


Advertisement: Support JavaWorld, click here!



December 2000

[HOME](#)
[FEATURED TUTORIALS](#)
[COLUMNS](#)
[NEWS & REVIEWS](#)
[FORUM](#)
[JVM RESOURCES](#)
[ABOUT JVM](#)

Cool Tools

Build your own languages with JavaCC

JavaCC makes it a snap to write your own compiler or interpreter for languages of your own design in Java

Summary

In this first edition of the new Cool Tools column, Oliver Enseling discusses JavaCC -- the Java Compiler Compiler. JavaCC facilitates designing and implementing your own programming language in Java. You can build handy little languages for problems at hand or build complex compilers for languages such as Java or C++. Or you can write tools that parse Java source code and perform automatic analysis or transformation tasks. Oliver will discuss some of the fundamentals of compiler construction and show how to write a handy little command-line calculator. (1,670 words)

By Oliver Enseling

Do you ever wonder how the Java compiler works? Do you need to write parsers for markup documents that do not subscribe to standard formats such as HTML or XML? Or do you want to implement your own little programming language just for the heck of it? JavaCC allows you to do all of that in Java. So whether you're just interested in learning more about how compilers and interpreters work, or you have concrete ambitions of creating the successor to the Java programming language, please join me on this month's quest to explore JavaCC, highlighted by the construction of a handy little command-line calculator.

Compiler construction fundamentals

Programming languages are often divided, somewhat artificially, into compiled and interpreted languages, although the boundaries have become blurred. As such, don't worry about it. The concepts discussed here apply equally well to compiled as well as interpreted languages. We will use the word *compiler* below, but for the scope of this article, that shall include the meaning of *interpreter*.

Compilers have to perform three major tasks when presented with a program text (source code):

1. Lexical analysis
2. Syntactic analysis
3. Code generation or execution

The bulk of the compiler's work centers around steps 1 and 2, which involve understanding the program source code and ensuring its syntactical correctness. We call that process *parsing*, which is the *parser's* responsibility.

Lexical analysis (lexing)

Lexical analysis takes a cursory look at the program source code and divides it into proper *tokens*. A token is a significant piece of a program's source code. Token examples include keywords, punctuation, literals such as numbers, and strings. Nontokens include white space, which is often ignored but used to separate tokens, and comments.

Syntactic analysis (parsing)

During syntactic analysis, a parser extracts meaning from the program source code by ensuring the program's syntactical correctness and by building an internal representation of the program.

Computer language theory speaks of *programs*, *grammar*, and *languages*. In that sense, a program is a sequence of tokens. A literal is a basic computer language element that cannot be further reduced. A grammar defines rules for building syntactically correct programs. Only programs that play by the rules defined in the grammar are correct. The language is simply the set of all programs that satisfy all your grammar rules.

During syntactic analysis, a compiler examines the program source code with respect to the rules defined in the language's grammar. If any grammar rule is violated, the compiler displays an error message. Along the way, while examining the program, the compiler creates an easily processed internal representation of the computer program.

A computer language's grammar rules can be specified unambiguously and in their entirety with the EBNF (Extended Backus-Naur-Form) notation (for more on EBNF, see [Resources](#)). EBNF defines grammars in terms of production rules. A production rule states that a grammar element -- either literals or composed elements -- can be composed of other grammar elements. Literals, which are irreducible, are keywords or fragments of static program text, such as punctuation symbols. Composed elements are derived by applying production rules. Production rules have the following general format:

```
GRAMMAR_ELEMENT := list of grammar elements
                  | alternate list of grammar elements
```

As an example, let's look at the grammar rules for a small language that describes basic arithmetic expressions:

```
expr := number
      | expr '+' expr
      | expr '-' expr
      | expr '*' expr
      | expr '/' expr
      | '(' expr ')'
      | - expr
number := digit+ ('.' digit+)?
digit := '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'
```

Three production rules define the grammar elements:

- expr
- number
- digit

The language defined by that grammar allows us to specify arithmetic expressions. An `expr` is either a number or one of the four infix operators applied to two `exprs`, an `expr` in parenthesis, or a negative `expr`. A number is a floating-point number with optional decimal fraction. We define a `digit` to be one of the familiar decimal digits.

Code generation or execution

Once the parser successfully parses the program without error, it exists in an internal representation that is easy to process by the compiler. It is now relatively easy to generate machine code (or Java bytecode for that matter) from the internal representation or to execute the internal representation directly. If we do the former, we are compiling; in the latter case, we talk about interpreting.

JavaCC

JavaCC, available for free, is a parser generator. It provides a Java language extension for specifying a programming language's grammar. JavaCC was developed initially by Sun Microsystems, but it's now maintained by MetaMata. Like any decent programming tool, JavaCC was actually used to specify the grammar of the JavaCC input format.

Moreover, JavaCC allows us to define grammars in a fashion similar to EBNF, making it easy to translate EBNF grammars into the JavaCC format. Further, JavaCC is the most popular parser generator for Java, with a host of predefined JavaCC grammars available to use as a starting point.

Developing a simple calculator

We now revisit our little arithmetic language to build a simple command-line calculator in Java using JavaCC. First, we have to translate the EBNF grammar into JavaCC format and save it in the file `Arithmetic.jj`:

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

```

PARSER_BEGIN(Arithmetic)

public class Arithmetic
{
}

PARSER_END(Arithmetic)

SKIP :
{
  " "
|  "\t"
|  "\n"
}

TOKEN:
{
  < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
|  < DIGIT: ["0"- "9"] >
}

double expr():
{
}
{
  term() ( "+" expr() | "-" expr() )*
}

double term():
{
}
{
  unary() ( "*" term() | "/" term() )*
}

double unary():
{
}
{
  "-" element() | element()
}

double element():
{
}
{
  <NUMBER> | "(" expr() ")"
}

```

The code above should give you an idea on how to specify a grammar for JavaCC. The options section at the top specifies a set of options for that grammar. We specify a lookahead of 2. Additional options control JavaCC's debugging features and more. Those options can alternatively be specified on the JavaCC command line.

The `PARSER_BEGIN` clause specifies that the parser class definition follows. JavaCC generates a single Java class for each parser. We call the parser class `Arithmetic`. For now, we require only an empty class definition; JavaCC will add parsing-related declarations to it later. We end the class definition with the `PARSER_END` clause.

The `SKIP` section identifies the characters we want to skip. In our case, those are the white-space characters. Next, we define the tokens of our language in the `TOKEN` section. We define numbers and digits as tokens. Note that JavaCC differentiates between definitions for tokens and definitions for other production rules, which differs from EBNF. The `SKIP` and `TOKEN` sections specify this grammar's lexical analysis.

Next, we define the production rule for `expr`, the top-level grammar element. Notice how that definition markedly differs from the definition of `expr` in EBNF. What's happening? Well, it turns out that the EBNF definition above is ambiguous, as it allows multiple representations of the same program. For example, let us examine the expression `1+2*3`. We can match `1+2` into an `expr` yielding `expr*3`, as in Figure 1.

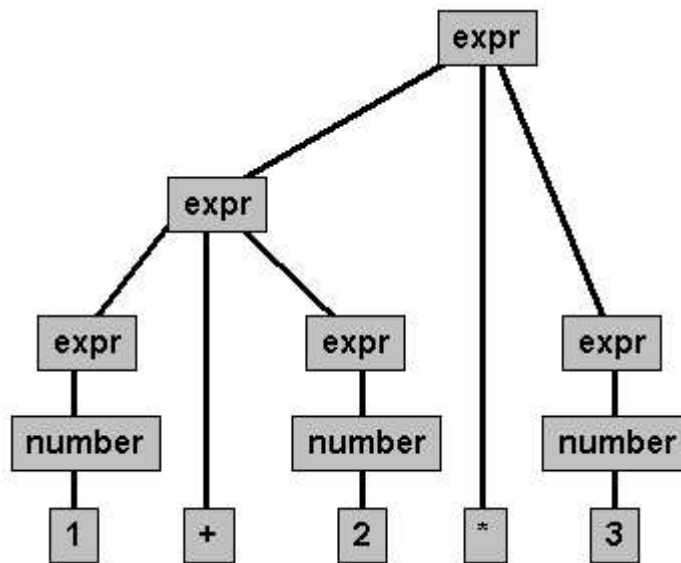


Figure 1. EBNF parse tree of 1+2*3

Or, alternatively, we could first match $2*3$ into an `expr` resulting in $1+expr$, as shown in Figure 2.

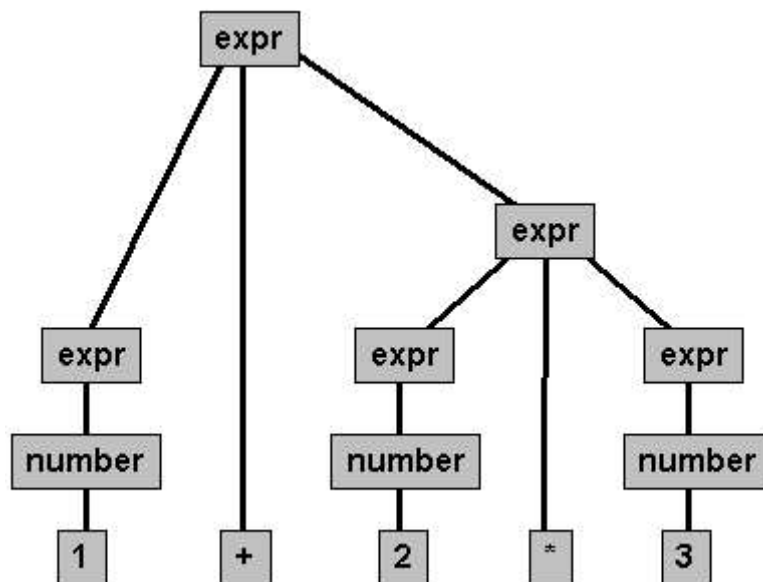


Figure 2. Alternative EBNF parse tree of 1+2*3

With JavaCC, we have to specify the grammar rules unambiguously. As a result, we break out the definition of `expr` into three production rules, defining the grammar elements `expr`, `term`, `unary`, and `element`. Now, the expression $1+2*3$ is parsed as shown in Figure 3.

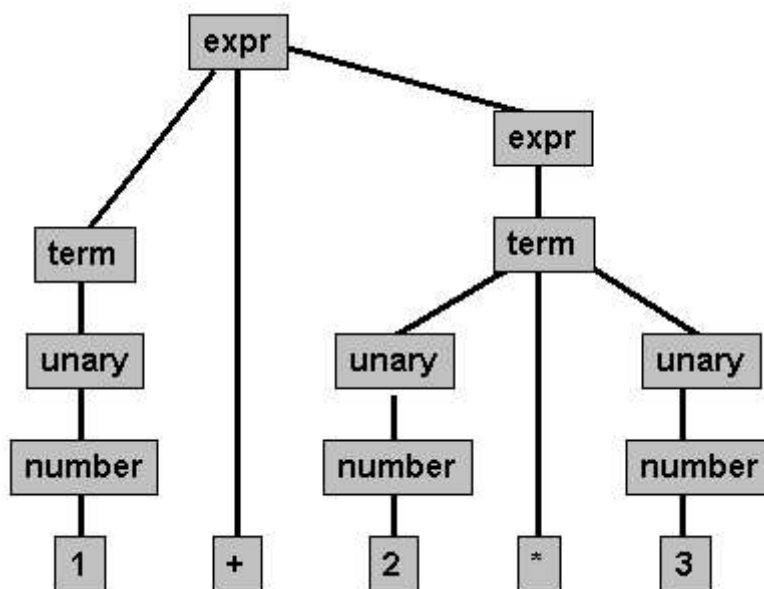


Figure 3. Parse tree of 1+2*3

From the command line we can run JavaCC to check our grammar:

```

javacc Arithmetic.jj
Java Compiler Compiler Version 1.1 (Parser Generator)
Copyright (c) 1996-1999 Sun Microsystems, Inc.
Copyright (c) 1997-1999 Metamata, Inc.
(type "javacc" with no arguments for help)
Reading from file Arithmetic.jj . . .
Warning: Lookahead adequacy checking not being performed since option LOOKAHEAD
is more than 1. Set option FORCE_LA_CHECK to true to force checking.
Parser generated with 0 errors and 1 warnings.
  
```

The following checks our grammar definition for problems and generates a set of Java source files:

```

TokenMgrError.java
ParseException.java
Token.java
ASCII_CharStream.java
Arithmetic.java
ArithmeticConstants.java
ArithmeticTokenManager.java
  
```

Together these files implement the parser in Java. You can invoke this parser by instantiating an instance of the class:

Arithmetic

```

public class Arithmetic implements ArithmeticConstants
{
    public Arithmetic(java.io.InputStream stream) { ... }
    public Arithmetic(java.io.Reader stream) { ... }
    public Arithmetic(ArithmeticTokenManager tm) { ... }

    static final public double expr() throws ParseException { ... }
    static final public double term() throws ParseException { ... }
    static final public double unary() throws ParseException { ... }
    static final public double element() throws ParseException { ... }
}
  
```

```

static public void ReInit(java.io.InputStream stream) { ... }
static public void ReInit(java.io.Reader stream) { ... }
public void ReInit(ArithmeticTokenManager tm) { ... }

static final public Token getNextToken() { ... }
static final public Token getToken(int index) { ... }

static final public ParseException generateParseException() { ... }

static final public void enable_tracing() { ... }
static final public void disable_tracing() { ... }
}

```

If you wanted to use this parser, you must create an instance using one of the constructors. The constructors allow you to pass in either an `InputStream`, a `Reader`, or an `ArithmeticTokenManager` as the source of the program source code. Next, you specify the main grammar element of your language, for example:

```

Arithmetic parser = new Arithmetic(System.in);
parser.expr();

```

However, nothing much happens yet because in `Arithmetic.jj` we've only defined the grammar rules. We have not yet added the code necessary to perform the calculations. To do so, we add appropriate actions to the grammar rules. `Calculator.jj` contains the complete calculator, including actions:

```

options
{
  LOOKAHEAD=2;
}

PARSER_BEGIN(Calculator)

public class Calculator
{
  public static void main(String args[]) throws ParseException
  {
    Calculator parser = new Calculator(System.in);
    while (true)
    {
      parser.parseOneLine();
    }
  }
}

PARSER_END(Calculator)

SKIP :
{
  " "
| "\r"
| "\t"
}

TOKEN:
{
  < NUMBER: (<DIGIT>)+ ( "." (<DIGIT>)+ )? >
| < DIGIT: ["0"-"9"] >
| < EOL: "\n" >
}

void parseOneLine():
{
  double a;
}
{
  a=expr() <EOL>    { System.out.println(a); }
| <EOL>
| <EOF>            { System.exit(-1); }
}

```

```

double expr():
{
    double a;
    double b;
}
{
    a=term()
    (
        "+" b=expr() { a += b; }
    | "-" b=expr() { a -= b; }
    )*
        { return a; }
}

double term():
{
    double a;
    double b;
}
{
    a=unary()
    (
        "*" b=term() { a *= b; }
    | "/" b=term() { a /= b; }
    )*
        { return a; }
}

double unary():
{
    double a;
}
{
    "-" a=element() { return -a; }
| a=element() { return a; }
}

double element():
{
    Token t;
    double a;
}
{
    t=<NUMBER> { return Double.parseDouble(t.toString()); }
| "(" a=expr() ")" { return a; }
}

```

The main method first instantiates a parser object that reads from standard input and then calls `parseOneLine()` in an endless loop. The method `parseOneLine()` itself is defined by an additional grammar rule. That rule simply defines that we expect every expression on a line by itself, that it is OK to enter empty lines, and that we terminate the program if we reach the end of the file.

We have changed the return type of the original grammar elements to return `double`. We perform appropriate calculations right where we parse them and pass calculation results up the call tree. We also have transformed the grammar element definitions to store their results in local variables. For instance, `a=element()` parses an `element` and stores the result in the variable `a`. That enables us to use the results of parsed elements in the code of the actions on the right-hand side. Actions are blocks of Java code that execute when the associated grammar rule has found a match in the input stream.

Please note how little Java code we added to make the calculator fully functional. Moreover, adding additional functionality, such as built-in functions or even variables is easy.

Conclusion

JavaCC, a parser generator for Java, makes writing parsers for programming languages a snap. JavaCC provides a high-level notation for defining grammars, and a concise specification for grammars and associated actions. It's also easy to read. Especially useful for Java programmers are the Java grammars that are part of the JavaCC distribution. (The Java 1.2 grammar is a separate download.) Based on those grammars you can create your own set of nifty Java tools such as your own version of `javadoc`, a Java interpreter, or a Java printer. There really is no limit to what you can do.

JavaCC includes two additional tools. `JJDoc` automatically generates documentation in HTML for your grammar in a

fashion similar to javadoc. JJTree automatically generates actions that build a tree structure when parsing a program. It provides a framework based on the Visitor design pattern that allows you to traverse the parse tree in memory.



About the author

Oliver Enseling is a seasoned software architect. Programming has been his passion since he was 16. A Sun-certified Java developer, Oliver works on distributed systems development in the Twin Cities area of Minnesota.

Resources

- To download this article's source code in zip format, go to:
<http://www.javaworld.com/jw-12-2000/cooltools/jw-1229-cooltools.zip>
- For more great articles on Java tools, visit the **Development Tools** section of *JavaWorld's* Topical Index:
<http://www.javaworld.com/javaworld/topicalindex/jw-ti-tools.html>
- For a freewheeling discussion on programming theory, visit *JavaWorld's Programming Theory & Practice* discussion, moderated by the intrepid Allen Holub:
<http://forums.itworld.com/webx?14@@.ee6b806>
- Sign up for the *JavaWorld This Week* free weekly email newsletter and keep up with what's new at *JavaWorld* :
<http://www.idg.net/jw-subscribe>
- JavaCC homepage:
<http://www.experimentalstuff.com/Technologies/JavaCC>
- You can access an entire repository of predefined grammars at:
<http://www.cobase.cs.ucla.edu/pub/javacc/>

Advertisement: Support JavaWorld, click here!



[HOME](#) | [FEATURED TUTORIALS](#) | [COLUMNS](#) | [NEWS & REVIEWS](#) | [FORUM](#) | [JW RESOURCES](#) | [ABOUT JW](#) | [FEEDBACK](#)

Copyright © 2003 JavaWorld.com, an IDG company