"Linux Gazette...*making Linux just a little more fun!*"

Compiler Construction Tools, Part III

Creating A Calculator Using JFlex And CUP

by Christopher Lopes, student at Eastern Washington University
April 26, 1999

This is the third part of a series begun in the April 1999 issue of Linux Gazette.
[see: Compiler Construction Tools, Part I]. Part II, giving detailed installation instructions for JFlex and CUP appears in this same issue.

This particular example is a modified version of the calculator example shown in the CUP manual. In particular, the companion JFlex specification file is included. Further, that file and the associated CUP specification file are commented extensively. The calculator example is the traditional first example to display the use of tools in the lex/yacc family. We are currently working on a project that would comprise a deeper example - an initialization language for a fuzzy logic engine to be used for decision making applications. If there is sufficient interest expressed in that longer term project, we will prepare an article for this or another venue.

Using JFlex

The purpose of JFlex in this project is to build a lexical analyzer for our calculator. This lexical analyzer, or scanner, will check the input for our calculator and make sure all character groupings are valid.

The lexical specification file for JFlex is broken up into three sections. Each of these sections are separated by *%%*.

User Code Section
%%
Options and Declarations Section
%%
Lexical Rules Section

User Code Section

Everything in this section will be copied into the generated lexer class before the class declaration. In this section one typically finds *package* and *import* statements. Our lexical specification for this section imports two classes, *sym* and *java_cup.runtime.**, and looks like the following.

```
import java_cup.runtime.*;
import sym;
```

In our example, the sym class is generated (along with the parser) by CUP.

Options and Declarations Section

This section contains options, lexical states, and macro declarations. Setting options will include extra code that will be included inside the generated scanner class. Options must begin a line and start with a *%*. There are many options that can be included. To obtain a list of options that can be included consult the manual that comes with JFlex. The options used in our lexical specification are below.

```
%class Lexer
%line
%column
%cup
```

The first option, class Lexer, tells JFlex to name the generated class *Lexer* and to write the code to a file called *Lexer.java*. The line option turns on line counting letting you access the current line number of the input with the variable *yyline*. The column option does a similar thing except it is for the current column number with the variable *yycolumn*. The last option, cup, puts JFlex into a mode that will make it compatible with a CUP generated parser, which is what we are using.

You next can declare member variables and functions for use inside the scanner. The code that can be added is Java code and is placed between *%{* and *%}*. It will be copied into the generated lexer class source. For our lexical specification two member functions will be declared. These functions create *java_cup.runtime.Symbol* objects. The first one just contains position information of the current token. The second contains this information as well as the value of the token. A link to this declaration is below.

Declarations

The last part of this section contains macro declarations. Macros are used as abbreviations for regular expressions. A macro declaration consists of a macro identifier followed by = and then the regular expression that it represents. A link to the macro declarations used in our lexical specification follows. A link is also supplied below that contains a list of what can be used to create a regular expression and what each item in that list means.

Macro Declarations

List of what can be used in Creating Regular Expressions

Lexical Rules Section

The last section of the lexical specification contains the regular expressions and actions that will be executed when the scanner matches the associated regular expression. The scanner will activate the regular expression that has the longest match. So if there existed two regular expressions "to" and "too" the scanner would match "too" since it is the longest. If two regular expressions are identical and have the same length then the scanner will match the regular expression that is listed first in the specification. If the scanner read in the string "to" and was looking for a regular expression to match what it read in it could activate either regular expression listed below. The second regular expression is possible since it contains a character class which allows for the string "to" to be matched. The scanner would pick the first regular expression in the list below since it was listed first.

```
"to"
[a-z]*
```

Actions can then be attached to each regular expression that the scanner can activate when it matches that regular expression. The actions for each regular expression are just Java code fragments that you can write. Actions that you might want to use could be printing something out or returning the token that the scanner just found to the parser. Example code that prints the token found by the scanner and returns it to the parser could be done as in the following.

```
"+"     { System.out.print(" + ");  return symbol(sym.PLUS); }
"-"     { System.out.print(" - ");  return symbol(sym.MINUS); }
"*"     { System.out.print(" * ");  return symbol(sym.TIMES); }
"/"     { System.out.print(" / ");  return symbol(sym.DIVIDE); }
```

JFlex allows the programmer to refine the specification by defining special *lexical states* used as start conditions. YYINITIAL is

a predefined lexical state and is the state in which the lexer initiates scanning input. It's the only one we'll use. Consequently, all our regular expressions will be recognized starting from that lexical state. However, one can define other such states which will essentially constitute the start of a new branch of the state machine. In the example below, lexical state <STRING> is reached by a transition from YYINITIAL. Regular expressions defined in that state section <STRING> will only be recognized in that branch.

```
<YYINITIAL> {
      \"                    { string.setLength(0); yybegin(STRING); }
      "="                   { return symbol(sym.EQ); }
      "=="                   { return symbol(sym.EQEQ); }
      "+"                   { return symbol(sym.PLUS); }
}

<STRING> {
      \"                     { yybegin(YYINITIAL);
                               return symbol(sym.STRINGLITERAL,
                               string.toString()); }
      [^\n\r\"\]+            { string.append( yytext() ); }
}
```

In the above code the scanner will begin in the state YYINITIAL. When it matches the regular expression \", which just means it is found a double quote, it will change the scanner to the STRING state. Now the only regular expressions that can be matched are the regular expressions listed for that state. So the scanner will stay in this branch until it matches another double quote - whereupon it will return to the YYINITIAL state again. Again, for our calculator we never employ such starting conditions other than the original YYINITIAL state. A link to the lexical rules we used are below.

Link to Lexical Rules

Link to the JFlex file lcalc.flex . This is the lexical specification used for our calculator. In it there are lots of comments explaining what is happening. It can be copied and both the CUP and Main files which are also supplied in this article can be copied so you can run this example project. Instructions on how to prepare each file and run the calculator are included. Jdk, JFlex, and CUP are needed to do this and can be downloaded for free at the web sites listed in this article.

For more information on JFlex consult the JFlex manual that is available when you download JFlex at the web site that is listed in this article.

Back to Top

Using CUP

The purpose of CUP in this project is to build a syntactic analyzer for our calculator. This syntactic analyzer, or parser, will check the input for our calculator and make sure it is syntactically correct.
That is to say that the statements in the input are arranged in a valid order according to our syntax specification.

The specification syntax for a CUP file is broken up into four sections.

1. Preliminary Declarations
2. Declarations of Terminals and Non Terminals
3. Precedence and Associativity of Terminals
4. Grammar

Preliminary Declarations

This section provides preliminary and miscellaneous declarations to specify how the parser is to be generated and supply parts of the runtime code. This section is optional and doesn't need to be included in a CUP specification file. For our calculator we

will have three items in this section. The first item will be an import declaration. We will import the class *java_cup.runtime.**.

```
import java_cup.runtime.*;
```

The next item we will add is parser code. The parser code will be placed directly into the generated parser class definition. It begins with *parser code {:* and ends with *:}* with all coded inserted in between. In the parser code we will change two methods. We will change the report_error and report_fatal_error method. We will modify these methods so the if a syntactic error or a fatal error occurs in the input then the error message that will be printed out will contain the line and column number in the input of where the error occurred. This extra information in error messages could prove very helpful when determining errors in the input.

Link to Parse Code

The last item we will add in this section indicates how the parser should ask for the next token from the scanner and has the form *scan with {: ... :}*. We will use this to tell the parser to call the scanner we created with JFlex.

```
scan with {: return lexer.yylex(); :};
```

Declarations of Terminals and Non Terminals

This section contains the symbol list and contains declarations that are responsible for naming and supplying a type for each terminal and non terminal. This section is required in a CUP specification. Terminals are declare with the syntax *terminal classname name1, name2, ...;*. Classname is the type of the object, such as Integer. If no classname is given then the terminal has no content for the lexer to pass up to the parser.. After the classname the name of the terminals are listed that you want to declare of that type as in the following.

```
terminal PLUS, MINUS, TIMES, DIVIDE, SEMI;
terminal Integer NUMBER;
```

Note that only NUMBER has an accompanying *classname*. In our example, it is the only terminal that carries content. For example, when the lexer recognizes a PLUS, it passes the associated code to the parser; but when it recognizes a NUMBER it not only passes the associated code for NUMBER, but also its value within the type wrapper class, *Integer*.

Non terminals are declared in the same manner. The only difference is the beginning of the declaration reflects that it is a non terminal instead of a terminal as in the following.

```
non terminal expr_list, expr_part;
non terminal Integer expr;
```

Precedence and Associativity of Terminals

This section specifies the precedence and associativity of terminals, it is an optional section that doesn't have to be included. This section can be used when parsing ambiguous terminals. Instead of using this section you could structure the grammar so that it is not ambiguous. For instance TIMES should have a higher precedence then PLUS. When the parser runs into a statement such as 5+4*3 it doesn't know whether the expression needs to be calculated as 5+(4*3) or (5+4)*3. To eliminate this ambiguity using this section you would declare the precedence as below. The highest precedence starts at the bottom of the list and the precedence gets less going up. The word left means that the associativity of the terminals at that precedence level goes from left to right.

```
precedence left PLUS, MINUS;
precedence left TIMES, DIVIDE;
```

To structure a grammar to eliminate the ambiguity you would create a structure like the one below. This structure eliminates the ambiguity because TIMES is further down in the grammar than PLUS. This will result in TIMES being applied before PLUS as you go back up the grammar.

Example of Grammar Structure

Grammar

The last section in the specification syntax contains the grammar for the parser. Each production in the grammar has a non terminal left hand side followed by *::=*, which is then followed by zero or more actions, terminals, or non terminals, and then followed by a semicolon. Each symbol on the right hand side can be labeled with a name, which can be used to carry content (e.g. a value) up the parse tree. A label name is given by a colon after the symbol name, then the name of the label as in the following where e1 and e2 are labels for expr. The left hand side automatically is assigned the label RESULT. An example using the label RESULT appears latter in this section.

```
expr ::= expr:e1 PLUS expr:e2
```

The label names must be unique in the production. If there exists several productions for the same non terminal they can be declared together and separated by |. The semicolon then needs to be placed at the end of the last production as in the following.

```
expr ::= expr PLUS expr
         | expr MINUS expr
         ;
```

Actions can also be inserted into the production. The action is just Java code and will be executed when the production has been recognized. Action is placed between the delimiters *{:* and *:}*. An example of part of a grammar with these options is below. A link to a file with the specification syntax for CUP named *ycalc.cup* also follows and the grammar in it can be studied.

```
expr     ::= factor:f PLUS expr:e
    {: RESULT = new Integer(f.intValue() + e.intValue()); :}
    |
    factor:f MINUS expr:e
    {: RESULT = new Integer(f.intValue() - e.intValue()); :}
    |
    factor:f
    {: RESULT = new Integer(f.intValue()); :}
    ;
```

Link to the CUP file ycalc.cup. This is the specification syntax used for our calculator. In it there are lots of comments explaining what is happening. It can be copied and both the JFlex and Main files which are also supplied in this article can be copied so you can run this example project. Instructions on how to prepare each file and run the calculator are included. Jdk, JFlex, and CUP are needed to do this and can be downloaded for free at the web sites listed in this article.

For more information on CUP consult the CUP manual that is available at the web site listed in this article.

Back to Top

Main for our Calculator

There is more than one way you can write the main for our project. One way expects the user to enter input as the program runs. The other way requires that you give it the name of an input file when you start up the program. The main described here uses the second way mentioned to retrieve input. The first thing we do is import three classes that we will use. The first class is for our parser, the next is the java_cup.runtime.Symbol class, and the last is the java.io.*; class. We then declare are class *Main*. In it we will call the parser to begin the syntactic analysis of the input file. The parser will then call the scanner, that will lexically analyze the input, when the parser needs the next token in the input file. The class *Main* contains two items. It first sets the variable *do_debug_parse* to false. We then define a method called *main*. We pass into *main* an array of strings which contains the parameters passed on the command line when the program was started. So in our case the first element of the string will contain the name of the text file we passed in when we started the program. The method then goes into a *try* block which is what actually calls the parser. The *try* block means that whatever is in the *try* block will attempted. If something fails, the program will exit that block. The first line in the *try* block creates a new parser object. The parser object invokes a new

Lexer object. The new Lexer object will use the string passed into *main* for its input when it is created. The second line will then start the parser. The code for the above follows.

```
try {
  parser p = new parser(new Lexer(new FileReader(argv[0])));
  Object result = p.parse().value;
}
```

Following the *try* block is a c*atch* block. The purpose of the *catch* block is to clean up an errors that happened in the *try* block. The *catch* block will take the exception, the reason why we were kicked out of the *try* block, and do whatever is needed to clean things up before the program exits. We don't do anything in the contents of our *catch* block. After the *catch* block we have the method *finally*. This method closes everything out. We don't do anything in this method either. The code for the *catch* block and method *finally* are below.

```
  catch (Exception e) {
      /* do cleanup here -- possibly rethrow e */
      } finally {
          /* do close out here */
        }
      }
```

That completes the contents of the method *main* and the class *Main*. We now have created a simple calculator using JFlex as our lexical analyzer and CUP as our syntactical analyzer.

Link to the Java file Main.java . This is the main used for our calculator. In it there are comments explaining what is happening. It can be copied and both the JFlex and CUP files which are also supplied in this article can be copied so you can run this example project. Instructions on how to prepare each file and run the calculator are included. Jdk, JFlex, and CUP are needed to do this and can be downloaded for free at the web sites listed in this article.

Back to Top

Compiling the Calculator

To setup the files to run the calculator you first need to use JFlex on the lexical specification file lcalc.flex. This will produce the file Lexer.java. The next step is to setup the CUP file ycalc.cup. Afterwards you compile the Lexer.java file that was created by JFlex. You finish the process by finally compiling the Main.java file. To do the above you would enter the following at the command line.

```
> jflex lcalc.flex
> java java_cup.Main < ycalc.cup
> javac Lexer.java
> javac Main.java
```

Then to run the calculator you would enter the following at the command line. The file test.txt is the input file for the calculator that will be scanned and parsed.

```
> java Main test.txt
```

Back to Top

Sample Input and Output

A sample input file could look like the following.

```
2+4;
```

```
5*(6-3)+1;
6/3*5+20;
4*76/31;
```

and so on. The output for the following input should then appear as follows.

```
2 + 4 = 6
5 * ( 6 - 3 ) + 1 = 16
6 / 3 * 5 + 20 = 30
4 * 76 / 31 = 9
```

Back to Top

Previous ``Compiler Construction Tools'' Columns

Compiler Construction Tools Part I, April 1998
Compiler Construction Tools Part II, May 1998

Published in Issue 41 of *Linux Gazette*, May 1999