rightmost derivation and describes how the parser works to apply the production rules of a language.[7] The number in the parenthesis indicates the number of lookahead tokens, with the most common variant being only one. LALR was developed as an alternative to the LR(1) parser[10], with the advantage of a smaller memory requirement at the expense of some language recognition power. The latest version of Beaver (0.9.11) was released in December 2012.

The beaver specificaion uses '%' before its directives. These are at the beginning of the specification and specify what terminals and non-terminals the parser uses, as well as their type and which production is the goal/start production. This is followed by the actual productions. Something to note about Beaver specifications is that terminal precedences are listed from high to low, and semantic actions feature a return statement.

A sample Beaver parser specification can be seen in figure 5. The parser specification is functionally identical to the JastAddParser one in figure 3. Note the similarities, but also the increased verbosity.

4. CUP parser generator

CUP[3] is another LALR(1) parser generator like Beaver, but uses a specification syntax similar to the one used by the Yet Another Compiler-Compiler[9] (YACC) parser generator, which in turn is similar to BNF. CUP is currently maintained by the Technical University of Munich and is continually being updated, with the latest version released in October 2015.

Unlike the Beaver parser generator, CUP does not support list and optional productions such as '?', '+' and '*'. Also, CUP does not use '%' in front of directives, and a non terminal needs to be declared before it can be set as start/goal production. CUP lists terminal precedences from low to high, unlike Beaver. For the semantic actions, the variable 'RESULT' must always be used.

A sample CUP specification can be seen in figure 6. The specification describes the same language as the one used in figure 5 and figure 3. Note the bigger difference in syntax, and that the start rule needs to be after the declaration of the production used in it.

5. JFlex scanner generator

JFlex[5] is a generator similar to Beaver and CUP, but for generating the scanner component of a compiler. JFlex is designed specifically to work with CUP, but can be paired with other parser generators if needed. Beaver equipped with a scanner integration API to facilitate the use of JFlex and similar tools. JastAddParser uses JFlex for scanner generation.

6. Implementation

To start off our work, we added a flag to enable CUP generation. This required us to rewrite the argument handling code, which originally was not designed to support easy introduction of additional flags. We chose not to use a library for arguments, since we only needed simple flags, and implementing it was a relatively easy task.

JastAddParser uses Apache ANT for building and testing. ANT is tool for automated software building. It is implemented in Java and runs on the Java platform, and uses xml files to describe the build task[1]. Initially, we had some difficulties with running the tests in JastAddParser due to an incorrect path in the build xml file.

JastAddParser, being specifically built around Beaver, only featured a single aspect for printing parser specifications, named PrettyPrint. Since we would be introducing another aspect for printing CUP specifications, we renamed the old aspect to BeaverPrint.

We then copied the contents of BeaverPrint to a new aspect, CUPPrint, to use as a starting point for CUP, and replaced all the Beaver-specific syntax with its CUP counterpart. Examples of this includes reordering rules to put the goal/start and precedence directives last, changing the order of parameters to the non-terminal directives, changing symbols (such as replacing "=" with "::=") and replacing "return x" with "RESULT = x" in productions. The order of terminal precedences had to be reversed, as Beaver lists them from highest to lowest, whereas most other parser generators, including CUP, do the opposite.

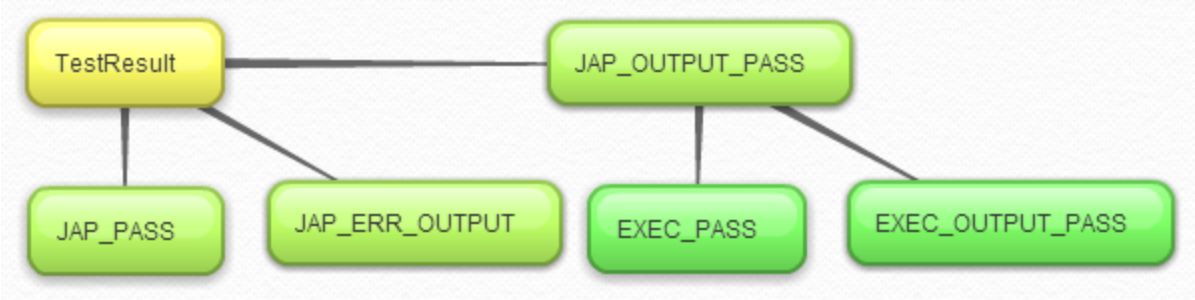

Figure 4. The test passes of the JastAddParser framework

```
%terminals TOKEN;
%goal list;
%typeof list = "TokenList";
%typeof list_item = "ListItem";
list =
    list.list list_item.list_item {: list.add(list_item);
        return list; :}
    | list_item.list_item {: return new TokenList(list_item); :}
;
list_item =
    TOKEN.TOKEN {: return new ListItem(TOKEN); :}
;
```

Figure 5. A small Beaver parser generator specification.

```
terminal TOKEN;
non terminal TokenList list;
non terminal ListItem list_item;
start with list;
list ::=
    list:list list_item:list_item {: list.add(list_item);
        RESULT = list; :}
    | list_item:list_item {: RESULT = new TokenList(list_item); :}
;
list_item ::=
    TOKEN:TOKEN {: RESULT = new ListItem(TOKEN); :}
;
```

Figure 6. A small CUP parser generator specification.

When JastAddParser is executed with the ”–cup” flag, it calls the CUPPrint which writes a CUP parser specification, instead of a Beaver specification, to a file.

We changed the test framework to run all tests twice: once for Beaver, and once for CUP, and we created the corresponding files for checking the test output.

In order to use the generated CUP parser, a scanner is needed. JFlex is a scanner generator natively supported by both Beaver and CUP, which is used by the JastAddParser test framework to do the EXEC_PASS and the EXEC_OUTPUT_PASS. We had troubles writing JFlex files for generating scanners for the CUP files, and were not able to succeed within the time frame of this project. This means that these passes are not supported when testing the CUP functionality. If one of these passes are the goal of the test it stops at the JAP_OUTPUT_PASS for the CUP testing.

7. Evaluation

JastAddParser now has support for generating a CUP specification in a similar manner to how it generates a Beaver specification. The code for generating these two specification have a lot in common, and could surely benefit from some abstractions. This work has been started in the PrintCommons JastAdd aspect.

We extended the test framework to generate and parse CUP specifications as well as the Beaver ones for all test cases. This works for tests that use BNF grammars. We originally planned to compare the performance of the generated parsers, but we were not able to construct suitable JFlex scanners for the cup parsers in the time frame of this project.

During the course of this project we have encountered some difficulties that all consumed a fair amount of time. At first we had trouble running the tests, even before we modified any code. This turned out to be an error in the build file, it did not correctly point to the source directory. Understanding the existing code has

not always been straightforward, and much time has been spent on this.

8. Conclusions

The greatest difficulty of implementing CUP support in JastAddParser has been that Beaver supports EBNF grammars, while CUP does not. This means that there is not a clear translation between a Beaver specification and a CUP one. This does not mean, however, that such a translation is impossible. It is possible to express a EBNF grammar in BNF, but it requires additional productions.

This is something that would require larger changes to the JastAddParser structure, it is currently not built to add productions to itself.

JastAddParser now has a new flag `--cup`, and when given JastAddParser generates valid CUP specification if the JastAddParser specification is in BNF.

Future work could include extending the testing with JFlex scanners for the generated CUP files, and comparisons between the parsed trees constructed by CUP and Beaver respectively. A more advanced task would be to support CUP generation for EBNF JastAddParser specifications.

References

[1] Apache Ant, . URL `http://ant.apache.org/`. [Online; accessed 2016-01-02].

[2] Beaver - a LALR Parser Generator, . URL `http://beaver.sourceforge.net/`. [Online; accessed 2016-01-04].

[3] CUP 0.11b, . URL `http://www2.cs.tum.edu/projects/cup/`. [Online; accessed 2016-01-04].

[4] JastAddParser, . URL `https://bitbucket.org/jastadd/jastaddparser/`. [Online; accessed 2016-01-04].

[5] JFlex, . URL `http://www.jflex.de/`. [Online; accessed 2016-01-04].

[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, August 2006. ISBN 0321486811.

[7] F. L. DeRemer. *Practical translators for LR(k) languages*. PhD thesis, MIT, Cambridge, MA, USA, 1969.

[8] T. Ekman and G. Hedin. The jastadd extensible java compiler. In *Proceedings of the 22nd annual ACM SIGPLAN conference on Object-oriented programming systems and applications*, OOPSLA '07, pages 1–18, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-786-5. . URL `http://doi.acm.org/10.1145/1297027.1297029`.

[9] S. C. Johnson. Yet another compiler-compiler. In *Unix Programmers Manual, Seventh Edition*, 1979.

[10] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1969.

[11] E. S. S. Standard. Ebnf: Iso/iec 14977: 1996 (e). 70, 1996. URL `http://www.cl.cam.ac.uk/mgk25/iso-14977.pdf`.