Implementation of Syntax Analyzer by using JavaCC

Aye Thiri Htun, Dr. Swe Zin Hlaing
University of Computer Studies, Yangon
ayethirihtun@gmail.com, swezinhlaingucsy@gmail.com

Abstract

A language compiler is one of the most crucial tools for the system development. Java Compiler Compiler (JavaCC) [tm] is the most popular parser generator for use with Java [tm] applications. For a given a context free grammar, it generates a recursive decent parser written in Java. A recursive descent parser defines a subroutine for each nonterminal in the grammar. Each subroutine is responsible for recognizing a sequence of terminal symbols produced by its nonterminal, and for removing that sequence of terminal symbols from the input sequence. It reads a grammar specification and converts it to a Java program. It can on any computer platform with a Java Virtual Machine. In this paper, we analyze and describe how to build a parser by applying JavaCC. We use LL (1) grammar rules, Left to Right scanning using one lookahead token. The proposed system is implemented by Java Programming Language.

Keywords: Java Compiler Compiler (JavaCC), recursive decent parser, context free grammar, Java Virtual Machine, LL(1) Grammar, lookahead token

1. Introduction

Every programming language has a set grammar rules associated with it. These grammar rules define the syntax of the language. Based on these grammar rules, a parser for the programming language can be constructed. There are two approaches that could be taken for the construction of the parser. Figure 1 shows the front end of the compiler; Lexical Analyzer and Parser.

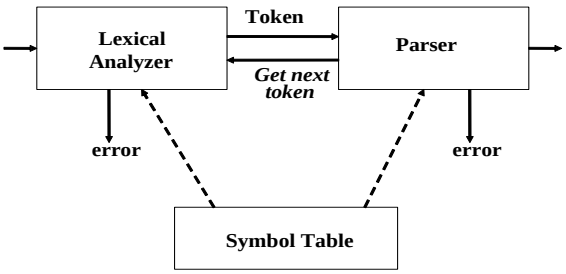


Figure 1 Position of parser in compiler model

Parsing (syntactic analysis) is one of the best understood branches of computer science. Parsers are already being used extensively in a number of disciplines: in computer science (for compiler construction, artificial intelligence), in linguistics (for text analysis, machine translation, and textual analysis of biblical texts), etc. Parsers can broadly broken down into Top-down parsers (LL: The first L means which scans left to right and the second L means that traces leftmost derivation of input string) and Bottom-up parsers (LR: L - Scan left to right and R - Traces rightmost derivation of input string). Typical notations LL (0), LL (1), LR (1) are LR (k), where k is the number of lookahead token [5]. This paper is organized as follows: session 1 introduce about the Parser, session 2 describes the theoretical background, session 3 discusses the related works, session 4 presents the design and implementation, session 5 describes the experimental results and the last session concludes this paper.

2. Related Works

Our work is designated to build the any kind of parser than can parse the specific grammar rules of the developed parser. JavaCC and the parsers it generates are certified as 100% pure Java and run on more than forty different platforms without any need for porting the code. In paper [6], Java based SPOOL environment applied the JavaCC grammar file we developed for supporting the UML-based CDIF Transfer Format. In paper [7], the Java Parsing System (JPS) tool was built that works in conjunction with the JavaCC parser generator to generate recursive-decent parsers written in Java, for languages specified by extended context free grammars.

Generated parsers communicate parser events to client applications through a standard API. JavaCC is a compiler generator that was jointly developed by Metamata and Sun Microsystems [5]. It produces recursive descent parsers and offers various strategies for resolving LL(1) conflicts. First, one can instruct JavaCC to produce an LL(k) parser with $k > 1$. Such a parser maintains k lookahead symbols and uses them to resolve conflicts. Many languages,

however, not only fail to be LL(1), but they are not even LL(*k*) for an arbitrary, but fixed *k*. Therefore JavaCC offers local conflict resolution techniques. In paper [2], the author describes the conflict resolving techniques. Paper [4] introduced a Java compiler generator called JavaCC and the application of the generator to develop a Java-based preprocessor for C/C++.

3. Theoretical Background

This session will discuss LL(1) grammar, parsing methods and the elimination of the left recursion for building the top down parser will be described. The Java Compiler Compiler will also be discussed.

3.1. LL (1) Grammar and Parsing Methods

A grammar whose parsing table has no multiply-defined entries is said to be LL(1) grammar. In LL(1), the first "L" denotes the left most derivation, the last "L" denotes the input scanned from left to right and "1" in parenthesis refers a look-head symbol do determine parser action. The parsing table of a grammar may contain more than one production rule. In this case, we say that it is not a LL(1) grammar. *Recursive descent parsing* is a top-down parsing method. Every nonterminal has one (recursive) procedure responsible for parsing the nonterminal's syntactic category of input tokens. When a nonterminal has multiple productions, each production is implemented in a branch of a selection statement based on input lookahead information. *Predictive parsing* is a special form of recursive descent parsing where we use one lookahead token to unambiguously determine the parse operations [3].

3.2. Elimination of Left Recursion

LL (k) parsers allow only right recursion in the production. Consider a commonly used syntax for Expression in C/C++ that calls 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 [4]. Therefore the conversion of the syntax into right recursion should be considered with cares.

Sample Grammar Rules

1. E→E'+'T | T
2. T→T'*'F | F
3. F→'('E')' | 'x'

Elimination of Left Recursive Grammar

1. E→TX
2. X→'+'TX | ε
3. T→FY
4. Y→'*'FY | ε
5. F→'('E')' | x

LL (1) Grammars

1. E→TX
2. X→'+'TX
3. X→ε
4. T→FY
5. Y→'*'FY
6. Y→ε
7. F→'('E')'
8. F→x

3.3. Java Compiler Compiler (JavaCC)

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. 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. Figure 2 shows the overall structure of a parser generated by JavaCC.

A parser specification for JavaCC consists of a set of grammar rules, each annotated with a semantic action that is a Java statement. Whenever the generated parser reduces by a rule, it will execute the corresponding semantic action fragment [8].

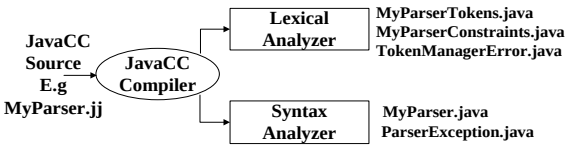


Figure 2 Generation of JavaCC Parser

4. Design and Implementation

In this session, we will outline the main contributions of the paper. We analyze the JavaCC grammar file format, implements the API for parsing Grammar files that is LL (1) grammar and we also provide the frame for writing LL(1) grammar files, Parsing system for input token streams. The system overview flowchart can be seen in Figure 3. The implementation program accepts the input grammar file. JavaCC Parser Generator parses and generates to the parser files (.java files). Our system compiles these Parser files into class files than can run on any platform. The compiled Parser file that is Parsing Program can be used to test the input token streams. The success or error messages for the input token streams are generated by the Parsing Program as shown in the system flowchart.

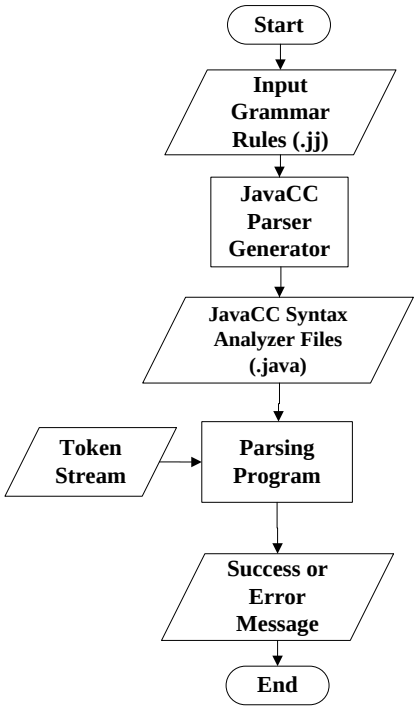


Figure 3 System Flowchart

4.1 LL(1) Grammar and JavaCC Input Grammar File Format for Sample Program

LL(1) Grammar for the sample parser is as shown in Figure 4. Our system provides the input grammar files format that is uploaded from the documentation of the JavaCC [5]. The file named " Parser.jj" is as shown in Figure 6. It enables to check the input token stream as shown in Figure 5:

```
<program>      --><header> VOID MAIN ()
                          <prog_block>
<header>       --> #INCLUDE <HEADER_FILE>
<prog_block>   --> { <block_stmt> }
<block_stmt>   --> <stmt>
<stmt>         --> <write>
<write>        --> COUT<< <write_list> ;
<write_list>   --> <write_listp> <write_list>'
<write_list>'  --> <write_listp> <write_list>' | ε
<write_listp>  --> <<  ID | << LITERAL
```

Figure 4 LL(1) Grammar File

```
# INCLUDE<HEADER_FILE>
VOID MAIN()
{
   COUT<< LITERAL << ID ;
}
```

Figure 5 Input token stream

```
PARSER_BEGIN(Parser)
public class Parser {
  public static void main(String args[]) throws
ParseException {
      Parser parser = new Parser(System.in);
      parser.Input();  }
}
PARSER_END(Parser)
SKIP :
{   " "   | "\t"   | "\n"  | "\r"   }
void Input() :{}
{    CheckSyntax()  <EOF>    }
void CheckSyntax() :{}
{
  header()  "VOID MAIN()"   prog_block()
}
void header():{}
{    "#" "INCLUDE<HEADER_FILE>"    }
void prog_block():{}
{    "{"   block_stmt() "}"    }
void block_stmt():{}
{       stmt()    }
void stmt():{}
{    write()  }
void write():{}
{
  "COUT<<"write_list()("<<"write_list())*";"
}
void write_list():{}
{   "ID"     | "LITERAL"    }
```

Figure 6 Parser.jj

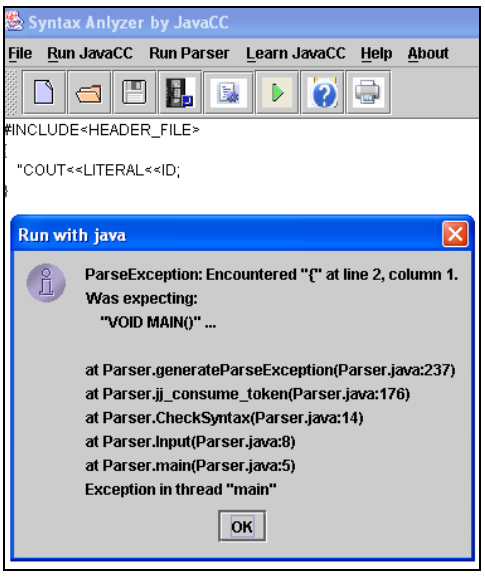
Figure 7 Parser.jj

4.2 Parser Files

JavaCC compiles Parser.jj with "javacc.bat" and output File Lists are as follows:
- Parser.java
- ParserCharStream.java
- ParserTokenManager.java
- ParserConstant.java
- ParseException.java
- Token.java
- TokenMgError.java

To generate parser class files for specific grammar, compile with "javac.exe". Class File Lists are as follows:
- Parser.class
- ParserTokenManager.class
- ParserConstant.class
- ParserCharStream.class
- ParseException.class
- Token.class
- TokenMgError.class

The parser that is generated by javacc be large, medium or small depends on the size of the grammar files. The parser generator can build parser for any kind of programming languages that can support LL(k) grammar. The error messages can be recognized according to the LL(k) grammar. In above example of Figure 6, if the input token streams do not include the "VOID MAIN()", the parser will generate error messages as shown in Figure 7. The error messages can be large, medium or small according to the input token streams' error status.



5. Experimental Results

This paper implemented the JavaCC Parser Generator API. The Input Grammar files can be typed in the text area or chosen by grammar sample frames.

According to the input grammar rules, the parser is generated. The generated parser can be tested by the input token stream.

The JavaCC parser generator used the predefined grammar rules that have to be generated parser to test the input token stream. The accuracy of the grammar file definition creates the definite success or error message depends on the predefined grammar. The runtimes of the test program named "Parser.jj, Parser.java, ParserTest.txt" which produced successful messages are measured in the following tables.

Table 1 Test Operating Systems

OS No.	Operating System	Disk Capacity	Memory	CPU
1.	Windows XP	160G	1GB DDR2	Intel Pentium Dual Core
2.	Windows Vista	160G	2GB DDR2	Intel Pentium Dual Core

Table 2 Runtime of Javacc

OS No.	Runtime (mili second)
1.	1.259968721671E12
2.	1.259929279994E12

Table 3 Runtime of Javac

OS No.	Runtime (mili second)
1.	1.259968813687E12
2.	1.259929308103E12

Table 4 Runtime of Java

OS No.	Runtime (mili second)
1.	1.259968875156E12
2.	1.259929375111E12

6. Conclusion

We described how to build a parser generator that is generated by JavaCC. The implementation of this system provides to build the syntax analyzer by using JavaCC. It can link between the *LL grammar building* and *analyzing and implementation of a language parser*. It can be used on any platform and can enhance according to the grammar rules expanded. This system produces success or error message of the user input testing token stream according to the grammar file definition. This system is part of the compiler front end. The Parser generator application is useful for testing LL(1) Grammar build by hand.

References

[1] A. Alfred, "Principles of Compiler Design", Bell Laboratories Murray Hill, New Jersey Ullman, Jeffery D. Princeton University, Princeton, New Jersey, 1997.

[2] A. Wöß, M. Löberbauer and H. Mössenböck, "LL(1) Conflict Resolution in a Recursive Descent Compiler Generator", Johannes Kepler University Linz, Institute of Practical Computer Science, 2005.

[3] D. Ben and K. Kiong, "Compiler Technology: Tools, Translators and Language Implementation", National University of Singapore. 1997.

[4] G. Succi and R. W. Wong, "The Application of JavaCC to Develop a C/C++ Preprocessor", University of Calgary, 2000.

[5] P. Alto, "Java Compiler Compiler: The Java Parser Generator". *Online documentation for version 0.7.1*. Sun Microsystems. Available at <http://www.sun.com/suntest/JavaCC >.

[6] P. Pagé, R. K. Keller, and R. Schauer, "A JavaCC Parser for the UML-Based CDIF Transfer Format", Université de Montréal Canda, 2000.

[7] P. Shaker, T. S. Norvell, "The Java Parsing System", Memorial University of Newfoundland Canda, 2004.

[8] W. Andrew, Appel and J. Palsberg, "Modern Compiler Implementation in Java, Second Edition", 2002.