The Application of JavaCC to Develop a C/C++ Preprocessor

Giancarlo Succi, Raymond W. Wong

Department of Electrical and Computer Engineering, University of Calgary
2500 University Dr. NW. Calgary, Alberta, Canada, T2N 1N4

Email: Giancarlo.Succi@acm.org, wongr@enel.ucalgary.ca

Abstract

The commonly available software metrics-extraction tools for C/C++ depend on commercial preprocessors to preprocess the source file before being input into the analyzers. The following paper introduces a Java compiler generator called JavaCC and the application of the generator to develop a Java-based preprocessor for C/C++. Some technical features to the development of preprocessor are also mentioned, such as (1) handling of rescanning in preprocessing with LL(k) parsers, (2) managing conditional compiling with the state-based features of the lexical analyzer in JavaCC, and (3) resolving macro replacement and expansion with parsers.

1 Introduction

C and C++ are widely used for building applications, ranging from simple programs to large operating systems. By extracting software metrics from previously completed projects, software engineers gain more understanding of software behaviour and developers' programming practices.

Currently, a few software tools have been developed to extract software metrics from C/C++ source codes, such as QualGen [1] and Panorama/OO-SQA [2]. However, these extractors depend heavily on external preprocessors to generate the required input files. The dependency on external preprocessors creates ambiguities, because preprocessors from compiler vendors insert different sets of keywords to enhance the basic source code. For example, Microsoft's preprocessor adds the keyword __inline to suggest optimization to the compiler. With all these specific keywords inserted in the input files, the metrics extracted is not guaranteed to be the same. To overcome the above difficulties, a basic C/C++ preprocessor without any specific keyword added into the source code is desired.

Sun MicroSystems has developed many supporting packages to extend the power of the Java programming language. In this paper, we refer to JavaCC (Java Compiler Compiler) [3] which is used to generate parsers in Java. With JavaCC, we developed a universal C/C++ preprocessor which runs on any computer platform with a Java Virtual Machine.

The format of this paper is as follows. In Section 2, a general introduction to JavaCC is provided. Section 3 describes the general architecture of the preprocessor. The use of lexical and syntax analyzers within JavaCC to develop a preprocessor is described in Section 4 and 5. The conclusion addresses some related problems and future modifications to the preprocessor.

2 How JavaCC works

JavaCC is a compiler generator that accepts language specifications in BNF-like format as input. The generated parser contains the core components of corresponding compiler of the specified language, which includes a lexical analyzer and a syntax analyzer. Figure 1 shows the overall structure of a parser generated by JavaCC.

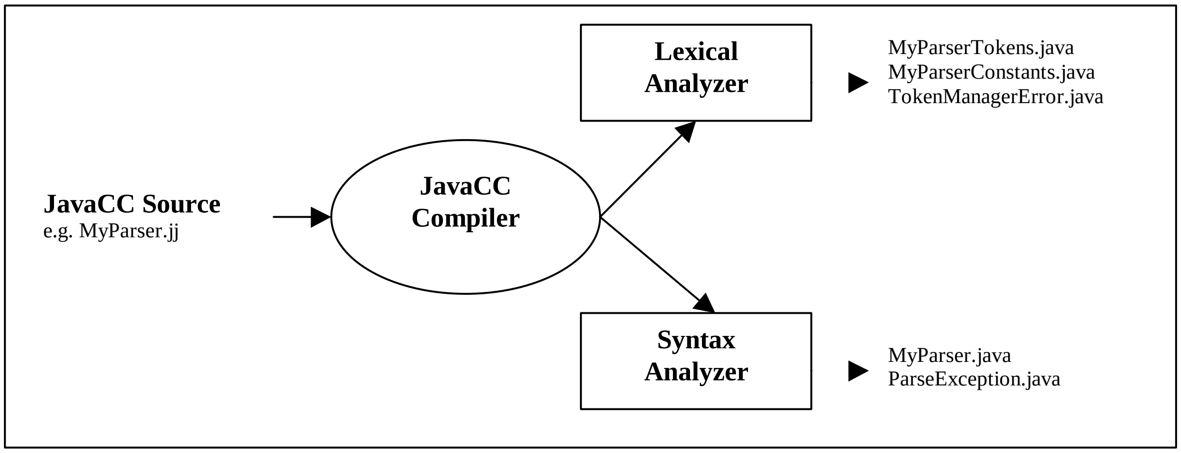

Figure 1. Generation of JavaCC Parser

2.1 Lexical Analysis with JavaCC

The lexical analyzer in JavaCC is called “TokenManager”. The TokenManager is used to group characters from an input stream into Tokens according to specific rules. Each specified rule in TokenManager is associated with an expression kind [4]:

`SKIP:` Throw away the matched expression
`MORE:` Continue taking the next matched expression to build up a longer expression
`TOKEN:` Creates a token using the matched expression and send it to the syntax analyzer
`SPECIAL_TOKEN:` Creates a token with the match expression and optionally send to the syntax analyzer, which is different from *`TOKEN`*.

The TokenManager is a state machine that moves between different lexical states to classify tokens. Figure 2 illustrates a sample state machine feature of the lexical analyzer. When the analyzer starts, it is in the *`Default State`* which waits for inputs. If the input is a character "A", it moves to *`State A`*. Then from *`State A`*, if the input is character "A", it stays in the same state. However, when the input is character "B" or "C", the system moves to the corresponding states. If the state machine is facing an unspecified situation, such as hitting a character "E" in *`State C`*, it generates lexical errors. The following code segment implements a portion of the TokenManager for the described lexical analyzer in JavaCC:

```
SPECIAL_TOKEN {
      <A: ("A") > : State_A //Switch to State A if input character is “A”
}

<State_A> SPECIAL_TOKEN : {
      <A_Again: ("A")* >    //Stay in State A if input character is “A”
|     <B: ("B") > : State_B //Switch to State B if input character is “B”
}

<State_B> SPECIAL_TOKEN : {
      <C: ("C") > : State_C //Switch to State C if input character is “C”
|     <D: ("D") > : Default //Switch to DEFAULT state if input character is “D”
}

<State_C> SPECIAL_TOKEN : {
      <C_Again: ("C")* >    //Stay in State C if input character is “C”
|     <D: ("D") > : Default //Switch to DEFAULT state if input character is “D”
}
```

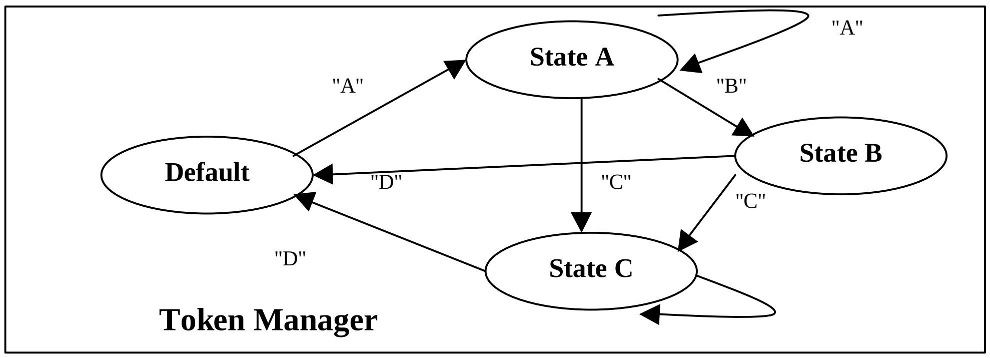

Figure 2. State Machine Features in Lexical Analyzer

2.2 Syntax Analysis with JavaCC

The syntax analyzer in JavaCC is a recursive-descent LL(k) parser. This type of parser uses k number of lookahead tokens to generate a set of mutually exclusive productions, which recognize the language being parsed by the parser. By default, JavaCC's syntax analyzer sets k to 1, but developers can override the number of lookahead tokens to any arbitrary number to match productions correctly.

LL(k) parsers allow only right recursion in the production [5]. Consider a commonly used syntax for Expression in C/C++ that call itself recursively:

```
Expression::= Expression operators Expression
             |"(" Expression ")"
             |<IDENTIFIER>
```

The left recursive production is not allowed in LL parsers, the syntax must be reconstructed so that the parser can recognize the production correctly with limited amount of lookahead tokens. Therefore, sequences of tokens that generate mutually exclusive situations in the production should be placed in the beginning of each possible case. In the example of Expression production, we uses the character *"("* and the token *<IDENTIFIER>* to separate the production into two mutually exclusive situations. In this way, the Expression production is structured as shown:

```
Expression::= <IDENTIFIER> (operator Expression)+
             |"(" Expression ")"
```

This approach always requires in any LL(k) parsers and it is often no trivial to implement the requiring changes in the structure of the grammar from the targeted language's BNF specification. Therefore the conversion of the syntax into right recursion should be considered with cares.

3 General Architecture of C/C++ Preprocessor

The directives defined in the preprocessor in C/C++ specify actions for (1) macro substitution, (2) conditional compilation and (3) source files inclusion [6]. The token manager and syntax analyzer in the preprocessing language shares different responsibilities. The token manager first identifies preprocessing directives and segments the required blocks of code for preprocessing. Then the syntax analyzer processes the code by applying the specified directives.

The major problem faced in the preprocessing is rescanning. Consider the following example:

```
#define hello       helloWorld
```

```
#define helloWorld  MyExample
```

Rescanning is a recursive step in preprocessing because the process continues until all defined macros are replaced. According to the example, a call to `hello` first generates `hellowWorld` as an intermediate result. Then the result is rescanned and replaced to `MyExample`. Because there are no further definition of `MyExample` is found, the rescanning process terminates.

The rescanning process is done by invoking the preprocessor recursivly. The code segment that requires rescan is input into another instance of the preprocessor. After processing the segment, the preprocessed result is returned into to the main preprocessor for further operations. The rescanning process is a supporting operation in preprocessing.

The differentiation of main and support processes separates the preprocessor into different functionality. The functionality of the main preprocessor parser is responsibility for managing the overall operations, including file inclusion and rescanning. It remembers the information of the preprocessing, so that it resumes or advances to the correct position after invoking other supporting processes. The preprocessor dispatches other operations to three supporting parsers.
1. The conditional expression replacer to replace macros expressed in a conditional statement
2. The conditional expression evaluator to evaluate conditional expressions
3. The macro expander to handle macro expansion

Table 1 lists the detail responsibilities of the various parsers used for preprocessing.

Name of Component	Functionality
Main Preprocessor Parser	**File Inclusion** • Locates and loads files from storage
	Handle define directive definition • Reads in simple and function-like Macro definitions
	Segment Preprocessing Blocks • Calls conditional expression replacer and conditional expression evaluator to evaluate which conditional compilation block should be included in the preprocessing • Calls macro expander for complex macro expansions
	Perform "Standard" Preprocessing • Handles string concatenation • Handles simple macro replacement • Performs rescanning • Generates preprocessed file
Conditional Expression Replacer	**Replace arguments in the conditional statements defined by macros**
Conditional Expression Evaluator	**Evaluate of condition statements**
Macro Expander	**Handle Macro Expansion** • Expands macros with parameters • Handles # and ## operators

Table 1. The Core Components of the C/C++ Preprocessor

4 Other Issues Faced in Token Manager

The Token Manager in the Preprocessor directly reads in the characters from input streams and recognizes tokens needed by the parser. The preprocessor separates the input into two different sets of tokens: (1) Preprocessing Directives and (2) Source Body.

The preprocessing directives are specific keywords defined by the C/C++ language specification. These directives define a set of actions to be performed by the preprocessor. They include:

- `#include` to include other files for preprocessing
- `#define` to associate a macro or replacement mechanism to a specific set of identifiers
- `#if, #elif, #endif` to establish conditional compilation units at compile-time

Each directive has its own syntax to define the associated action. For example, the `#include` directive uses the syntax `"file-name"` and `<file-name>` to differentiate the location of the include file. This situation is handled an `INCLUDE_STATEMENT` state in the TokenManager.

```
SPECIAL_TOKEN {
       <POUND_INCLUDE: "#" "include"> : INCLUDE_STATEMENT
}
```

The `INCLUDE_STATEMENT` has two different cases, for handling both local or global file inclusions. After identifying the include file name, the Token Manager changes back to the `DEFAULT` state to match other tokens.

When considering the selection of compilation units in the preprocessing, the state-machine capability of the token manager in JavaCC simplifies the development. A more complex example is the evaluation of conditional blocks in preprocessing. After the evaluation of the conditional stations in `#if` or `#elif` statements, the Token Manager may advance to two different states depending on the result:

1. If the evaluated result is `TRUE`, the preprocessor should include the current statement block for preprocessing – `DEFAULT State`
2. If the evaluated result is `FALSE`, the preprocessor should discard the current block and move to the next preprocessing block – `IGNORE_IF State`

For simplicity, the following example illustrates how JavaCC is capable of handling the `#if` situation first by moving the state of Token Manager to `IF_EXPRESSION`:

```
SPECIAL_TOKEN {
       <IF: "#" "if"> : IF_EXPRESSION
}
```

In the `IF_EXPRESSION` state, the preprocessor identifies the conditional expression from the source file and sends the identified block into the conditional expression replacer as the argument. After processing the input argument, the result passes into the conditional expression evaluator for evaluation. The evaluated result is sent back to the preprocessor and it determines the next state of the TokenManager depending on the situation.

After processing the directives of preprocessing in the Token Manager, the built tokens which form the source body of the input code are passed to the syntax analyzer for string concatenation, macro replacement, and macro expansion.

5 Syntax Analyzer and Preprocessing

With most of the preprocessing work done by the TokenManager, the major focus of the syntax analyzer is to process all the predefined macros in `#define` directive by (1) replacing or (2) expanding them accordingly. Because of the need of checking each token in the source body for replacement, the syntax analyzer reads in each non-numeric token and checks against the lookup table that stores the predefined macros. The searching in the lookup table is time consuming and the large lookup table occupies a significant amount of resources in the Java Virtual Machine.

The direct replacement of macro is handled in the main preprocessor. The code for such operation follows:

```
t=<IDENTIFIER>  // Read in a non-numerical token
{
      //Check if the token is a predefined macro
      if (DefineSet.isDefined(t.image)) {
            DefineHandler dh = DefineSet.getDefinition(t.image);

            //Check to see if the token is a Simple Macro
            if (dh.isSimpleMacro()) {
                  //Replace the token with the predefined macro
                  resultingFile.print(" "+dh.replace());
            }
      } else { //it is not a definition nor a macro
          resultingFile.print(" "+t.image);
      }
}
```

The replacement algorithm checks each identifier against the lookup table. If the matched identifier requires direct replacement the macro will be replaced with the suggestion.

However, the macro definition with parameters, or function-like macro definition requires a more complex algorithm for expansion. For example, given the macro definitions:

```
#define addition(A,B)     A + B
```

The call `addition(Parameter1, Parameter2)` yields `Parameter1 + Parameter2`. The problem of macro definition is further complicated by the introduction of `#` and `##` operators. The `#` operator will be replaced in the expansion by a string literal corresponding to the argument, while the `##` operator concatenates the argument and the adjacent token together. Consider the following example (for more complex cases, see [6]):

```
#define message(m)  Print(m #)
#define hello(a)    Hello ## a
```

The call to `message(test)` and `hello(World)` gives the following results respectively: `Print("test")` and `HelloWorld.`

Generally, the body of the complex macro consists two types of elements:
1. Bound Element – The constant part of the macro that does not need to be processed
2. UnBound Element – The argument part of the macro that needs to be replaced by the passed-in argument when called

The introduction of these two elements to describe the macro body breaks down the body of the directive `#define addition(A,B)` into three components as follows:
1. UnBound Element – `A`
2. Bound Element – `+`
3. UnBound Element – `B`

When there is a call to addition with *`Parameter1`* and *`Parameter2`* as arguments, the UnBound Elements used to describing the body will be replaced to Bound Elements with the actual contains. As we referred to the previous example, the body of the call *`addition(Parameter1,Parameter2)`* results in:

1. Bound Element – *`Parameter1`*
2. Bound Element – `+`
3. Bound Element – *`Parameter2`*

The described expansion process seems to be a trivial task, but it is actually a complex process. So far, we have only considered the case of one macro body with ordered parameters. However, according to the preprocessing language, it is possible to have a macro body that calls other macros either requiring expansion or replacement. The recursive property of macro expansion greatly complicates the macro expansion process.

To handle the macro expansion problem, the support parser macro expander is called. It uses the name of the macro, and the parameter list as input to the parser. Then, the macro expansion proceeds in the following steps:

1. Look up the corresponding definition body for the specified macro
2. Break down the body into a set of Bound and UnBound elements
3. Replace the UnBound elements with actual parameters

The code segment used to handle macro expansion is included as an example:

```
//Initial Macro Expansion
MacroExpander marExpander = new MacroExpabder(macroName,parameters);
//Start Macro Expansion
marExpander.ReduceArguments();
//Obtain the Processed Set of Bound Elements from MacroExpander
Vector setOfElemnts=marExpander.bodyComponents();
```

With this process, *#* and *##* are handled when building up the bound element set. After the first attempt to expand the macro, the final step for macro expansion is to rescan the expanded macro for further macro replacements and expansions.

6 Future Research and Conclusion

Although the first version of the preprocessor is completed, the processing speed of the application is extremely slow when compared with other commercial preprocessors. In the beginning of 1999, the same company that develops JavaCC has announced another parser generator with more enhanced features called MetaMata Parse [7]. The new product claims to have better speed performance, and stronger support for multiple parsers.

The current stage of the research is to evaluate the gain and lose between JavaCC and MetaMata Parse, and revise the architecture to optimize the performance of the preprocessor when implementing the next version with the selected parser generator.

7 References

[1] Scientific Toolworks (May 1999), *QualGen*,
URL: http://www.scitools.com/qualgen.html

[2] International Software Automation(May 1999), *Panorama for Java/C/C++ Software Testing, QA, Documentation & Maintenance*,
URL: http://www.softwareautomation.com/index.htm

[3] Metamata, Inc. (May 1999), *Java Compiler Compiler (JavaCC)*,
URL: http://www.metamata.com/JavaCC

[4] Metamata, Inc. (May 1999), *JavaCC Documentation*,
URL: http://www.metamata.com/JavaCC/DOC/

[5] Aho A., R. Sethi, J. Ullman, "Compiler Principle Technique and Tools", Addison Wesley, 1986

[6] Ellis M. and Stroustrup B., "The Annotated C++ Reference Manual", Addison Wesley, 1990

[7] Metamata, Inc. (May 1999), M*etamata Parse*,
URL: http://www.metamata.com/products/parse.html