International Journal of Research in Engineering, Science and Management
Volume-2, Issue-5, May-2019
www.ijresm.com | ISSN (Online): 2581-5792

CUP Parser Generator for JustAdd (EDAN70)

K. Raghavendra[1], R. Mithuna[2], Sini Anna Alex[3]
[1,2]Student, Department of CSE, M. S. Ramaiah Institute of Technology, Bangalore, India
[3]Assistant Professor, Department of CSE, M. S. Ramaiah Institute of Technology, Bangalore, India

***Abstract*: In the method of implementing CUP support we have a tendency to additionally modularized the pre-processor to form it easier to support extra generators within the future. this can build JastAddParser less enthusiastic about the continued support and development of Beaver. it's additionally interesting to examine however a CUP generated programme performs compared to 1 designed by Beaver.**

***Keywords*: JastAddParser, Token, construction kinkly braces.**

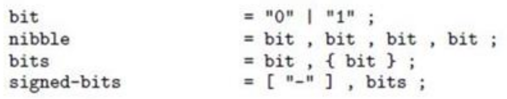
1. Introduction

Beaver and Construction of helpful Parsers (CUP) square measure 2 open supply LALR Java programme generators. A programme generator may be a program that takes a programme specification as input and produces programme code that implements the specification. The programme code will then be connected with a scanner to create the primary stages of a compiler that may rework ASCII text file into code. The scanner reads the ASCII text file and divides the character sequence into tokens like numbers, variable names, operators, etc. The take apartr's task is then to create a parse tree out of the tokens, that is later utilized by a code generator to provide the ultimate code. JastAddParser may be a preprocessor for the Beaver programme specification, that adds some options to the Beaver specification. This project aims to feature support for generating CUP specifications to JastAddParser. There square measure 2 main motivations for doing this. Firstly, to form JastAddParser less enthusiastic about Beaver, that has not seen any updates for a short time. Secondly, within the method modularize JastAddParser, to form future modifications of this type easier to perform.

2. Background

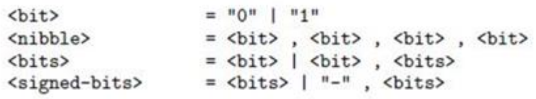
Parser specifications outline the assembly rules of a programing language descriptive linguistics. Specifications square measure usually written in Backus-Naur type, a proper notation for context-free grammars. A context-free descriptive linguistics may be a descriptive linguistics during which all production rules encompass a nonterminal image breaking down into variety of non-terminals and/or terminals (tokens). Context-free implies that any such rule will invariably be applied to a nonterminal image despite the symbols preceding the nonterminal. Extended Backus-Naur type (EBNF) is AN extended variant of the Backus-Naur type, with else support for multi-line rules and shorthand symbols representing repetition, exceptions, etc. AN example is seen in figure one. the instance options four production rules with a nonterminal on the left facet of the assignment, and variety of terminals or non-terminals on the correct. Terminal strings square measure boxed in quotation marks, a vertical bar rep-resents 'or', and commas square measure for concatenation. every rule is terminated by a punctuation. Symbols within sq. brackets square measure nonobligatory, whereas kinky braces indicate repetition.

All grammars expressed in EBNF is born-again to BNF. Figure a pair of shows however one will convert AN EBNF illustration with optional and repetition to BNF. Nonterminal square measure boxed in angle brackets. No semicolons square measure required as a result of a rule is usually delineate by one line.



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

Fig. 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>
```

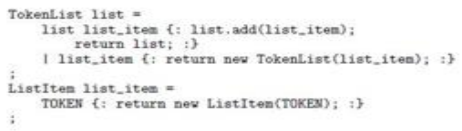
Fig. 2. The grammar from Fig. 1, expressed in BNF

A. *JastAddParser*

JastAddParser may be a pre-processor for Beaver that enables programme specifications to be split into modules and additionally uses a rather different syntax. The specification for JastAddParser is constructed with Beaver in mind, and so shares several similarities. Moreover, the implementation of JastAddParser takes advantage of this by, as an example, storing components of the specification that don't want transformation as raw strings internally, and simply prints it at the right location. JastAddParser uses the meta-compilation system JastAdd2 to get AN AST category hierarchy.

A JastAdd Parser specification is seen in figure three. The ex-ample options one terminal of kind 'TOKEN' and 2 nonterminal, 'list' and 'list item'. whereas being terribly like Beaver specification, it's slightly less windy. Another noteworthy feature is that JastAddParser supports each definitions with":=" in addition as those with "=". the previous replaces previous definitions, whereas the latter adds on to them. Extend J (previously referred to as JastAddJ) is AN extensile Java compiler designed with JastAdd Parser. it's not as quick because the common place

```
TokenList list =
list list_item
| list_item come back new TokenList(list_item); :}
;
ListItem list_item =
TOKEN come back new ListItem(TOKEN); :}
;
```



```
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); :}
;
```

Fig. 3. A small JastAddparser specification

javac compiler, however it is extended to support custom languages supported Java.

JastAddParser uses a take a look at framework for machine-driven testing. every test suit for this framework incorporates a separate directory that contains the take a look at input file, the take a look at parameters and also the expected output of the test suit. The take a look at parameters embody any flags that ought to be passed to JastAddParser and also the goal take a look at pass for the test suit. The take a look at passes square measure connected as in figure four and works as follows:

B. *Jap pass*

Tests with this pass as goal can pass if JastAddParser success-fully parses the computer file.

- *JAP ERR OUTPUT*: Tests with this goal can pass if JastAddParser fails to take apart the computer file.
- *JAP OUTPUT PASS:* Tests with this goal passes if JastAddParser with success parses the computer file, and also the output matches the expected output.
- *EXEC PASS:* Tests with this pass as goal passes if the output from the Japanese OUTPUT PASS is wont to take apart the take a look at information with-out errors.
- *EXEC OUTPUT PASS:* Tests with this goal pass if the output type the Japanese OUTPUT PASS is wont to take apart the take a look at information, and also the expected AST is gen-erated.

C. *CUP programme generator*

CUP is another LALR programme generator like Beaver, however uses a specification syntax like the one utilized by the yet one more Compiler-Compiler (YACC) programme generator, that successively is analogous to BNF. CUP is presently maintained by the Technical University of metropolis and is frequently being updated, with the newest version free in October 2015. Unlike the Beaver programme generator, CUP doesn't support list and nonobligatory productions like'?','+' and '*'. Also, CUP doesn't use'%' before of directives, and a non-terminal must be declared before it is set as start/goal production. CUP lists terminal precedence's from low to high, in contrast to Beaver. For the linguistics actions, the variable 'RESULT' should be used.

A sample CUP specification is seen in figure half-dozen. The specification describes identical language because the one employed in figure five and figure three. Note the larger distinction in syntax, which the beginning rule must intend the declaration of the assembly employed in it.

D. *JFlex scanner generator*

JFlex may be a generator like Beaver and CUP, except for generating the scanner part of a compiler. JFlex is intended specifically to figure with CUP, however is paired with different programme generators if required. Beaver equipped with a scanner integration API to facilitate the utilization of JFlex and similar tools. JastAddParser uses JFlex for scanner generation.

E. *The Beaver programme generator*

Beaver may be a programme generator for generating LALR parsers from AN EBNF descriptive linguistics specification. LALR stands for Look-Ahead Left-to-right, right derivation and describes however the programme works to use the assembly rules of a language. the quantity within the parenthesis indicates the quantity of lookahead to-kens, with the foremost common variant being only 1. LALR was developed as another to the LR(1) parser, with the ad-vantage of a smaller memory demand at the expense of some language recognition power. the newest version of Beaver was free in Dec 2012. The beaver specificaion uses'%' before its directives. These square measure at the start of the specification and specify what terminals and non-terminals the programme uses, in addition as their kind and that pro-duction is that the goal/start production. this is often followed by the particular productions. one thing to notice regarding Beaver specifications is that terminal precedences square measure listed from high to low, and linguistics actions feature a come statement.

A sample Beaver programme specification is seen in figure five. The programme specification is functionally a twin of the JastAddParser one in figure three. Note the similarities, however additionally the accumulated style.

3. Implementation

To start off our work, we have a tendency to else a flag to change CUP generation. This needed U.S. to rewrite the argument handling code, that orig-inally wasn't designed to support straightforward introduction of extra flags. we have a tendency to selected to not use a library for arguments, since we have a tendency to solely required easy flags, and implementing it absolutely was a comparatively straightforward task.

JastAddParser uses Apache hymenopteron for building and testing. hymenopteron is tool for machine-driven code building. it's enforced in Java and runs on the Java platform, and uses xml files to explain the build task. Initially, we have a tendency to had some difficulties with running the tests in JastAddParser thanks to AN incorrect path within the build xml file.

JastAddParser, being specifically designed around Beaver, solely featured one facet for printing programme specifications, named Pretty Print. Since we might be introducing another facet for printing CUP specifications, we have a tendency to renamed the previous facet to Beaver Print.

We then derived the contents of Beaver Print to a replacement facet, CUP Print, to use as a place to begin for CUP, and replaced all the Beaver-specific syntax with its CUP counterpart. samples of this includes rearrangement rules to place the goal/start and precedence directives last, dynamic the order of parameters to the non-terminal directives, dynamic symbols (such as substitution" =" with": =") and substitution" 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 different programme generators, together with CUP, do the other.

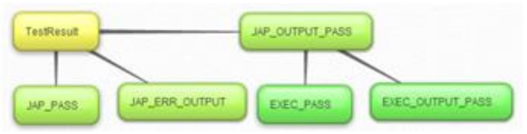


Fig. 4. The test passes of the JastAddParser framework

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

terminal TOKEN;
non terminal TokenList list;
non terminal ListItem list_item;
start with list;
list ::=
list:list list_item:list_item
| list_item:list_item
;
list_item ::=
TOKEN:TOKEN
;
```

When JastAddParser is dead with the "–cup" flag, it calls the CUPPrint that writes a CUP programme specification, rather than a Beaver specification, to a file.We modified the take a look at framework to run all tests twice: once for Beaver, and once for CUP, and that we created the corresponding files for checking the take a look at output. In order to use the generated CUP programme, a scanner is required. JFlex may be a scanner generator natively supported by each Beaver and CUP, that is employed by the JastAddParser take a look at framework to try and do the executive department PASS and also the executive department OUTPUT PASS. we have a tendency to had troubles writing JFlex files for generating scanners for the CUP files, and weren't able to succeed at intervals the timeframe of this project. this implies that these passes aren't supported once testing the CUP practicality. If one in every of these passes square measure the goal of the take a look at it stops at the Japanese OUTPUT PASS for the CUP testing.

4. Evaluation

JastAddParser currently has support for generating a CUP specification in an exceedingly similar manner to however it generates a Beaver specification. The code for generating these 2 specification have loads in common, and will sure take pleasure in some abstractions. This work has been started within the PrintCommons JastAdd facet. We extended the take a look at framework to get and take apart CUP specifications in addition because the Beaver ones for all take a look at cases. This works for tests that use BNF grammars. we have a tendency to originally planned to match the performance of the generated parsers, however we have a tendency to weren't able to construct appropriate JFlex scanners for the cup parsers within the timeframe of this project.

During the course of this project we've got encountered some difficulties that everyone consumed a good quantity of your time. initially we have a tendency to had hassle running the tests, even before we have a tendency to change any code. This clothed to be a slip within the build file, it failed to properly purpose to the supply directory. Understanding the present code has not invariably been simple, and far time has been spent on this.

5. Conclusion

The greatest issue of implementing CUP support in JastAddParser has been that Beaver supports EBNF grammars, while CUP does not. this implies that there's not a transparent translation between a Beaver specification and a CUP one. This doesn't mean, how-ever, that such a translation is not possible. it's attainable to precise a EBNF descriptive linguistics in BNF, however it needs extra productions. This is one thing that will need larger changes to theJastAddParser structure, it's presently not designed to feature productionsto itself. JastAddParser currently incorporates a new flag cup, and once given JastAddParser generates valid CUP specification if the JastAddParser specification is in BNF. Future work might embody extending the testing with JFlex scanners for the generated CUP files, and comparisons between the parsed trees created by CUP and Beaver severally. A more advanced task would be to support CUP generation for EBNF JastAddParser specifications.

References

[1] Apache Ant, http://ant.apache.org/.
[2] Beaver-A LALR programme Generator, http://beaver. sourceforge.net/
[3] CUP 0.11b, http://www2.cs.tum.edu/projects/cup/
[4] JastAddParser, https://bitbucket.org/jastadd/ jastaddparser/.
[5] JFlex, http://www.jflex.de/
[6] A. V. Aho, M. S. Lam, R. Sethi, and J. D. Ullman. Compilers: Principles, Techniques, and Tools (2nd Edition). Addison Wesley, August 2006.
[7] F. L. DeRemer. sensible translators for LR(k) languages. Ph. D. thesis, MIT, Cambridge, MA, USA, 1969.
[8] T. Ekman and G. Hedin. The jastadd extensile java compiler. In Proceedings of the twenty second annual ACM SIGPLAN conference on Object-oriented programming systems and applications, OOPSLA ’07, pages 1–18, New York, NY, USA, 2007. ACM.