SIMPLIFYING PARSER GENERATION

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

ABSTRACT

This paper describes Squirrel, a parser generator that uses a modification of the Packrat algorithm [7]. Squirrel is designed to avoid many of the complexities often associated with parser generators. Students in a CS3 course were able to learn to use Squirrel in about a week. These students were successful in using Squirrel to create parsers for subsets of the Logo programming language and HTML. Squirrel is distributed under the terms of the GNU Lesser General Public License, and it may be downloaded from http://ozark.hendrix.edu/~ferrer/software/squirrel/.

INTRODUCTION

The parsing of text input is an essential concept for computer science students to learn. Since our computer science program is situated in a liberal arts college in which the number of courses we may require is tightly constrained, we are not able to spend a large amount of time on this topic. In particular, we are unable to offer a course in compiler construction, wherein this topic is traditionally covered in depth. Hence, we are covering it in a third-semester applications development course, using Java.

There are two practical options for implementing parsers: Have the students write a recursive descent parser, or use a parser generator. To have students write a recursive descent parser for any but the simplest of input languages makes for a tedious, highly bug-prone task, a task most professionals avoid in practice by using a parser generator. Given the limited class time available for describing how a parser generator is to be used, it is important that it be as simple as possible.

Existing parser generators for Java often require the learning of a nontrivial input language [1, 2, 3, 4, 5, 6, 10], separate the parser from the scanner [3, 6], cannot handle left recursion [1, 2, 4, 6, 10], do not automate building abstract syntax trees [1, 3, 6], use the complex LALR parsing algorithm [5, 3] and can require an additional compilation step to generate the parser [1, 2, 3, 5, 6, 10]. As all of these issues introduce what we consider to be gratuitous complexity, we created a new parser generator to target the needs of our students.

Simplicity is achieved in the resulting parser generator (Squirrel) as follows. To minimize new syntax for students to learn, the parser generator is a single Java class. All grammar productions are specified using method calls. To eliminate an additional compilation step, the parser is generated automatically when the first string is sent to it for parsing. By using the Packrat algorithm [7], lexical analysis is seamlessly included. The observable behavior of the Packrat algorithm is close to that of a recursive descent parser, and is about as easy to understand. By automatically generating a syntax tree, certain common forms of left recursion can be handled by using tree rotations.

PACKRAT PARSERS

Packrat parsers [7] implement a variation of recursive-descent parsing in conjunction with top-down dynamic programming. Parse trees for substrings of the input are stored in a two-dimensional array. One dimension represents positions of characters in the input string. The other dimension represents symbols in the grammar. Each cell holds one of three possible types of values:

1. A parse tree representing a maximum-length parse from the input position that matches the symbol from the grammar;
2. A data structure representing a failure to match the symbol at that input position;
3. Nothing, if no attempt has been made to parse at that position with that symbol.

Parsing proceeds in a recursive-descent fashion from the start symbol. As each symbol is encountered, the parser attempts to look up a tree for that symbol at the current input position from the array. If nothing is found, parsing continues recursively.

For each nonterminal symbol, when a right-hand side with a successful match is encountered, no further right-hand alternatives will ever be considered. Hence, whether the parser will accept certain input strings as part of its language is highly sensitive to the ordering of the right-hand alternatives for each nonterminal's production.

Consider this production: <addExpr> ::= <num> | <addExpr> "+" <num>. For the input string "5 + 2", <number> will match "5" and <addExpr> will match on its first alternative, whereupon a Packrat parser will immediately terminate. This termination would have to be considered a failure, because there remains input that has not been matched. In order to match the entire input string, the order of the right-hand side must be reversed. That yields the production: <addExpr> ::= <addExpr> "+" <num> | <num>. This revised production successfully matches the entire string "5 + 2".

A packrat parser has both time and space complexity bounded above by O(ns), where n is the size of the input string and s is the number of symbols in the grammar. Given the assumption that the number of characters in an input string will be multiple orders of magnitude larger than the number of symbols in the grammar, these parsers can be considered to run in time that is linear in the size of the input string.

Packrat parsers are easily extended by lookahead operations that can compensate for the greedy selection of right-hand side alternatives. Both the "!" and "&" operators are included in Squirrel. In general, these parsers accept a large subset of context-free grammars, which have been dubbed "Parsing Expression Grammars" [8].

The "!" operator is followed by a production that is to be excluded. For example, a string in a programming language could be matched by the following productions: <str> ::= <quote> <strchars> <quote> and <strchars> ::= !<quote> <anychar>. This latter production will match any character that does not first match <quote>. The attempt to match <quote> does not consume any of the input; instead, it works as a filter that disallows matching inputs from further consideration. The "&" operator is similar, but rather than excluding its argument it mandates the inclusion of its argument.

INCORPORATING LEFT RECURSION

From a teaching standpoint, it is desirable that students learn to associate parse-tree structures with corresponding grammatical structures. Unfortunately, the semantics of Packrat parsing are a poor match for left recursive grammars. This follows from the

requirement that only a maximum-length match is permitted for a symbol at a given input position. Left recursion would imply multiple simultaneous matches of a symbol at the same input position.

Consider again the production: <addExpr> ::= <addExpr> "+" <num> | <num>. This type of grammar pattern is common when it is desired for the parser to output parse trees that reflect conventional operator precedence. Addition is traditionally evaluated in a left-to-right ordering. Hence, the leftmost computation should ideally appear at the maximum depth for the corresponding parse tree, while the rightmost computation should appear at depth one. A right-recursive version of this rule would produce a parse tree that corresponds to a right-to-left evaluation ordering.

Methods for factoring left recursion are well-known but complex to use and tricky to automate [9]. Hence, Squirrel was customized to recognize a subset of left-recursive grammars for which the transformation is straightforward.

We begin by transforming each left-recursive rule into a right-recursive rule that matches the same input. Once Squirrel has constructed a parse tree using the right-recursive version of the rule at a given position of the input, a left rotation of the parse tree is performed.

In general, left-recursive rules that are eligible for this transformation will have exactly two right-hand alternatives that match the following pattern:
<recursive> ::= <recursive> <sym1> <sym2> … <finalSymbol> | <finalSymbol>.

This pattern is transformed into the following equivalent right-recursive rule:
<recursive> ::= <finalSymbol> <sym1> <sym2> … <recursive> | <finalSymbol>.

The production: <addExpr> ::= <num> "+" <addExpr> |<num> is a right-recursive equivalent of our earlier example. It recognizes the same strings, but it parses them from the right instead of from the left. The input string "5 + 4 – 3 – 2" would be parsed as follows. The first parse node generated has children "5", "+", and an anticipated tree resulting from a recursive evaluation of <addExpr>. The string "4 – 3 – 2" remains to be parsed. The next node generated has children "4", "-", and another anticipated tree. The final node generated has children "3, "-", and "2". If a right-recursive tree happened to be the desired outcome, then the resulting parse tree (following a preorder traversal) would look like this:

<addExpr>
    5
    +
    <addExpr>
        4
        -
        <addExpr>
            3
            -
            2

However, when rebalancing for left recursion, a left rotation is performed as each subtree is completed. The final rebalanced tree looks like this:

<addExpr>
    <addExpr>
        <addExpr>
            5
            +
            4
        -
        3
    -
    2

It is important to note that incorporating left recursion by tree rebalancing is a technique that can be incorporated into any parsing algorithm. Nothing about the Packrat algorithm facilitates this; it follows from the automatic generation of parse trees.

SYNTAX SIMPLIFICATION

The incorporation of lexical analysis into the parsing algorithm helps reduce the necessary syntax to learn, because no additional syntax is needed for building a scanner. A different method name is used for adding tokens, but that is the only difference that the user must consider. Further simplification of the syntax arises from the decision to automatically generate syntax trees. Because no client code is required for generating tree nodes, no special grammar annotation is needed for this purpose.

By both constructing and invoking the parser from regular Java source code, the extra compilation step required by many parser generators is eliminated. Furthermore, the amount of new syntax to be learned is further reduced. The grammar itself is constructed by invoking methods on an object of the Squirrel class, using Java's standard method invocation syntax. All of the method parameters are strings. The first parameter of each method is the right-hand side of a nonterminal. The other parameters represent different alternatives for the left-hand side of the nonterminal.

There is a minimal amount of syntax that must be learned for encoding the productions within these strings. Nonterminal symbols are enclosed in angle brackets. Spaces separate the terminals and nonterminals in a row from each other. Square brackets surrounding a symbol denote that the symbol is optional. The “!” and “&” symbols are used to denote lookahead symbols, as discussed earlier.

The following code example illustrates the simplified syntax. The example begins with the creation of a Squirrel object. Next, the various productions are added. Then an input string is sent to the Squirrel object, and finally the resulting parse tree is captured and is ready for traversal. The parse tree data type is an inner class.

```
Squirrel parser = new Squirrel();
parser.addRuleStr("<expr>", "<expr> [<s>]  <op> [<s>] <num>", <num>”);
parser.addTokenStr(“<op>”, “+”, “-“);
parser.addTokenStr(“<num>”, “<digit> [<num>]”);
parser.addTokenStr(“<digit>”, “0”, “1”, “2”, “3”, “4”, “5”, “6”, “7”, “8”, “9”);
parser.addTokenStr(“<s>”, “<space> [<s>]”);
```

```
parser.addTokenStr("<space>", " ");
Squirrel.ParseTree tree = parser.parse("5 - 4 + 3 - 2");
```

The above grammar will recognize simple arithmetic expressions using addition and subtraction, with zero or more spaces between the arguments and each operator. Special token nonterminals are provided for matching single upper-case and/or lower-case letters, as well as single whitespace and alphanumeric characters.

The methods "addRuleStr()" and "addTokenStr()" are distinguished by the data structures that are generated when they are triggered. The "addRuleStr()" method will generate a parse tree node with a child for each matched right-hand side element. The "addTokenStr()" method generates a leaf containing the entire matched string.

When using a parser generator with students, it is important that they understand the algorithm the generated parser uses. For understanding a Packrat parser, students need to be familiar with recursion, arrays, trees, and grammars (including terminal and nonterminal symbols). Incorporating left recursion (as described earlier) also requires students to understand tree rotation. Given an understanding of these concepts, explaining the algorithm to students is very straightforward. The author was able to present the algorithm in two 50-minute lecture periods: The first was dedicated to recursive-descent parsing, while the second lecture explained the Packrat algorithm.

CLASSROOM EXPERIENCE

The learning objectives for parsing in our CS3 course are:

1. To understand how a grammar is designed for a particular input language.
2. To learn how to use a parser generator to generate a parser from a grammar.
3. To understand the restrictions on what a grammar can express.
4. To apply parsing to transform arbitrary-length inputs into tree-structured data.

The course consists of three large programming projects. Students work individually on the first two projects. The first project is a turtle graphics GUI with a Logo interpreter. The second project is a simple but functional graphical web browser. Students work in teams on the third project. Each team chooses its topic for the third project. Many different topics related to building large programs are covered, including design patterns, multithreading, human-computer interaction, and network programming.

Squirrel was used in teaching parsing in this course in the Fall 2006 semester. Fourteen students took the course. All of the students had previous experience with recursion, trees, and multidimensional arrays. None of the students had previous exposure to grammars. Students first completed a homework assignment in which they built parse trees by hand when given a grammar and an input string. Students were given five days to complete this assignment.

Once the hand-parsing assignment was complete, students were given two days to create a working parser for a small subset of the Logo programming language. Nine of the students were able to produce a working parser on time. The other five students were able to do so eventually, even though their submissions were late. They expanded their parsers to handle a much larger subset of the Logo language during a later assignment. They also used Squirrel in the web browser assignment to implement parsers for HTML.

It was our experience that Squirrel was helpful in meeting our learning objectives. Very little class time and project time needed to be spent in covering language syntax and

other issues irrelevant to our learning objectives. The instructor and students were able to spend a great deal of time discussing issues in grammar design and language restrictions.

For example, students learned that HTML is too complex a language to be expressed by a context-free grammar, so they wrote their parsers to discriminate between tags and non-tags and to parse tag contents. Squirrel proved to be efficient enough to parse large HTML documents very quickly, although occasionally it ran out of stack space due to the recursion depth. The problem was resolved by introducing an optimization to implement tail-recursive productions with iteration.

CONCLUSION

The Squirrel parser generator has been a big success in simplifying the presentation of parsing to students in a third-semester programming course. 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 cycles.

Future work will focus on ways in which Squirrel can be made even more useful to students. Robust error recovery would be helpful; currently, it is only able to report a single syntax error before quitting. When errors are encountered, it currently returns canned error messages. It would be preferable for it to return an error data structure that enabled students to generate customized error messages.

Because Java maintains a relatively shallow call stack by default, we are considering reengineering Squirrel to use bottom-up dynamic programming rather than top-down dynamic programming in order to eliminate recursion. This should provide a more comprehensive solution to the problem than our tail-recursion optimization.

The left recursion that Squirrel can handle has been excellent for parsing arithmetic expressions. It will be important to explore other situations in which left recursion is helpful, so as to determine how the parser's capability to handle left recursion might be most productively expanded.

REFERENCES

[1] Sun Microsystems, JavaCC, 2007, https://javacc.dev.java.net/, retrieved Jan.19, 2007.
[2] Parr, T., ANTLR, 2007, http://www.antlr.org/, retrieved Jan. 19, 2007.
[3] Hudson, S., CUP, 2007, http://www2.cs.tum.edu/projects/cup/, retrieved Jan.19, 2007.
[4] Cederburg, P., Grammatica, 2007, http://grammatica.percederberg.net/, retrieved Jan. 19, 2007.
[5] Gagnon, E.M., SableCC, 2007, http://sablecc.org/, retrieved Jan. 19, 2007.
[6] Parsers, Inc., SLK, 2007, http://home.earthlink.net/~slkpg/, retrieved Jan. 19, 2007.
[7] Ford, B., Packrat parsing: simple, powerful, lazy, linear time, in *Proceedings of the International Conference on Functional Programming*, October 2002.
[8] Ford, B., Parsing expression grammars: a recognition-based syntactic foundation, in *ACM SIGPLAN - SIGACT Symposium on Principles of Programming Languages (POPL),* January 2004.
[9] Aho, A.V., Sethi, R., Ullman, J.D., *Compilers: Principles, Techniques, and Tools*, Addison-Wesley, 1986.
[10] Grimm, R. Rats!, 2007, http://www.cs.nyu.edu/rgrimm/xtc/rats.html, retrieved Jan. 19, 2007.