# AN INTERACTIVE PARSER GENERATOR FOR CONTEXT-FREE GRAMMARS

Gabriel J. Ferrer
Department of Mathematics and Computer Science
Hendrix College
Conway, AR 72032
(501) 450-3879
ferrer@hendrix.edu

## ABSTRACT

This paper describes a parser generator that accepts arbitrary context-free grammars. It generates a parser using the Earley algorithm [1]. It enables the user to develop, edit, and test a grammar in an interactive graphical environment. This GUI visualizes both the operation of the Earley algorithm as well as the generated parse trees. The generated parsers are saved as fully-functional Java source files, ready to be incorporated into an application. These Java programs can be reloaded into the GUI for further editing of the grammar. Employing this parser generator in a sophomore-level software development course enables students to become proficient in writing a parser with two days of lecture and one assignment.

## INTRODUCTION

In our college's Computer Science program, students are expected to learn to process text input using both regular expressions and parser generators. While regular expressions are sufficient for many purposes, they suffer from two intrinsic problems: It is difficult to introduce named abstractions, and it is not possible to represent recursive structures. For this reason, we require our students to learn how to process text input using a parser generator.

Parser generators are typically covered as part of a compiler construction course (e.g. [10] [11]). However, the number of courses we can require for the Computer Science major is constrained by Hendrix College's liberal arts curriculum; this makes it difficult for us to require (or offer) a compiler-construction course. We instead cover parsing in a sophomore-level programming course.

To write a parser by hand for any but the simplest of input languages makes for a tedious, highly bug-prone task, a task typically avoided in practice by using a parser generator. But parser generators themselves are associated with a number of practical problems, especially in the context of the limited time we have available for covering this topic:

1. In order to improve runtime efficiency, most parser generators (e.g., [3] [4] [5] [6]) accept only a subset of context-free grammars.
2. The parsing algorithms (e.g., LALR parsers [2][11]) often require considerable study to understand.
3. Many parser generators (e.g., [3] [4] [6]) require that input be preprocessed with a lexical analyzer.
4. When developing a grammar, it is difficult to get immediate feedback on the changes, as testing each alteration requires first generating a parser and running the compiler.
5. The API for interacting with the output of a parser generator is often intimidating, and disproportionate to what is needed to build a working parser.

In order to address these problems, we would like a parser generator to have the following features:

1. Accept an unrestricted context-free grammar.
2. Accept regular expressions as terminal symbols.
3. Employ an algorithm readily comprehensible by a 2nd-year computer science student.

4. Allow the programmer to interact with the grammar without an explicit code-generation step.
5. Provide an API that enables the programmer to write tree-walking code quickly and easily. In addition, the API should allow the grammar to change with as few changes as possible to the tree-walking code that depends upon it.

Furthermore, given the widespread use of automated unit testing (e.g. JUnit), it would be helpful for a parser generator to support automated unit testing as well.

This led us to create our own parser generator, Grambler, to meet our requirements. Grambler is an implementation of the Earley parsing algorithm. It allows the user to specify an arbitrary context-free grammar, and it will generate a Java class that corresponds to that grammar. The user may mix regular expressions into the context-free grammar as well, thus seamlessly incorporating lexical analysis into the parser.

Grambler also allows the user to interact with the grammar. Once a grammar is specified, the user can enter text and request that it be parsed. The GUI will then display a parse tree for the text using the grammar, and will also show the parse table generated by the Earley algorithm. The user can also generate JUnit tests to verify that alterations to the grammar preserve its ability to parse previously successful inputs.

This paper is organized as follows. First, the Earley algorithm is summarized, and some modifications we made to the algorithm are described. Next, each feature of the GUI is described, with an emphasis on the pedagogical role of each feature. We then describe the API for interacting with the parse trees that Grambler generates.

## IMPLEMENTING THE EARLEY PARSING ALGORITHM

The Earley algorithm is a dynamic-programming algorithm that builds a table of parses of input prefixes. The algorithm terminates when the top-level production has matched the entire input string. Each row of the table corresponds to one character position in the input string. (For an input string of length *n*, the first row is row zero, and the final row is row *n*.) Each row contains a list of every possible match between a production and the input string at that position.

Each possible match is called a *state*. A state is defined by the following parameters:

- Input position where this match started (the *origin* position)
- Current input position (i.e., the current row)
- A production from the grammar
- Next candidate symbol from that production

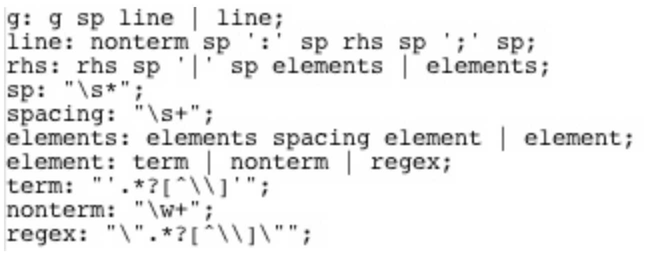
The algorithm starts by adding a state to row zero for each alternative of the top-level production. The first symbol from the production will be the next candidate symbol, and both the starting and current input positions will be zero. Then, for every state in every table row, a table update is performed.

The nature of a table update depends upon the type of the state. A state is *complete* if all production symbols have been matched. A state is *scanning* if the next candidate production symbol is a terminal symbol; if it is a non-terminal symbol, the state is *predicting*.

If the state is predicting, it will add a new state to the same table row for every right-hand alternative of the next candidate production symbol. If the state is scanning and the current character is a match for the terminal symbol, it will add a new state to the *next* table row, with an updated current-input position and an updated next-candidate symbol. If the state is complete, it will loop through all states in its origin position; if the completed state's production is the next-candidate symbol for an origin state, a new state derived from the origin state is inserted at the current table row, with an updated next-candidate symbol.

As our implementation of this algorithm permits both arbitrary-length fixed strings as well as regular expressions as terminal symbols, it was necessary to modify the update for scanning states. First, the length of the matching input substring is computed. This length is then added to the current character position to determine the row into which the new state is to be placed.

To demonstrate the input language for context-free grammars, Figure 1 gives as an example the grammar for that input language. Grambler is self-hosting; it generated its own parser.



```
g: g sp line | line;
line: nonterm sp ':' sp rhs sp ';' sp;
rhs: rhs sp '|' sp elements | elements;
sp: "\s*";
spacing: "\s+";
elements: elements spacing element | element;
element: term | nonterm | regex;
term: "'.*?[^\\]'";
nonterm: "\w+";
regex: "\".*?[^\\]\"";
```

*Figure 1: Grammar for grammars*

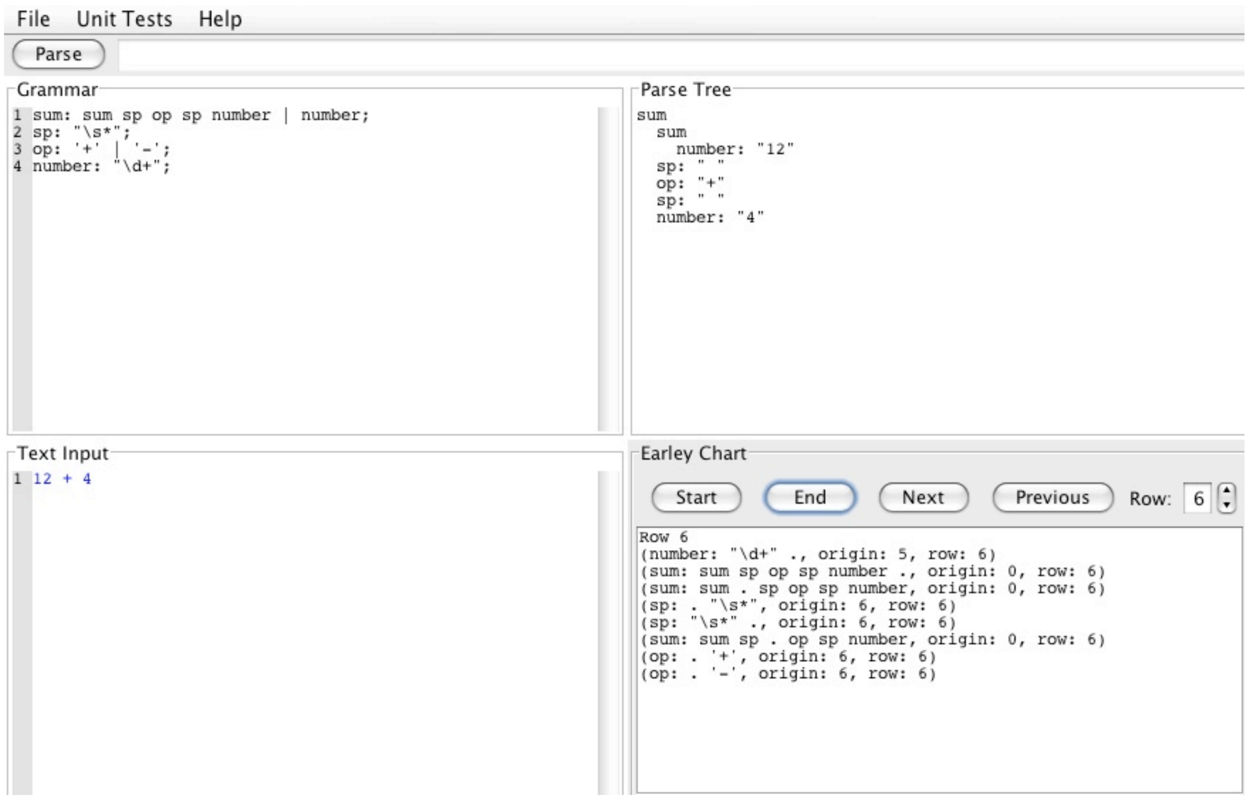
Each production is structured as follows. The left-hand side symbol is followed by a colon; alternatives are separated with a vertical line, and the production as a whole is terminated by a semicolon. The left-hand side of the first production is the start symbol for the grammar.

The right-hand elements are separated by spaces. Elements without quotation marks are nonterminals. Elements within single quotation marks are string literal terminals. Elements within double quotation marks are regular expression terminals. The string literal within each pair of double quotes is used to construct a java.util.regex.Pattern object for regular expression matching.

**GRAPHICAL USER INTERFACE**

The Grambler GUI gives the user the following options:

- Create/edit a grammar
- Export/import a grammar to/from a Java file
- Create/edit a text input for the grammar to process
- Open/save a text input or grammar as a text file
- Export a text input as a JUnit test
  - Check acceptance/rejection only
  - Check matching parse tree
- Parse the current text input using the current grammar
  - Determine whether the parse succeeded
  - View the parse tree resulting from the parse
  - View any Earley table row generated while parsing



```
sum: sum sp op sp number | number;
sp: "\s*";
op: '+' | '-';
number: "\d+";
```

```
sum
  sum
    number: "12"
  sp: " "
  op: "+"
  sp: " "
  number: "4"
```

```
12 + 4
```

```
Row 6
(number: "\d+" ., origin: 5, row: 6)
(sum: sum sp op sp number ., origin: 0, row: 6)
(sum: sum . sp op sp number, origin: 0, row: 6)
(sp: . "\s*", origin: 6, row: 6)
(sp: "\s*" ., origin: 6, row: 6)
(sum: sum sp . op sp number, origin: 0, row: 6)
(op: . '+', origin: 6, row: 6)
(op: . '-', origin: 6, row: 6)
```

*Figure 2: Graphical User Interface*

Figure 2 shows a screenshot of the user interface. The user creates a grammar in the upper-left area. The user then enters a text input to be parsed in the lower-left area. Once the grammar and text input are ready, the user clicks the Parse button. The upper-right area shows a parse tree. The lower-right area shows the table rows generated by the Earley algorithm.

The parse tree is presented based on a preorder traversal (similar to the visualization from [10]). Each row of the text output is a tree node. Each level of indentation indicates a level of tree depth. The example above contains a tree with three levels and seven total nodes.

The Earley chart is visualized one row at a time. The user can select a row directly, jump to the start or end, or iterate among consecutive rows. As some rows may contain no states, the Next and Previous buttons jump only between rows that do have states. Each state is described by its production, a period (".") before the next candidate symbol, and the current and origin input positions for the state. The purpose of this aspect of the GUI is to enable a user who is puzzled as to why a parse is failing to inspect the chart to find out precisely how far into the input the parser got. From there, the user can inspect the states to figure out which productions were attempted, and from there infer which productions failed to advance.

The File menu allows the user to export the grammar to a Java file that, when compiled and executed, will parse the language the grammar specifies. The user can also import the grammar from a Java file in the format it generates.

The Unit Tests menu allows the user to generate two kinds of unit tests: acceptance checks and tree checks. An acceptance check will generate a unit test that checks to see if the error status of the generated tree is identical to the error status of parsing the current text input.

A tree check ensures that the generated tree is identical to that in view on the GUI. These unit tests are appended to a JUnit-compatible file that is automatically generated, based on the name of the Java file used for saving the grammar.

## APPLICATION PROGRAMMER INTERFACE

Once a grammar is complete, Grambler will generate Java code corresponding to it. Specifically, it will generate a class that extends the Grammar class from the Grambler API. Objects of the grammar class have a parse() method that takes a String parameter and returns a Tree object corresponding to the concrete syntax tree for the input parameter. In the event of a syntax error, a Tree object is still returned containing the error information.

A successful tree walk requires the ability to:

1. Determine the grammar label corresponding to each tree node;
2. Retrieve the children of each interior node; and
3. Reconstruct the original text input corresponding to a node.

To achieve these goals, the Tree class provides the following methods:

1. The getName() method returns the name of the left-hand side element corresponding to this Tree node.
2. The hasNamed() method determines whether a child with a given name is present for this tree Node. The getNamed() method retrieves the Tree object corresponding to a given name. If more than one child shares the same name, an additional integer parameter can be supplied to resolve the ambiguity.
3. The toString() method returns the input substring corresponding to this Tree object.

Syntax errors are flagged by the isError() method. If a syntax error is present anywhere within the tree, isError() will be true for all of the ancestor nodes of the error, including the root node. The getErrorMessage() method returns a String giving the the line number of the error as well as a prefix of the line that was successfully matched immediately prior to the error.

## RELATED WORK

The parser generator DParser [7] directly inspired the grammar syntax employed in Grambler. While DParser addresses most of the problems with existing parser generators listed in the introduction, is not suitable for our pedagogical goals for two reasons. First, being a traditional command-line parser generator, it lacks the level of interactivity we require. Second, the Earley algorithm was easier for us to conceptualize as an interactive visualization, in comparison to the Tomita algorithm [9] DParser employs.

In both VAST [12] and Tree-Viewer [10], syntax tree visualizations were created for educational purposes. We prefer the text-oriented preorder-traversal approach from [10], as nearly a full line of text is available for describing each node. In the graphical-box approach in [12], much less information can be displayed for each node.

Grammar Editor [8] is an interactive editor for context-free grammars employing the CYK algorithm. As with Grambler, a user enters a grammar and text to be parsed, and the program will display a parse tree. It cannot, however, create a parser generator that can be incorporated into a user program.

## CONCLUSION

The Grambler parser generator has been successful in simplifying the presentation of parsing to students in a third-semester programming course. Every project in the class had a working parser after two days of lecture and one assignment. We have been able to focus our valuable course time on higher-level issues involved with incorporating parsers into computer programs without being distracted by other issues such as obscure syntax and extended edit-compile-test cycles.

Grambler is freely available for download from http://code.google.com/p/grambler/.

## REFERENCES

[1] Earley, J., An efficient context-free parsing algorithm, *Communications of the ACM*, 13 (2): 94-102, 1970.

[2] Aho, A.V., Sethi, R., Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.

[3] Parr, T., ANTLR, http://www.antlr.org/, retrieved 5/16/12.

[4] JavaCC, https://javacc.dev.java.net/, retrieved 5/16/12.

[5] Grimm, R., Rats!, http://cs.nyu.edu/rgrimm/xtc/rats-intro.html, retrieved 5/16/12.

[6] Gagnon, E., SableCC, http://sablecc.org/, retrieved 5/16/12.

[7] Plevyak, J., DParser, http://dparser.sourceforge.net/, retrieved 5/16/12.

[8] Burch, C., Grammar Editor, http://ozark.hendrix.edu/~burch/proj/grammar/, retrieved 5/16/12.

[9] Tomita, M., An efficient context-free parsing algorithm for natural languages, In *Proceedings of the International Joint Conference on Artificial Intelligence*, pp. 756-764, 1985.

[10] Vegdahl, S., Using visualization tools to teach compiler design, *Journal of Computing Sciences in Colleges*, 16 (2), January 2001.

[11] Demaille, A., Levillain, R., Perrot, B., A set of tools to teach compiler construction, In *Proceedings of ITiCSE-08*, June 2008.

[12] Almeida-Martinez, F.J., Urguiza-Fuentes, J., Velazquez-Iturbide, J.A., VAST: Visualization of Abstract Syntax Trees within language processors courses, In *Proceedings of SoftVis '08,* September 2008.

[13] Wall, L., Apocalypse 5: Pattern Matching, http://perl6.org/archive/doc/design/apo/A05.html, retrieved 7/31/12.