Recursive Descent Parsing

Daniel F. Savarese

Revision History

2014-03-07

Andrew Duncan[2] cleverly noticed the productions in Example 2 left out the right-recursive terms even though the code examples parsed them. This error was carried over from the original May 2001 column and had gone unnoticed until today. I've updated the right-recursive grammar to include the missing terms. Thanks Andrew!

2012-05-19

Converted and edited 2001 version of article for Web.

Note

This article was published originally in the May 2001 issue of *Java Pro* with the title "Express Yourself." That column was especially popular. Over the years, I've run into code in programs copied from it verbatim. Please don't do that. Sample code is for learning, not production use—and you don't learn much from copying. Use what you learn to write your own original and improved code.

The techniques described in this article form the basis for the Vareos[3] Calculator application, a scientific calculator implemented in JavaScript and SVG. You can find an expanded version of the expression grammar from this article in the help page for Calculator.

Changes. I have updated the code to use Java features that didn't exist at the time. Specifically, the code now uses enumerations, generics, `CharSequence` instead of `String` for input, the so-called "enhanced for loop," and java.util.regex instead of jakarta-oro for regular expressions. Also, the exponentiation operator has changed from `**` to `^`.

XML Isn't Always the Answer

Even though XML provides a versatile information representation structure, it is not appropriate for use by all applications. Sometimes information is expressed more appropriately in a form as close as possible to its natural representation. For example, even though mathematical expressions can be encoded in XML, it is more natural to define a grammar for parsing mathematical expressions if you are writing a calculator or spreadsheet program.

When XML isn't the right tool, you must assume the burden of parsing and interpreting information, rather than use one of many available XML parsers.[4] Even with an XML parser, you must interpret the meaning of what has been parsed relative to your application's semantics. Before you attempt to write a parser, you need to understand the structure of the information you are working with. The information may be passive data, requiring conversion into program data structures, or it may be active data (e.g., a spreadsheet macro), encoding a set of instructions requiring interpretation. In either case, the information must be translated into another form.

Context-Free Grammars

An information stream that a program parses can be thought of as a language, and the specification of its structure as a syntax. You can define the syntax of many languages using a context-free grammar. Context-free grammars were first applied to describing computer languages when John Backus drafted the Algol 60 specification. As a historical note, Algol is the origin of many of the syntactic structures present in the C language and its syntactic

[1] https://www.savarese.org/

[2] http://www.andrewduncan.ws

[3] http://www.vareos.com/

[4] At the time the original article was written, XML was being overused and misused for every data-representation need. Today, programmers aren't as wedded to XML as back then (e.g., JSON has become widely accepted), making the introductory paragraphs less apropos of the times.

brethren, including Java. The notation Backus used later became known as Backus-Naur Form (BNF) in honor of both Backus's and Naur's contributions to the Algol 60 report.

BNF defines a set of productions consisting of a non-terminal followed by a combination of terminals and non-terminals. A terminal is an atomic token, like a parenthesis. A non-terminal is a language construct that expands ultimately into a series of tokens, such as a mathematical expression.

Example 1. A Left-recursive Grammar for Simple Mathematical Expressions

Non-terminals and Operators

```
[1]       expression ::= expression + term | expression - term | term
[2]             term ::= term * exponent | term / exponent | exponent
[3]         exponent ::= exponent ^ factor | factor
[4]           factor ::= number | ( expression )
```

Terminals

```
[1]           number ::= [0-9]+(\.[0-9]+)?([eE][+-]?[0-9]+)?
```

The preceding grammar defines a syntax for simple mathematical expressions, including addition, subtraction, multiplication, division, exponentiation, and parenthetical grouping. The ::= symbol separates the left side of the production from the right side. The | symbol separates possible expansions of a production. In the example grammar, operators of equal precedence occur in the same production. Operators of lower precedence, such as addition and subtraction, occur in productions that are expanded first. Productions are expanded beginning with a non-terminal designated as the start symbol, which is, by convention, defined by the first production—in this case *expression*. The *number* terminal expands to a regular expression that matches number literals.[5]

The presented grammar has one problem that makes it difficult to use for parsing: The recursion present in the productions for *expression* and *term* is left-recursive. A parser has no way of predicting which production to apply at any given stage, making it possible to enter an infinite loop. For a parser to predict which production to apply, it must be able to choose unambiguously a production based on an input token.

Left-recursive productions of the form

```
[1]                A ::= Aa | B
```

can be transformed into right-recursive productions of the following form:

```
[1]                A ::= BR
[2]                R ::= aR |
```

The empty expansion after the bar in the second production represents the empty set. Using this transformation rule, the example grammar can be transformed into the following right-recursive grammar:

Example 2. A Right-recursive Grammar for Simple Mathematical Expressions

```
[1]       expression ::= term moreterms
[2]        moreterms ::= + term moreterms | - term moreterms |
[3]             term ::= exponent moreexponents
[4]    moreexponents ::= * exponent moreexponents | / exponent moreexponents |
[5]         exponent ::= factor morefactors
[6]      morefactors ::= ^ factor morefactors |
[7]           factor ::= number | ( expression )
```

[5]Regular expressions are used to extract terminal tokens from the input stream. The regular expression for *number* doesn't match negative numbers. Negative numbers can be handled by adding a unary negation operator to the grammar.

Tokenization

The right-recursive grammar is well-suited to predictive parsing because each input token indicates unambiguously which production should be applied. Before parsing an expression with this grammar, the expression's constituent elements must be identified. This process is called tokenization: the act of converting an input stream into a set of tokens.

Figure 1 presents a `Tokenizer` class for performing relatively general tokenization based on a set of regular expressions. `setInput()` defines the input for `Tokenizer`. The `Tokenizer` constructor initializes a list of `TokenType` (see Figure 8) instances, each of which associates a lexeme with a token. A lexeme is an instance of a token in the input. For example, both + and − are arithmetic operators of equal precedence and therefore represent the same type of token, but each is a different lexeme.

Regular expressions are a relatively compact and flexible way of denoting tokens; they are a standard part of lexical analysis tools such as Lex. Even though regular expressions may be overkill for tokenizing mathematical expressions, I didn't want to hard code lexemes into `Tokenizer`. I wanted to make it easy to represent tokens such as floating point numbers, which would otherwise require a separate grammar definition, placing their extraction in the parsing instead of the tokenization process. Therefore, the lexemes corresponding to each `TokenType` are encoded in a regular expression stored in each `TokenType` instance.

Figure 1. Tokenizer Class

```
package example;

import java.util.Collection;
import java.util.Iterator;
import java.util.ArrayList;
import java.util.regex.Pattern;
import java.util.regex.Matcher;

/**
 * A reasonably generic tokenizer for tokens that can be
 * defined with non-conflicting regular expressions.
 */
public class Tokenizer {
  private ArrayList<TokenType> tokenList;
  private Pattern separator;
  private Matcher matcher;

  public static final String DEFAULT_SEPARATOR = "\\s+";

  public Tokenizer(Collection<TokenType> token_types, String separator) {
    tokenList = new ArrayList<TokenType>(token_types);
    this.separator = Pattern.compile(separator);
    matcher = this.separator.matcher("");
  }

  public Tokenizer(Collection<TokenType> token_types) {
    this(token_types, DEFAULT_SEPARATOR);
  }

  public void setInput(CharSequence input) {
    matcher.reset(input);
  }

  /**
   * Returns the next token in the input.  If at end of input
   * returns null.  If no token is found, throws an exception.
   */
  public Token nextToken() throws TokenNotFoundException {
    Iterator<TokenType> it;

    // Skip over any separators.
    matcher.usePattern(separator);
    if(matcher.lookingAt()) {
      matcher.region(matcher.end(), matcher.regionEnd());
    }

    it = tokenList.iterator();

    while(it.hasNext()) {
      TokenType token_type = it.next();

      matcher.usePattern(token_type.getPattern());

      if(matcher.lookingAt()) {
        Token token = new Token(token_type.getType(), matcher.group());
        matcher.region(matcher.end(), matcher.regionEnd());
        return token;
      }
    }

    if(matcher.hitEnd()) {
      return null;
    }

    throw new TokenNotFoundException();
  }
}
```

`Tokenizer` extracts tokens from the input stream one at a time via the `nextToken()` method. First, the method skips over any token separators—by default, whitespace. Then it attempts to match each token type pattern, starting from the current input offset, until it finds a match. It is possible to implement the method more efficiently, so that only one match attempt needs to be attempted per method call. Doing so, however, would make the method harder to understand as an example.

The `Token` class in Figure 2 represents a token as a token value and a token type. The `MathTokenType` class in Figure 3 is both an enumeration of token types—each token type stores a corresponding regular expression pattern matching instances of the type—and a container for lexeme definitions. The `MathOperator` class in Figure 4 defines an enumeration of operators, a mapping between lexemes and operators, as well as an evaluation function for each operator.

Figure 2. Token Class

```
package example;

public class Token {
  private Object type;
  private Object value;

  public Token(Object type, Object value) {
    this.type  = type;
    this.value = value;
  }

  public Object getValue() { return value; }

  public Object getType()  { return type;  }
}
```

Figure 3. MathTokenType Class

```
package example;

import java.util.regex.Pattern;

public enum MathTokenType implements TokenType {
  // Token types
  TOK_OPEN_PAREN("\\("),
  TOK_CLOSE_PAREN("\\)"),
  TOK_ADDITIVE("[-+]"),
  TOK_MULTIPLICATIVE("[*/]"),
  TOK_EXPONENT("\\^"),
  TOK_NUMBER("\\d+(?:\\.\\d+)?(?:[eE][+-]?\\d+)?");

  // Lexemes
  public static final String LEX_ADD      = "+";
  public static final String LEX_SUBTRACT = "-";
  public static final String LEX_MULTIPLY = "*";
  public static final String LEX_DIVIDE   = "/";
  public static final String LEX_EXPONENT = "^";
  public static final String LEX_OPEN_PAREN  = "(";
  public static final String LEX_CLOSE_PAREN = ")";

  private Pattern pattern;

  MathTokenType(String pattern) {
    this.pattern = Pattern.compile(pattern);
  }

  public Object getType() {
    return this;
  }

  public Pattern getPattern() {
    return this.pattern;
  }
}
```

Figure 4. MathOperator Class

```
package example;

import java.util.HashMap;

public enum MathOperator {
  ADD      { double evaluate(double a, double b) { return a + b; } },
  SUBTRACT { double evaluate(double a, double b) { return a - b; } },
  MULTIPLY { double evaluate(double a, double b) { return a * b; } },
  DIVIDE   { double evaluate(double a, double b) { return a / b; } },
  EXPONENT { double evaluate(double a, double b) { return Math.pow(a, b); } };

  static final HashMap<String, MathOperator> operators;

  static {
    operators = new HashMap<String, MathOperator>();

    operators.put(MathTokenType.LEX_ADD, ADD);
    operators.put(MathTokenType.LEX_SUBTRACT, SUBTRACT);
    operators.put(MathTokenType.LEX_MULTIPLY, MULTIPLY);
    operators.put(MathTokenType.LEX_DIVIDE, DIVIDE);
    operators.put(MathTokenType.LEX_EXPONENT, EXPONENT);
  }

  public static final MathOperator getOperator(Object token) {
    return (MathOperator)operators.get(token);
  }

  abstract double evaluate(double a, double b);
}
```

Parsing and Evaluation

Figure 5 shows how to put the tokenization classes together to parse an expression and convert to an intermediate representation for later evaluation. Even though it is possible to compute the value of an expression as part of the parsing process, I wanted to demonstrate the separation between parsing and evaluation present in systems such as scripting languages, which compile a program into an intermediate byte code that is interpreted later.

Figure 5. MathParser Class

```
package example;

import java.util.Arrays;
import java.util.LinkedList;

/**
 * Parses the following grammar and translates the expression to
 * a postfix stack for later evaluation:
 *  expr -> term moreterms | term
 *  moreterms -> + term | - term
 *  term -> exponent moreexponents | exponent
 *  moreexponents -> * exponent | / exponent
 *  exponent -> factor morefactors | factor
 *  morefactors -> ^ factor
 *  factor -> number | ( expr )
 */
public final class MathParser {
  Tokenizer tokenizer;
  transient LinkedList<Object> stack;
  transient Token lookahead;

  void match(String token) throws ParseException {
    if(token == null || lookahead.getValue().equals(token)) {
      try {
        lookahead = tokenizer.nextToken();
      } catch(TokenNotFoundException e) {
        throw new ParseException(e);
      }
    } else {
      throw new ParseException();
    }
  }

  void parseNumber() throws ParseException {
    double number;

    if(lookahead == null) {
      throw new ParseException();
    }

    try {
      number = Double.parseDouble((String)lookahead.getValue());
      stack.push(new Double(number));
    } catch(NumberFormatException e) {
      throw new ParseException(e);
    }
    match(null);
  }


  void parseFactor() throws ParseException {
    if(lookahead == null) {
      throw new ParseException();
    }

    if(lookahead.getType() == MathTokenType.TOK_OPEN_PAREN) {
      match(MathTokenType.LEX_OPEN_PAREN);
      parseExpression();
      match(MathTokenType.LEX_CLOSE_PAREN);
    } else {
      parseNumber();
    }
  }

  void parseExponent() throws ParseException {
    Token token;

    parseFactor();

    // parse morefactors
    while(lookahead != null) {
      if(lookahead.getType() == MathTokenType.TOK_EXPONENT) {
        token = lookahead;
        match((String)lookahead.getValue());
        parseFactor();
        stack.push(MathOperator.getOperator((String)token.getValue()));
      } else {
        break;
      }
    }
  }

  void parseTerm() throws ParseException {
    Token token;

    parseExponent();

    // parse moreexponents
    while(lookahead != null) {
      if(lookahead.getType() == MathTokenType.TOK_MULTIPLICATIVE) {
        token = lookahead;
        match((String)lookahead.getValue());
        parseExponent();
        stack.push(MathOperator.getOperator((String)token.getValue()));
      } else {
        break;
      }
    }
  }

  void parseExpression() throws ParseException {
    Token token;

    parseTerm();

    // parse moreterms
    while(lookahead != null) {
      if(lookahead.getType() == MathTokenType.TOK_ADDITIVE) {
        token = lookahead;
        match((String)lookahead.getValue());
        parseTerm();
        stack.push(MathOperator.getOperator((String)token.getValue()));
      } else {
        break;
      }
    }
  }

  public MathParser() {
    tokenizer = new Tokenizer(Arrays.asList((TokenType[])MathTokenType.values()));
  }

  public LinkedList<Object> parse(String input) throws ParseException {
    tokenizer.setInput(input);

    match(null);

    if(lookahead == null) {
      throw new ParseException();
    }

    stack = new LinkedList<Object>();
    parseExpression();

    if(lookahead != null) {
      throw new ParseException();
    }

    return stack;
  }
}
```

MathParser applies recursive descent parsing to convert an infix mathematical expression to a postfix expression stored in a stack. Postfix expressions have only one possible interpretation, making them easy to evaluate with MathEvaluator (see Figure 6). MathEvaluator does not attempt to be efficient; instead, it demonstrates how postfix expressions can be evaluated easily. Operators are popped off the stack and their operands are evaluated before applying the operator and pushing the result back onto the stack.

Figure 6. MathEvaluator Class

```
package example;

import java.util.LinkedList;

public final class MathEvaluator {
  transient LinkedList<Object> stack;

  void evaluate() {
    Object obj = null;
    Double d1, d2;

    if(stack.size() > 1) {
      obj = stack.pop();

      if(obj instanceof MathOperator) {
        MathOperator op = (MathOperator)obj;
        evaluate();
        d2 = (Double)stack.pop();
        d1 = (Double)stack.pop();
        stack.push(new Double(op.evaluate(d1.doubleValue(), d2.doubleValue())));
      } else {
        evaluate();
        stack.push(obj);
      }
    }
  }

  public double evaluate(LinkedList<Object> operations) {
    Double result;
    stack = new LinkedList<Object>(operations);
    evaluate();
    result = (Double)stack.pop();
    return result.doubleValue();
  }
}
```

The recursive descent parsing algorithm used by MathParser is a form of top-down parsing. Top-down parsing applies productions to its input, starting with the start symbol and working its way down the chain of productions, creating a parse tree defined by the sequence of recursive non-terminal expansions. The tree is such that the parsing starts at the top and works its way down to the leaves, where terminal symbols (in our example, numbers) reside, terminating recursion.

Recursive descent parsers are straightforward to implement once you have defined a right-recursive grammar. All you have to do is write a function for each production. The functions mirror exactly the productions; they decide how to expand a non-terminal based on a lookahead token. The lookahead token is provided by `Tokenizer`. For example, `parseExpression()` directly calls `parseTerm()` because *term* is the first non-terminal in the production. Rather than write a separate function for *moreterms*, I've inlined the function. If the lookahead token is null, *moreterms* is expanded to the empty set; if it is an additive operator, it is expanded based on the presence of the operator. Eventually, either an unexpected token will be encountered, causing a `ParseException` (see Figure 10), or the expression will be parsed successfully.

If you were writing a calculator program with partial result evaluation, you would evaluate the expression while you were parsing it. `MathParser` translates the infix expression to a postfix representation for later evaluation, a technique more appropriate for a command-line calculator such as the POSIX **bc** command. `MathToken` translates the expression by pushing operators onto a stack after a non-terminal has been expanded, and pushing numbers onto the stack when they are encountered at the leaves of the parse tree. The expression `1+2*3` becomes `1 2 3 * +` and `1-2+3` becomes `1 2 - 3 +`. If a LISP-like prefix notation were the target of the translation, operators would be pushed onto the stack before a non-terminal was expanded, converting `1+2*3` into `+ 1 * 2 3`.

The `ParseExample` driver program in Figure 7 combines `MathParser` and `MathEvaluator` to calculate mathematical expressions given on the command line. A sample run verifies that operator precedence is enforced properly:

```
java example.ParseExample '2+(2^4*(7+2^6))'
Expression: 2+(2^4*(7+2^6))
Result: 1138.0
```

Figure 7. ParseExample Class

```
package example;

import java.util.LinkedList;

public final class ParseExample {
  public static final void main(String[] args) {
    MathParser parser;
    MathEvaluator evaluator;
    LinkedList<Object> result;

    if(args.length < 1) {
      System.err.println("usage: ParseExample expr1 expr2 ...");
      System.exit(1);
    }

    try {
      parser = new MathParser();
      evaluator = new MathEvaluator();

      for(String expression : args) {
        result = parser.parse(expression);
        System.out.println("Expression: " + expression);
        System.out.println("Result: " + evaluator.evaluate(result));
      }
    } catch(ParseException e) {
      throw new RuntimeException("Parse error.", e);
    }
  }
}
```

Parser Generators

For complicated grammars, writing your own recursive descent parser can require a lot of work. In addition, not all grammars can be transformed for recursive descent parsing. Complex inputs can create a lot of recursion and make you run out of stack space. There exist more efficient ways to parse languages based on finite state machines and transition tables. For anything but the simplest grammars, you will want to use a compiler to generate a parser for you, provided you define the grammar. Perhaps the most well-known tool for generating parsers in C is YACC (Yet Another Compiler Compiler). Similar tools, such as JavaCC and ANTLR,[6] are available for Java. They are both able to parse a class of grammars called *LL(k)* and differ fundamentally from YACC in how they generate parsers.

I've glossed over many details. Studying the sample code should fill in the gaps. For a more detailed treatment of the topic, you may wish to read *Compilers: Principles, Techniques, and Tools* by Aho, Sethi, and Ullman (Addison-Wesley, 1986).

Support Classes

Figure 8. TokenType Interface

```
package example;

public interface TokenType {
  public Object getType();
  public java.util.regex.Pattern getPattern();
}
```

Figure 9. TokenNotFoundException Class

```
package example;

public class TokenNotFoundException extends Exception {
  public TokenNotFoundException() { }

  public TokenNotFoundException(String message) {
    super(message);
  }

  public TokenNotFoundException(String message, Throwable cause) {
    super(message, cause);
  }

  public TokenNotFoundException(Throwable cause) {
    super(cause);
  }
}
```

Figure 10. ParseException Class

```
package example;

public class ParseException extends Exception {
  public ParseException() { }

  public ParseException(String message) {
    super(message);
  }

  public ParseException(String message, Throwable cause) {
    super(message, cause);
  }

  public ParseException(Throwable cause) {
    super(cause);
  }
}
```

[6]In 2001, these were popular tools. Other tools may be more popular today.