

Lex and Yacc for Java

By: Kyriakos Assiotis

Supervisor: Dr. Pete Jinks

Date: 02/05/07

Chapter 1: Lex and Yacc for Java - Introduction

Chapter 2: Background and literature survey

Section 2.1: Lexical analysis and parsing

Section 2.2: Lex and Yacc

Section 2.3: Selection Criteria

Section 2.4: Research

Section 2.5: Research Results

Section 2.6: Why ANTLR, JavaCC and SableCC?

Chapter 3: Implementation and Testing

Section 3.1: ANTLR

Section 3.2: JavaCC

Section 3.3: SableCC

Chapter 4: Results

Chapter 5: Conclusion

Chapter 1: Lex and Yacc for Java -

Introduction

Lex and Yacc for Java. As the project title suggests my main objective is to find suitable alternatives of Lex and Yacc with the difference that these alternatives will support generating code in the Java programming language. One of the reasons why this project was proposed is because of the fact that Java is quite popular today and so the university should be able to at least offer it as an alternative for the COMP20121: The Implementation and Power of Computer Languages lab exercises. Also there is the issue that Lex and Yacc are considerably outdated and applications with a lot more features and flexibility are now available.

Nowadays there are lots of alternatives for Lex and Yacc and in order for me to find the most suitable ones I plan to compare them with a set of predefined desired features. Some of the tools today offer automatic tree building, better error handling and some even combine Lex and Yacc into one integrated tool. After I have narrowed down the list of programming tools available I will take the few remaining and test them with suitable examples, mainly variations of the calculator example. Hopefully, upon completing my tests I will have a thorough knowledge about them and I will be able to conduct some valuable and useful evaluations.

In chapter 2 I will briefly describe what lexical analysis and parsing means. Moreover, I will say a few words about Lex and Yacc and then I will describe the criteria I used for researching and comparing the different applications available. In addition I will describe how I came up with the conclusion that ANTLR, JavaCC and SableCC are the three applications that are best suited to use for my tests.

Chapter 3 is all about how I used ANTLR, JavaCC and SableCC to implement and test code. The example I used for implementation and testing is the calculator example, since it is very easy to use and understand.

Then in chapter 4 I describe what I learned from using ANTLR, JavaCC and SableCC and present my opinion and evaluation for them.

Chapter 6 is the final chapter in this report and it will hold my final conclusions and proposed solutions for the Lex and Yacc for Java problem.

Chapter 2: Background and literature

survey

Section 2.1: Lexical analysis and parsing

Basically what lexical analysis does is to take a sequence of characters and convert them into a series of tokens. Those tokens will then be analyzed to determine their grammatical structure with respect to a given formal grammar. This process is called parsing. Lexical analysis and parsing can be used in many applications. We can use them from creating a calculator to building a compiler for a known language or even a compiler for one of our own languages. There are a lot of problems available that can easily be fixed with some knowledge of Lexical analysis and parsing. More information about lexical analysis and parsing can be found in the COMP20121 lecture notes as well as books about the theory of parsing or books about compilers.

Section 2.2: Lex and Yacc

Lex is a programming tool that generates lexical analysers. It works by reading an input stream specifying the lexical analyser and then by outputting source code implementing the lexical analyser in the C programming language. The strength of Lex lies in recognizing strings and characters. As we can see from the example it can recognize the characters very well, but when it comes to recognizing a grammar Lex is really weak. That's where Yacc comes in.

[0-9]+

"+"

"*"

```
-  
"/"  
;  
[ \t\n]  
[^-0-9+*/; \t\n]+
```

A calculator example using just Lex

Yacc (an acronym for “Yet Another Compiler Compiler”) is a programming tool that generates parsers. It generates the parsers by recognizing grammars in Backus–Naur form (BNF). The code generated for the parser is in the C programming language. Yacc uses an LR parser technique (also known as bottom-up parsing) which derives the grammar from the leaves. LR means that the parser will read the input from left to right and produce a rightmost derivation. That is as opposed to LL parsing which also reads the input from left to right but produces a leftmost derivation. Both are methods for parsing context-free grammars. As we can see from the example below Yacc is pretty adequate when it comes to recognizing grammars, but very, very weak when it comes to recognizing characters. That is where Lex comes in and plays a significant part.

```
line: exp ';' '\n'  
exp: term | exp '+' term | exp '-' term  
term: factor | term '*' factor | term '/' factor  
factor: number | '(' exp ')'  
number: digit | number digit  
digit: '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

A calculator example using just Yacc

A remainder of what we saw from the two calculator examples above, Lex is weak when dealing with grammars while Yacc cannot recognize characters efficiently. That’s why we use them together. We have Lex read the input and create a suitable lexer that Yacc can match with one of its grammars and create the required parser. So a calculator using both Lex and Yacc will look something like this:

line : exp ';' '\n'
exp : term | exp '+' term | exp '-' term
term : factor | term '*' factor | term '/' factor
factor : number | '(' exp ')'
number : digit | number digit
digit: [0-9]+

An example using both Lex and Yacc

Section 2.3: Selection Criteria

Before I start to look at the different programming tools available I have to derive a list of features. The purpose of this list of features is to be a helpful mean in distinguishing the best applications from the rest; these criteria are mostly concerned for parsers, Scanners are more or less the same. Below I present this list along with a short explanation of each feature:

- Top-down Parsing (LL): The kind of parsing method that the application supports. Considered to have better error-handling mechanisms than bottom-up parsers.
- Bottom-up Parsing (LR): The kind of parsing method that the application supports. Considered to support more context-free grammars than top-down parsers
- Supports LALR: If it supports this specialized form of LR parsing. It is supposed to support even more grammars than LR or SLR do.
- Supports SLR: If it supports this specialized form of LR parsing.
- Tree Building Pre-processor: If there is an option to automatically build an abstract syntax tree for the grammar.
- Available Documentation: If there is available documentation for the application

- Available Examples: If there are available examples for the application
- Mailing Lists or Newsgroups: If there are available mailing lists or newsgroups that you can subscribe
- Supports EBNF: If the parser supports the Extended Backus–Naur form
- Full Unicode Support: If the parser provides full Unicode support

Section 2.4: Research

As soon as I had my features list ready I was able to start my research. As a starting point I used two links¹ that were made available from my project description at the university's web page. From those web pages I was able to find a lot of the available tools and thus I was able to begin looking at them. I was looking to see if they had some examples to get me started as well as if there was an obvious way for me to get help if I was facing problems with the tool, mainly through documentation, mailing lists and forums. In addition, I was looking for tools that could offer more options than what Lex and Yacc offer. For example, tools that integrate the two of them in one tool or the option to built trees.

Parser Generators:

BYACC: Byacc is basically Yacc but with the difference that it generates code for Java. So, proposing Byacc as an alternative would seem to be ideal, but in fact we will be missing out on the other features that are available from similar programming tools. Features like the automatic building of trees.

¹ <http://catalog.compilertools.net/java.html>
<http://java-source.net/open-source/parser-generators>

CUP: Cup is more or less like Byacc. It's another LALR parser generator that was build after Yacc that supports our required Java language.

JAY: Jay is another Yacc look-alike programming tool.

Beaver: Beaver is an LALR parser generator. The Beaver API provides easy integration with JFlex and JLex. It supports EBNF as many other parser generators do, but it also provides functions for building abstract syntax trees (AST). Beaver also provides documentation and a few examples to get you started as well as open discussion forums.

JELL: Jell is a parser generator that recognizes LL grammars. Jell provides documentation and examples to get you started.

OOPS: OOPS is an LL parser generator tool. It recognises grammars in EBNF, provides documentation and examples and also provides a way of building trees.

Grammatica: Grammatica is an LL parsing generation tool. It can generate code in both C and Java programming languages. It also provides documentation and mailing lists.

SJPT: The Simple Java Parsing Toolkit is a parser generator that supports both top-down and bottom-up parsing.

JB: The Java Bison tool basically takes parsers generated by Bison in the C language and provides a way to execute them in the Java language. Quoting from a JB website "The parser is written using the standard Bison language, but the action parts are written in Java. JB takes the mixed C-Java file output by Bison and scans it to extract the parse tables, constants, and actions (in Java)²".

JACCIE: Since I could not find any web pages about Jaccie I assumed that it either changed its name to something else or it is no longer available. The only information I could find was "Jaccie includes a scanner generator and a variety of parser

² <http://serl.cs.colorado.edu/serl/misc/jb.html>

generators that can generate LL(1), SLR(1), LALR(1) grammars. It has a debugging mode where you can operate it non-deterministically³”.

JTopas: JTopas is another parser generator tool. As stated in its website “Use it when the JDK tokenizers are too limited, JavaCC, JTB (Java tree builder) etc. are too complicated, or you need dynamic parser configuration⁴”

PAT: PAT is another parser generator tool that provides documentation and examples.

Lexical Analyzers:

JAX: Jax is a lexical analysis generator like Lex. Jax only recognizes the 7 bit ASCII characters, which means that it does not support the full Unicode standard. Unlike Lex though, Jax does not support defining macros or context dependent expression matching. It has some examples and documentation available.

LOLO: Lolo is a Lexical analysis tool. It recognizes the Unicode standard and it provides documentation and examples.

JLEX: JLex is another tool that generates lexical analyzers. It is basically Lex but with the difference that it generates code for the Java language. It is built in such a way that it integrates perfectly with the Cup parser generator. It also supports the full Unicode standard.

JFLEX: JFlex is in a few words a better and upgraded version of JLex. It has all the featured that JLex has and it is compatible with JLex. Moreover, it can also integrate with Byacc, create standalone scanners as well as generate scanners faster than JLex.

Tools that combine lexical analysis and parser generation:

³ <http://www.thefreecountry.com/programming/compilerconstruction.shtml>

⁴ <http://sourceforge.net/projects/jtopas>

COCO/R: Coco/R is a programming tool that combines Lex and Yacc. It has available documentation, it supports EBNF and even it has an Eclipse plug-in. Moreover, there are versions of Coco/R that also support the C programming language. The fact that Coco/R integrates Lex and Yacc as well as it can support C makes Coco/R a very interesting tool to look and have a further go at.

Generic Interpreter: Generic Interpreter is another programming tool that combines both lexical analysis and parser generation. Its main strength is that it supports LL and LR parsing, as well as SLR. It also has available documentation and provides a lot of examples to get started with.

RunCC: RunCC is a programming tool that combines the lexical analysis and the parser generation in one. The parse generator that it provides can be LR, SLR or LALR. It provides some documentation and examples and it also specifies that it features simplicity so it claims to be better to use with small languages.

ANTLR: Another Tool for Language Recognition looks like one of the better candidate tools I have seen. It combines lexical analysis and parsing, it uses the top-down parsing method, it has examples and documentation available and it also provides means for building trees.

JavaCC: JavaCC is another tool which combines Lex and Yacc in one. Like ANTLR it uses the LL parsing method, it provides documentation and several examples and it comes with a tree building pre-processor.

SableCC: SableCC like ANTLR and JavaCC provides a way for automatic tree-building, but unlike them it uses the LALR parsing method. It provides documentation and a tutorial to get started.

Section 2.5: Research Results

After having a look at these programming tools I had to decide which ones I was going to choose in order to test and actually use. From my research I found that even though the LALR parsing method was the best, in terms of being able to recognize the most grammars, most parser generating tools opted to use the LL parsing method as the error recovery is handled better and easier than tools with LALR parsing (Yacc uses the LALR parsing method). As far as lexical analysis is concerned, I found out that most tools that generate code for the Java language are basically copies of Lex with just a few advancements, mainly due to the progress and discovery of new processing techniques.

So for the Lex tool alternatives we have JLex and JFlex which are the most popular and probably best suited for Lexical analysis for the Java language. LOLO and Jax could also be used, even though Jax is much weaker than JLex and JFlex.

As far as parsing is concerned, there is a whole bunch of alternatives to choose from. Byacc, Cup and Jay are all Yacc (LALR) look-alike tools with Byacc and Cup being more popular than Jay. Beaver is also an LALR parsing method tool but with the difference that it provides functions for building trees. This fact makes Beaver a little more interesting than the other aforementioned bottom-up parsing tools. Instead of bottom-up parsers, I could also choose some top-down parsers to have a further go at. Jell, Grammatica and OOPS are examples of LL parsing tools, with OOPS providing a way to build trees. Moreover, there are tools available that provide both parsing methods. SJPT and JACCIE are examples of those kinds of tools, even though I could not find any additional information about JACCIE. PAT and JTopas are also 2 parser generator tools that I could not find all the information I wanted; I couldn't find whether they parse bottom-up, top-down or even in both ways. Lastly, as far as parser generators are concerned, JB is a tool that just translates in Java from the Bison tool

Tools like Coco/R, Generic Interpreter and RunCC seem very interesting because they combine the lexical analysis and the parsing generation. From those

tools, the Generic Interpreter can use the LL, the LR as well as the SLR parsing methods to recognize grammars, though it does not support the LALR method, which Coco/R and RunCC do.

Still, the most interesting tools that I saw were the ones that combined both lexical analysis and parsing generation and provided means to automatic tree building. Those tools are ANTLR, JavaCC and SableCC and they are the ones that I chose to test and further evaluate. Between the three of them I can use both top-down and bottom-up parsing, try their respective tree building functions and overall have a comprehensive opinion about them.

Section 2.6: Why ANTLR, JavaCC and SableCC?

First of all, I chose the three of them because they all satisfied all of my selection criteria; Beaver was the only other parser that satisfied them. Hence, they kind of automatically became lead contestants for having a further go at. Moreover, they combined the parser and the scanner, which I believe is a little bit more helpful than having to use a tool for lexical analysis and then another for parsing the grammar; If I chose Beaver I would most probably have use it with JFlex. In addition, ANTLR and JavaCC use a different parsing method than Yacc which makes it a little bit more interesting to see how grammars can be recognized. The main reason for choosing ANTLR, JavaCC and SableCC is that the three of them look like the most complete applications for the Java language that are readily available.

Chapter 3: Implementation and Testing

In order for me to get started I had to decide what course of action I was going to take in order to test ANTLR, JavaCC and SableCC. After some serious thought I decided to implement the calculator example on all three of them. I will start by implementing a very simple version of it, so I can get to know how the three tools work, and then I plan to build and extend on that example.

The basic calculator is a great example to use because, first of all, it is one of the COMP20121 course lab exercises. In addition, it can be easily represented in the BNF form, also there are a lot of possible ways to extend that example- e.g. make it calculate powers. That simplicity and power of the calculator example, I believe, will show the strengths and weaknesses of ANTLR, JavaCC and SableCC. I plan to start working the example with ANTLR, then JavaCC and lastly SableCC.

Section 3.1: ANTLR

In order for me to get started and gain some experience in using ANTLR I decided to do the cut and paste example available from the ANTLR website⁵. This example will show me, in a small scale, just how ANTLR really works. Before I tried the example I had to download the ANTLR software from the website⁶. I opted to download the ANTLR2.7.7 version, since it was mentioned that it was the current stable version. After downloading the software I unpacked it in a suitable folder-home/S04/assiotk4/project/antlr. After doing that I was ready to try the example I found. The instructions were pretty straight forward, all I had to do was to cut and paste some code in my text editor; what this example really does is to take 2 arguments with the first argument being any lower case or upper case letter and the 2 any number, as well as being able to ignore white space. When I was done creating

⁵ <http://wwwantlr.org/article/cutpaste/index.html>

⁶ <http://wwwantlr.org/download.html>

and compiling- a small problem I faced when I first tried to compile this text file was that I had not defined the classpath to contain the ANTLR source file- the first text file which contained the lexer and the parser, I found out that a separate Java file was needed in order for the lexer and parser to work. This “main” file was needed just to create the lexer and the parser and then begin the parsing by calling the starting method; in this case we only have one rule called “startRule”. The little program worked just fine once I ran it.

From doing this example I learned how to write the lexer and parser in one file and how to make those two integrate with each other. Moreover, I found out that while using ANTLR I will have to create a Java file where I will have to create instances of the lexer and the parser and of course have a command that calls the first rule of the parser. I also found it helpful to create a little script which will contain all the commands needed for compiling all the necessary files. These commands are the export classpath command, the run of the antlr.Tool on the text file, which contains the lexer and the parser, and then the compilation of all the Java files. If all is done correctly then all I have to do is run the main Java file.

Knowing how to use ANTLR, even its basic operations, I was able to implement the first step of the calculator example. There was an available calculator example in the ANTLR website⁷ which I was going to use as my starting point. At first I would implement a grammar that would recognize integer addition, subtraction and multiplication actions. The grammar will also recognize actions within brackets and it will follow the natural mathematical precedence. The lexer will have to recognize parenthesis, any integers and also ignore white space. Following this tutorial I learned how to enable the lookahead specifications of ANTLR and also how to enable ANTLR to recognize all the ASCII characters. Once I finished this first implementation of the calculator, I decided to follow and finish the tutorial.

The next step was to print the value that was resulted from the calculators’ operations. This was easily done by having a variable that was altered according to the correct action. This variable was passed down to all the grammar rules and then

⁷ <http://www.cs.usfca.edu/~parrr/course/652/lectures/antlr.html>

returned with the new value. When all the calculations were over the variable was returned to the main Java file where there was a simple print statement, so we can view the result beneath the calculation. By doing this example I was able to find out how actions can be performed within the grammar, as well as how to use variables with ANTLR.

Next in the tutorial was a lesson on how to build abstract syntax trees with ANTLR. In conjunction with the ANTLR documentation the example was pretty straightforward. As far as the grammar was concerned all that was done was to specify which elements were going to be subtree roots; this was done by annotating them with the '^' character. An option to leave tokens out from the tree building process exists by using the exclamation mark. This example also showed me how to use actions within the tree parser. The value of the calculation is now computed in the tree parser.

Once I was done with the tutorial I decided to extend the calculator in order for it to work indefinitely, unless terminated by the user. The current working model of the calculator allows only one operation to be calculated. This of course is not efficient. I would like the calculator to allow multiple operations, while still maintaining its current form. For example the tree as well as the value of each operation should be calculated and shown right after the operation is inputted. I was able to achieve that by adding a further rule to the grammar, but not without complications, which I will mention in the next chapter.

```
export CLASSPATH=/home/S04/assiotk4/project/antlr/antlr.jar:.  
java antlr.Tool calc.g  
javac *.java
```

This is the shell script I used for successfully compiling my code. First of all I had to export the classpath to include the ANTLR source file, then run the ANTLR tool on my code and lastly compile all the Java files that were automatically created by the ANTLR tool as well as the Java file I use for calling the lexer and parser.

```
class ExprParser extends Parser;
```

```

options {    buildAST=true;}
begin : FLUSH | (expr)*;
expr: mexpr ((PLUS^|MINUS^) mexpr)*
    ;
mexpr
    : atom (STAR^ atom)*
    ;
atom: INT
    | LPAREN! expr RPAREN!
    ;

class ExprTreeParser extends TreeParser;
options {
    import Vocab=ExprParser;
}
expr returns [int r=0]
{ int a,b; }
    : #(PLUS a=expr b=expr) {r = a+b;}
    | #(MINUS a=expr b=expr) {r = a-b;}
    | #(STAR a=expr b=expr) {r = a*b;}
    | i:INT          {r = (int)Integer.parseInt(i.getText());}
    ;
class ExprLexer extends Lexer;
options {
    k=2;
}
LPAREN: '(' ;
RPAREN: ')' ;
PLUS : '+' ;
MINUS : '-' ;
STAR : '*' ;
FLUSH : '\n' ;
INT : ('0'..'9')+ ;
WS : ('

```

```

| '\r'
| '\t'
)
{$setType(Token.SKIP);}
;

```

This is what my final code for ANTLR looks like. At the top we have the parser with the option for automatic tree building set to true. Then we specify the rules which the grammar must follow as well as which expressions will be subroots. Then the tree parser is specified. This is where all the actions, for computing the value, take place. And lastly we have the lexer where we specify which sequences of characters will be recognized, as well as which sequences of characters will be ignored. In the options of the lexer 'k' stands for the look-ahead specification for this parser.

```

import antlr.collections.*;
import java.io.*;

public class Main {
    public static void main(String[] args) throws Exception {
        try
        {
            ExprLexer lexer = new ExprLexer(System.in);
            ExprParser parser = new ExprParser(lexer);
            while (true){
                parser.begin();
                AST t = parser.getAST();
                System.out.println(t.toStringTree());
                ExprTreeParser treeParser = new ExprTreeParser();
                int x = treeParser.expr(t);
                System.out.println(x);
            }
        } //try
        catch(Exception e) {

```

```
System.err.println("exception: "+e);
    }//catch
}//main
}//Main
```

This is the main method that I created. It works by first creating an instance of the lexer, where in this case it will take its arguments from the command line at runtime. Then an instance of a parser is created based on the already created lexer. Then I created an infinite loop so I can have the calculator to be able to compute infinite computations. Then the starting rule of the parser is called so we can begin parsing. Once done with parsing will create an instance of the abstract syntax tree and then traverse that tree in order to find our operation's value. Finally that value is returned and printed. I have to mention that a lot of different Java files are generated when running the ANTLR tool. It is important that all of those files are compiled.

```
1+2-(3*4)+5
( + ( - ( + 1 2 ) ( * 3 4 ) ) 5 )
-4
```

<AST>:0:0: unexpected AST node:

0

This is the result I got when I tried to execute that mathematical operation. The evaluation of the operation as well as the creation of the abstract syntax tree with the printing of the final value, work just fine. That other output resulted from trying to run the same process on a new line character.

Section 3.2: JavaCC

With ANTLR out of the way I was able to move on to my next tool of choice, JavaCC. Like ANTLR I had a look at some available examples in order to find exactly how JavaCC operates. From what I saw it required only one file to operate and that file was going to have a Java-like format. I must declare a main method from where the parser will be called and then declare the lexer under the name `TOKEN`. Finally I had to declare the parser in a way similar to declaring Java methods. Using actions within the parser was pretty straightforward.

I was now ready to begin implementing the calculator with the help of JavaCC. But before I did that I had to download and unpack the latest working version of JavaCC from its website⁸ - in the folder `/home/S04/assiotk4/project/javacc-4.0`. With the software downloaded and ready to use I begun writing the code for the calculator. The first basic part of the calculator was easily done. The lexer was roughly the same with ANTLR with only the notation changing. The parser had to be written with the JavaCC format, but the rules were still the same. Now I had to make the calculator to compute and print the value of the operation carried. I have to say I found doing this with JavaCC to be less trivial than doing it with ANTLR; not that it was particularly hard with ANTLR. I came to that conclusion because JavaCC only needs one file to run and so you do not have the need to pass values around and since implementing the actions in the parser is pretty much the same with both tools that reason is sufficient enough to give a small edge to JavaCC.

It was now time to try and use the automatic tree building that JavaCC offers, called `JJTree`. Unfortunately for me, despite my efforts I was not able to successfully use that option of JavaCC. From what I read from the JavaCC documentation I had to create another file specifically for the tree parser and somehow integrate it with my grammar. So I decided to skip the tree building option momentarily and try to make the calculator execute multiple operations, while showing the result right after each one of them. Luckily, I was able to achieve that by just adding a rule to my grammar. Even though I was not able to build trees with JavaCC, I was capable enough to make

⁸ <https://javacc.dev.java.net/>

it perform those multiple operations better than what ANTLR did. I was now ready to tackle the last of my choices SableCC.

options

```
{  
    LOOKAHEAD=2;  
}
```

PARSER_BEGIN(Calc)

public class Calc

```
{  
    public static void main(String args[]) throws ParseException  
    {  
        Calc parser = new Calc(System.in);  
        while(true)  
        parser.begin();  
    }  
}
```

PARSER_END(Calc)

SKIP:

```
{  
    " " | "\r" | "\t"  
}
```

TOKEN:

```
{  
    <INT : ("0"-"9")+> |  
    <PLUS : "+"> |  
    <MINUS : "-"> |  
    <STAR : "*"> |  
    <LPAREN : "("> |  
    <RPAREN : ")"> |  
    <FLUSH : "\n">
```

```
}
```

```
void begin():
```

```
{
```

```
int a;
```

```
}
```

```
{
```

```
<FLUSH> | a=expr() {System.out.println(a);}
```

```
}
```

```
int expr():
```

```
{
```

```
int a;
```

```
int b;
```

```
}
```

```
{
```

```
a=mexpr() (<PLUS> b=expr() {a = a+b;}
```

```
| <MINUS> b=expr() {a = a-b;})*
```

```
{return a;}
```

```
}
```

```
int mexpr():
```

```
{
```

```
int a;
```

```
int b;
```

```
}
```

```
{
```

```
a=atom() (<STAR> b=atom() {a = a*b;})*
```

```

    {return a;}
}

int atom():
{
    Token t;
    int a;
}

{
    t=<INT> { return Integer.parseInt(t.toString()); } |
    <LPAREN> a=expr() <RPAREN> {return a;}
}

```

This is an example of how JavaCC recognizes grammars. The options are declared first, in this case just the look-ahead. Then we declare the parser and inside the parser I had a main method that created an instance of the parser and then went into an infinite loop, forever calling the first rule. The Lexer is declared in two steps. Firstly, after the title “SKIP” we have all the sequences of characters that we need not recognize. And secondly, we have the title “TOKEN” where all the sequences of characters that we need to recognize are declared. In addition to the lexer will also declare the rules of the grammar. In JavaCC declaring rules looks a lot like declaring methods. Like ANTLR, when running the JavaCC tool a lot of Java files are automatically generated. All of those files must successfully compile in order for the program to work.

```

3+5*6
33
9-6*(4-2)
-3

```

We can see from this example that the multiple operations calculator works just fine. When I compared it with the result I got from ANTLR, I realized how important the automatic tree building is when it comes to understanding grammars.

Even though it works just like it is supposed to, it seems naked without an abstract syntax tree.

Section 3.3: SableCC

From my research of the various tools available SableCC seemed to be one of the best around but when I tried to test it I found many difficulties. The first obstacle that I encountered was that I could not use the same grammar as with ANTLR and JavaCC. The idea was the same but I had to implement the calculator with a different way. With SableCC I could not recursively recall the same rules, in order for me to overcome this I had to define a separate rule for every expression that I wanted to be used more than once. And basically that's all that I managed to do with SableCC. The tutorial that was available didn't prove to be of much help because that tutorial describes a way to build a compiler for a small Pascal language, which is a little bit more advanced than I was looking for. That tutorial only helped me to understand how to define the lexer and the parser and how the file is ran. I could not understand how to implement actions within SableCC and I could not find any useful help on how to use its tree building pre-processor. Moreover, I remember facing some trouble with exporting the correct classpath; I had to try several options until I found the right one. I have to say I was a bit disappointed with my inability to cope with SableCC because it did look like one of the best tools around.

Package calc;

Tokens

int = ['0' .. '9']+;

plus = '+';

minus = '-';

star = '*';

lparen = '(';

rparen = ')';

ws = (' ' | 13 | 10)+;

Ignored Tokens

ws;

Productions

expr = {mexpr}mexpr tail* ;

**tail = {plus} plus mexpr
 | {minus} minus mexpr ;**

mexpr = {atom} atom mtail* ;

mtail = {star}star atom ;

atom = {int} int | {expr} lparen expr rparen ;

This is as far as I have managed to get with SableCC. The package declaration is just the name of the file, which is then followed by the lexer where we specify which sequences of characters we should recognize and then declare which sequences of those would be ignored; this declaration of the lexer reminds a bit the declaration of the JavaCC lexer. Under the name “Productions” the parser rules are declared. Here in contrast with both JavaCC and ANTLR, we have to break down the expressions so no expression can recursively call itself. I have not been able to figure out why the “things” in the curly brackets are needed for, e.g. {mexpr}mexpr.

Chapter 4: Results and Evaluation

When I finished my implementation, or at least my effort for implementation, with the three tools it was time to reflect on what I've learned from all of them. Firstly, I would like to comment on just how easy or hard it was for me to try and work with these tools. As a student with limited knowledge on grammar and parser theory I relied on the tools themselves to provide me with enough feedback so I can achieve something. I found ANTLR to be the best tool of the three in terms of providing examples just to get started. JavaCC was not that bad in the particular field, even though the clarity and usefulness of examples was not as good as ANTLR I was still able to, fairly easily, understand how JavaCC operates. That was not the case with SableCC though. I had a really hard time in understanding how I was supposed to write a calculator with the help of SableCC and I have to say that I still don't fully comprehend on how things work with it. And if that wasn't enough the unfathomable error messages of SableCC came to add to my woes. I could not differentiate whether a mistake was a simple syntactic or major semantic one. And that came as opposed to ANTLR where I have to say that one of its error messages is still carved in my mind. I'm talking about a specific error message where you simply forget to end a statement with the semicolon. This error message mentions, amongst other possible errors, in clear English that you may have forgotten to end the previous statement.

Now as far as actually writing code, I found JavaCC to be the easiest of the three. I believe that that was because I'm more familiar with writing code using the Java style that JavaCC uses and also because of the fact that JavaCC requires everything to go in just one file; trees excluded. With ANTLR I had to use a notation that looked a lot like Lex and Yacc, which evidently is better when proposed as Lex and Yacc alternative. From the little I learned about SableCC I can say that it also uses a notation like Lex and Yacc but that is when it comes to recognizing very simple grammars. Like I mentioned before, I was not able to figure out how actions can be used within SableCC, hence I cannot know whether its actions are similar to those used in Yacc; from what I know I believe that they are not.

One of the reasons why I chose the three of them became obsolete after the implementation and testing was over. I'm talking about tree building as I was only able to use the automatic tree building option with ANTLR. Nonetheless I'm sure there is an easy and efficient way to use that option with the other two tools as well. I do know that for JavaCC I have to create a JJTree file but I couldn't find out how to integrate the JJTree file with the JavaCC file. As far as SableCC is concerned I found it pointless to even bother and search for suitable examples of tree building, and that was because I could not even understand how to use actions with it. Returning to ANTLR though, I said in the previous chapter that I will mention some of the complications I encountered when implementing the calculator that allows multiple operations with the automatic tree building processor. Basically what I did is to restrict each operation to a single line and to have the parser in an infinite loop. By doing so I could ensure that whenever a new line was recognized the program will calculate and print the previous operation. The problem with the extra code that I wrote was that I could not find a way of distinguishing, in my main method, between the new line and the operation, so that a tree will not be built for each new line. I think the problem was that I could not find a way to return that value so I could have an if-then-else statement in my infinite loop.

With JavaCC though, I was able to achieve the result that I wanted, regarding the multiple operation calculator. The single line for each operation restriction also applied in this case. Even though it is disappointing not to be able to integrate the JJTree tool with my working calculator example I was fairly satisfied. It was disappointing because I believe I could have made it work and thus have a full working model of what I was trying to achieve.

The results I got from working with SableCC were even more degrading to me than I had ever hoped to. I can't help but have the feeling though that if a suitable tutorial, for amateur users, about SableCC had existed then I might have actually enjoyed working with it. With the current situation though, it was one hurdle after the other, and all I eventually managed to do was create the most basic of the calculator examples. That example had neither a way to calculate and print the resulting value nor an automatically built abstract syntax tree of the grammar it recognized.

Nevertheless I still regard SableCC as a remarkable tool and I'm sure that once someone gets the hang of it, it becomes a valuable tool.

Chapter 5: Conclusion

Before I conclude this report I would just like to mention what I have gained from this experience. First of all I learned a bit more about lexical analysis and parsing and I am confident enough that if a situation arises where knowledge for the above material is required, then I will be able to rise to the challenge. In addition, I now believe that if I get enough practice with a certain tool, I can eventually create my own small language and build a compiler for it. I have also learned that little shell scripts help save a lot of time and effort, as well as how to use the CLASSPATH. I also learned how to be a little more consistent than what I was. I have to say that tackling a large project on my own was a unique unforgettable experience and I hope that this experience that I gained will come to good use later on.

As I have shown in the previous chapters there are a lot of different tools, which essentially do the same job, available for the Java language. As alternatives of Lex and Yacc one could automatically propose that their counterparts should be used. By counterparts I mean JLex or JFlex in conjunction with Cup or Byacc. To have these as alternatives for the lab would be the easy choice to make. By doing so though we will be missing the opportunity to offer something better as alternatives. After using ANTLR, JavaCC and even SableCC I am of the opinion that one of them, if not all, should be used as an alternative to the lab exercises. Even though I regard those three tools, which I've actually tried to use, as suitable alternatives I would not like to immediately disregard all the other options available. I am of the opinion that tools like Generic Interpreter, Coco/R and even RunCC, which also combine the parsing and lexical analysis, could fare pretty well if put to the task.

The fact that all of the lab exercises can be executed by most of the Java-based tools that I have seen, makes the choice of just one a bit more insignificant. That is because the university could offer the worst amongst these tools and still be perfect for the labs exercises. So I think it will be useful to offer tools that students will find helpful and intriguing enough in order for them to want to use them for other purposes

than just the COMP20121 lab exercises. Finally what I've gained from doing this project is the belief that if the university is really considering offering an option to use the Java-language with a more advanced tool than both Lex and Yacc combined, then it should considered offering direct alternatives of them for the C language as well.

Reference:

“My project at the University of Manchester”
<http://intranet.cs.man.ac.uk/ugrad/projects/year06/lang.html#A6>

“ANTLR”
<http://www.antlr.org/>
<http://www.cs.usfca.edu/~parrr/course/652/lectures/antlr.html>
<http://www.antlr.org/doc/getting-started.html>
<http://www.antlr.org/article/cutpaste/index.html>
<http://www.antlr.org/doc/index.html>

“SableCC”
<http://www.sablecc.org/>
<http://www.sablecc.org/features.html>
<http://www.sablecc.org/documentation.html>
<http://www.brainycreatures.org/compiler/sablecc.asp>

“Beaver”
<http://beaver.sourceforge.net/index.html>

“Cup”
<http://www.cs.princeton.edu/~appel/modern/java/CUP/>
<http://www.cs.princeton.edu/~appel/modern/java/CUP/manual.html>

“JavaCC”
<https://javacc.dev.java.net/>
<https://javacc.dev.java.net/doc/features.html>
<https://javacc.dev.java.net/doc/docindex.html>

“JFlex”
<http://www.jflex.de/>
<http://www.jflex.de/features.html>
<http://www.jflex.de/docu.html>

“JTopas”
<http://jtopas.sourceforge.net/jtopas/>
<http://sourceforge.net/projects/jtopas/>

“RunCC”
<http://runcc.sourceforge.net/>

“Grammatica”
<http://grammatica.percederberg.net/index.html>
<http://grammatica.percederberg.net/doc/rationale.html>
<http://grammatica.percederberg.net/doc/release/features.html>

“SJPT”

<http://sjpt.sourceforge.net/>

“Byacc”

<http://byaccj.sourceforge.net/>

“Coco/R”

<http://www.ssw.uni-linz.ac.at/Projects/Coco/Coco.html/#Java>

<http://www.ssw.uni-linz.ac.at/Projects/Coco/Coco.html/Doc/UserManual.pdf>

“Generic Interpreter”

<http://www.csupomona.edu/~carich/gi/>

“Jax”

http://linux4u.jinr.ru/usoft/WWW/www_blackdown.org/kbs/jax.html

“Jay”

<http://www.informatik.uni-osnabrueck.de/alumni/bernd/jay/>

“Jell”

http://linux4u.jinr.ru/usoft/WWW/www_blackdown.org/kbs/jell.html

“JLex”

<http://www.cs.princeton.edu/~appel/modern/java/JLex/>

<http://www.cs.princeton.edu/~appel/modern/java/JLex/current/manual.html>

“LOLO”

<http://luna2.informatik.uni-osnabrueck.de/alumni/bernd/lolo/>

“OOPS”

<http://luna2.informatik.uni-osnabrueck.de/alumni/bernd/oops/>

“PAT”

<http://www.javaregex.com/>

<http://www.javaregex.com/new.html>

“Wikipedia”

http://en.wikipedia.org/wiki/Lex_programming_tool

<http://en.wikipedia.org/wiki/Yacc>

<http://en.wikipedia.org/wiki/Parsing>

http://en.wikipedia.org/wiki/Lexical_analysis

All of the above sites were entered frequently between October and May, with last date entered being 02/05/07.