CUPV - A Visualization Tool for Generated Parsers

Alan Kaplan and Denise Shoup
Department of Computer Science
Clemson University
Clemson, SC 29634-0974
{kaplan,shoup}@cs.clemson.edu

Abstract

Compiler projects frequently use parser generators to help students design and construct non-trivial translators. Unfortunately, the code and data structures produced by such generators, and hence the overall parser, can be difficult to understand and debug. In this paper, we present an extendible and flexible tool for visualizing the operation of generated parsers. The objective of this tool is to provide students with a deeper understanding of parsing algorithms, data structures and techniques.

1 Introduction

Compiler courses are among the most challenging, yet rewarding, courses in computer science curriculums, since they integrate techniques from almost every area in computer science, including language theory, data structures, operating systems and software engineering. Central to most compiler courses is a substantial project that requires the design and development of a non-trivial compiler. This project typically plays a significant role in helping students understand both compiler theory and practice.

To help students build a compiler, projects frequently use parser generator tools. In general, parser generator technology is extremely useful for rapidly creating reliable parsers for non-trivial programming language compilers. Tools based on this technology automatically generate parsers by translating a simple input specification of a programming language, based on a context-free grammar and associated semantic actions, into program source code (such as Java or C++). Parser generator tools exist for both LL[1] and LALR[2] class grammars and support development in a variety of programming languages including Ada, C, C++ and Java. Prominent examples include Yacc [7], Bison [5], AYacc [2] and CUP [6].

Although parser generators are valuable teaching and software development tools, understanding and debugging parsers produced by them can be very difficult for students. One reason is that the input specification used by a parser generator is not readily accessible inside the generated parser. In order to debug a generated parser, for example, students must resort to *ad hoc* techniques, such as manually inserting trace statements into the semantic actions of a grammar specification. Clearly, such techniques can be tedious and frustrating for students, and as a result, hinder rather than enable students' ability to learn about compiler construction. Another reason that parser generators can pose problems for students is that various internal data structures, such as state transition tables and symbol stacks, are typically hidden in generated parsers. This is unfortunate since being able to view these data structures can better help students understand their importance and role in the operation of modern compilers. Finally, many students find parsing algorithms, especially LALR parsing algorithms, to be nonintuitive. For example, the order in which productions in a grammar are selected and applied is implicit in the input specification. Students are not given the opportunity to actually see the dynamic properties of a generated parser as it executes.

As a step toward addressing some these shortcomings, we are developing an approach to support visualization of compilers. Our overall objective is to develop techniques and tools that allow students to view all aspects of a compiler as they implement and test their compiler project. In this paper, we describe a tool, called CUPV,[3] for visualizing the operation of LALR gener-

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advant -age and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGCSE 2000 3/00 Austin, TX, USA
© 2000 ACM 1-58113-213-1/00/0003...$5.00

[1] Left-to-right parse, Leftmost-derivation
[2] Look-ahead Left-to-right parse, Rightmost-derivation
[3] Clemson University Parser Visualizer

ated parsers. Specifically, CUPV provides a graphical user interface (GUI) to a parser, which shows the parser stack, along with various other key data structures that are used in the parser. CUPV can optionally depict reductions as well as individual item sets. The CUPV visualization tool has been developed as an extension to CUP [6] – a Java-based parser generator tool.

The remainder of the paper is organized as follows. In Section 2, we describe various features of the CUPV visualization tool by applying it to a simple, though illustrative example. In Section 3, we outline how CUPV parsers can be customized for individual compiler projects. Section 4 provides an evaluation our approach. We conclude with a discussion of related work in Section 5 and a summary in Section 6.

2 The CUPV Tool

The CUPV visualization tool is designed as an extension to CUP (Constructor of Useful Parsers) [6], a Java-based parser generator. CUP translates a context-free grammar specification of a language into a set of parsing tables, which are then used to drive an LALR parsing algorithm. In addition, semantic actions can be associated with grammar productions by embedding Java code in the input specification. This allows semantic information, such as abstract syntax trees, to be calculated during a parse.

For example, Figure 1 shows a fragment of a typical CUP input specification, in this case for a simple calculator. The grammar has three productions, which can recognize legal calculations (e.g., addition, subtraction, etc.) The specification includes declarations for the various terminals and non-terminals defined by the grammar. The types of semantic values associated with terminals and non-terminals are also included. For instance, a Java Integer is associated with the nonterminal **expr** and terminal **NUMBER**. The semantic actions in Figure 1 actually compute the value of an arithmetic expression. Using this specification, CUP can create a parser that recognizes and evaluates expressions.

CUPV extends CUP by producing a parser with a graphical user interface (GUI). To help illustrate CUPV, we will use the calculator example in Figure 1. When a CUPV-generated parser is run, a GUI for the parser will appear. Figure 2 shows the GUI when applied to the input string

$$3 * (5 \% 2) - ((16 / 4) - 2);$$

The primary focus of the parser GUI is the parse stack (left hand portion of Figure 1). Each element in the parse stack is a button, which is labeled with a symbol (terminal or nonterminal) and a parse state. As will be discussed in in Section 2.2, clicking a stack button will

```
/* Terminals (tokens returned by the scanner). */
terminal            SEMI, PLUS, MINUS, TIMES, DIVIDE, MOD;
terminal            UMINUS, LPAREN, RPAREN;
terminal Integer    NUMBER;
/* Non terminals */
non terminal Object     expr_list, expr_part;
non terminal Integer    expr;

/* The grammar */
expr_list ::= expr_list expr_part
              |
              expr_part  ;

expr_part ::= expr:e SEMI
              {: System.out.println ("= " + e); :} ;

expr      ::= expr:e1 PLUS expr:e2
              {: RESULT = new Integer(e1.intValue() + e2.intValue()); :}
              |
              expr:e1 MINUS expr:e2
              {: RESULT = new Integer(e1.intValue() - e2.intValue()); :}
              ... ;
```

Figure 1: Simple Calculator Specification

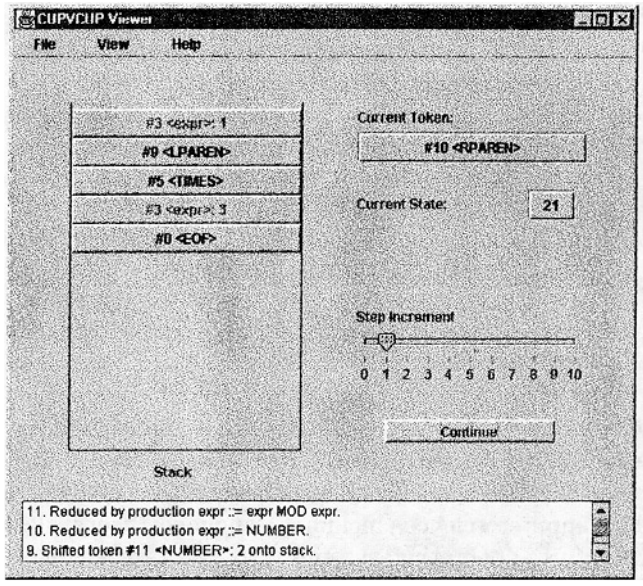


Figure 2: The Viewer Interface

display the semantic value associated with that symbol. The GUI also can display other parser data structures including the current state of the parser, the current token in the input string, a log of actions performed and details about reductions. In Figure 2, for instance, the parser is currently in state 21, and the last action performed was the reduction

$$\langle expr \rangle ::= \langle expr \rangle \text{ MOD } \langle expr \rangle$$

2.1 Controlling Parser Execution

Parser execution is controlled with a slider bar (lower right hand portion of Figure 2) that allows a user to specify the number of steps the parser should execute before waiting for interaction by a user. The Start-/Continue/Stop button (shown as a Continue button in Figure 2) is used to initiate a parse, and, subsequently, to interrupt and continue the execution of the parser.

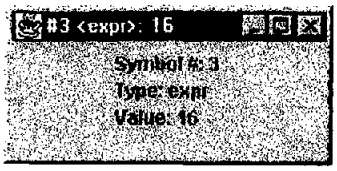


Figure 3: Display of a Semantic Value

For example, if the step increment is set to 4 when the user clicks the Start button, the parser will execute four steps and then stop until the user clicks the Continue button. If the slider bar is set to a value of 0, then the parse will continue to completion. At any time when the parser is executing, the Stop button may be clicked to interrupt execution. The parser will stop when it completes the currently executing step and wait for the Continue button to be clicked.

For example, in Figure 2, the parser is at a point where it has processed the input string up to the first ')' character (token RPAREN). The <NUMBER> token with value 3 has been read, shifted, and reduced to a <expr> nonterminal with the same value. The <TIMES> and <LPAREN> tokens have been read and shifted onto the stack. The substring "5 % 2" has been read, shifted, and reduced to the <expr> nonterminal with value 1.

2.2 Display of Semantic Values

As noted above, symbols are displayed by a CUPV parser in the form of buttons on the stack. A symbol that has an associated value is displayed with a label colored in pink, signifying that the button may be clicked to display its semantic value. Figure 3 shows the display for the <expr> symbol, whose value is 16. This particular display can be customized to view arbitrary semantic values. For example, in practical situations, an abstract syntax subtree is often used as a semantic value during parsing. This is feature is discussed in greater detail in Section 4.

2.3 Displaying Reductions

The View–Reductions option can be used to warn users when a reduction is about to occur. Sequences of symbols that appear on the top of the stack will be highlighted (by changing their label color to green) when the sequence matches a right-hand side of a production. A status message "Getting ready to reduce" will also appear. When the user clicks the Continue button, a new window will appear containing a graphical display of the reduction. Figure 4 shows the reduction

<expr> DIVIDE <expr> ::= <expr>.

This display shows the symbols that form the right-hand side of the production as they appear on the parse stack,

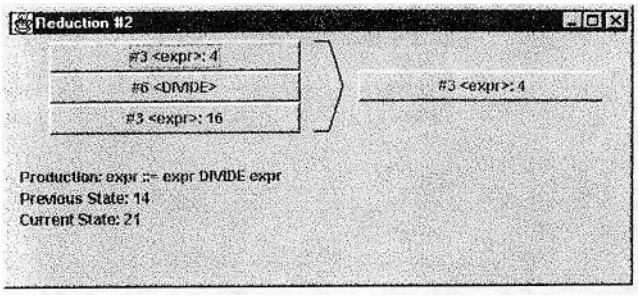


Figure 4: A Reduction Display

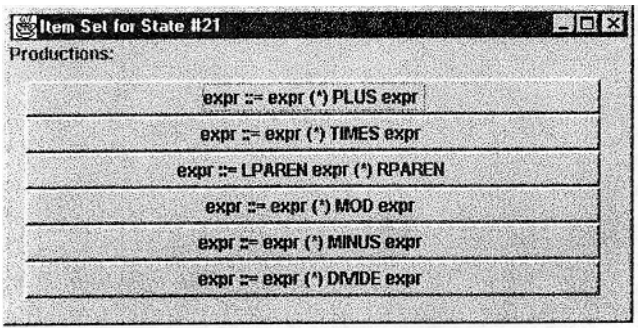


Figure 5: Display of the Item Set for a State

the left-hand side terminal that will replace them on the stack, the production matching the reduction, and the previous and current states of the parser. As with all symbol displays, any of the symbols in the reduction display may be clicked to view their semantic values.

2.4 Viewing Parser States

The current state of the parser is displayed by a CUPV parser as a labeled button. For example, in Figure 2, the current state of the parser is state number 21. This button may be clicked to display the item set for the current state. Figure 5 shows the item set for state number 21 in the calculator example. Each production in the item set, along with the current position in the production, is depicted. The "(*)" in each production in the item set shown in Figure 5 is used to indicate the current position of the parse. Any of the productions in the item set may be clicked to present a display of the lookahead for that production. The lookahead contains the set of symbols which may follow the associated production in an acceptable string. In Figure 6, the lookahead is shown for the production:

expr ::= expr (*) PLUS expr

If at the end of input, the only symbol on the stack is the start symbol for the grammar, then the string is accepted. This means that the entire input string must be reduced to the start symbol. Any condition other than this causes the parser to report a syntax error. If the parser reports a syntax error during a parse, a

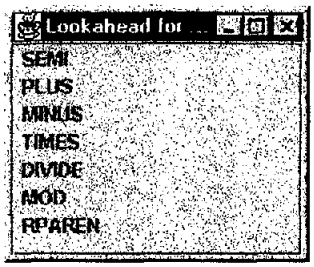
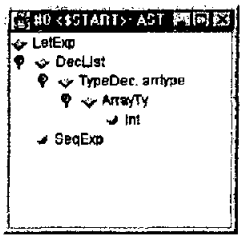


Figure 6: Display the Lookahead for a Production




```
array.tig :

let
    type arrtype = array of int
in
end
```

Figure 7: Display of an Abstract Syntax Tree and the Tiger Input

warning window will appear on the screen.

2.5 Summary

A CUPV parser allows users to visualize the execution of a generated parser. Data structures that are normally hidden from users are made explicit and may be manipulated. This makes it easier for students to understand the operation of a parser. It also provides a useful debugging facility since users can step through the execution of parser. Students can examine the semantics values, item sets, and lookaheads to determine where in a specification an error originated. This allows the student to concentrate on specific productions or actions in the specification, reducing the amount of work that must be done to correct the problem.

3 Customizing Semantic Displays

As shown in the previous section, a CUPV parser can display semantic values associated with symbols by clicking a button on the parse stack. In the calculator example, the semantic values are relatively simple – just integer values. In a more realistic parser, the semantic values can be quite complex. As noted, many compiler projects require the creation of abstract syntax trees as an internal representation. The CUPV tool is quite flexible since it allows users to create displays for any semantic values.

The CUPV tool is actually based on a general framework that can be used to define and customize visualization capabilities for generated parsers [9]. Although the details of the framework are beyond the scope of this paper, customized displays for semantic values can be created by simply extending a CUPV-defined class. This class defines the behavior for the parse stack buttons. Specifically, the class defines a method, called getDisplay that creates and returns a Java Swing frame (i.e., a JFrame object). The CUPV GUI invokes this method when a parse stack element is clicked. Thus, a user can have any semantic value displayed as along as it can be depicted in a Java Swing frame component.

4 Evaluation

In this section, we provide some preliminary evidence that demonstrates the utility of our approach. The compiler project described in [1] is intended for compiler courses. It relies on the use of the JLex [3] lexical analyzer generator and the CUP parser generator in the creation of a compiler for an Algol-like programming language called Tiger. The Tiger grammar has approximately 50 tokens and almost 100 productions. Tiger serves as a useful basis for evaluation since it is a non-trivial language with which useful programs can be written, and it makes use of abstract syntax trees to represent the structure of a program. To test our approach, we created a CUPV parser for this project.

Creating a CUPV parser for Tiger involved defining a class that displays abstract syntax trees. As discussed in Section 3, this basically requires extending the class that defines parse stack buttons. For Tiger, any parse stack button with an associated abstract syntax tree can be clicked to display the content of the tree. Figure 7 shows an example display, along with the corresponding Tiger input program.

Semantic values in the Tiger language are defined by an Absyn package, which contains classes describing the abstract syntax trees for various symbol types. Each of the classes within the Absyn package extends one of three classes: Absyn, DecList, or ExpList. By adding a getDisplay method to each of these base classes, we are able to create abstract syntax tree displays. Specifically, the getDisplay method returns a JFrame containing the tree rooted at the object.

Symbols with values that are abstract syntax trees are displayed in the CUPV parser with the label "AST." When the developer clicks on a parser stack button, a window containing the abstract syntax tree is depicted. Handles to the left of the nodes allow users to open or close branches of the tree. In Figure 7, the entire abstract syntax tree is displayed.

This exercise illustrates that with minimal effort, a user may expand the capabilities of CUPV. Any language attribute for which a graphical display can be created

can be integrated into CUPV, allowing the developer to visualize more effectively the execution of a parser generated for that language.

5 Related Work

Several tools have been developed that provide graphical depictions of an executing parser. Tools such as Visual Yacc [10] and Gyacc [8] were created for the purpose of visualizing parsers generated by Yacc. Other tools, such as LRParse and LLParse [4], were developed for the purpose of helping students understand parsing techniques.

The Visual Yacc tool [10] was developed as a classroom tool to assist in the teaching of grammar writing. It does not focus on displaying the actions of a parser or the content of parse tables. The developers of this system did discover that the tool "could be used for debugging as well as teaching," [10] but it was not developed specifically for this purpose. A Yacc parser specification must be modified by adding Motif calls before the visualization tool may be used. This tool does not display parser actions such as reductions; instead, the result of the reduction is seen in its effect on the stack and parse tree.

The GYacc tool [8] was created as an aid to the development of valid grammars, but it can only be used with Yacc-generated parsers. This tool is separate from the generated parser; it displays a simulation of the parser execution based on a Yacc specification and an input string.

LRParse [4] is a visualization tool developed for teaching purposes. It graphically depicts the construction of LR(1) parse tables from an input grammar and the use of the tables to parse input strings. LLParse is a similar tool for LL(1) parsers. These tools, however, can not be used in conjunction with parser generators.

6 Summary

CUPV allows a user of the CUP parser generator to visualize the execution of a generated parser. This tool was designed to make comprehending and debugging of parser specification easier for students, by enabling the visualization of several critical aspects of parser execution. Without such a tool, students are forced to rely on more tedious and error-prone methods for comprehending and debugging generated parsers.

Anticipated future work includes the incorporation of other visual aspects of compiler construction. The visualization could be expanded to include other phases of compiler development, such as the creation of symbol tables, the generation of intermediate code, and the translation of intermediate code to actual code. Other future directions include exploring ways of generalization our framework so that visualization capabilities can be incorporated into any parser generated tool.

References

[1] Appel, A. W. *Modern Compiler Implementation in Java.* Cambridge University Press, Cambridge, UK, 1998.

[2] Arcadia Environment Research Project, Department of Information and Computer Science, University of California. *Ayacc User's Manual, Arcadia Document UCI-94-01*, version 1.1 ed. Irvine, CA, Mar. 1994.

[3] Berk, E., and Dirichs, M. JLex: A lexical analyzer generator for Java, 1998.

[4] Blythe, S. A., James, M. C., and Rodger, S. H. Llparse and lrparse: Visual and interactive tools for parsing. In *Proceedings of the 25th Technical Symposium on Computer Science Education* (New York, NY, USA, Mar. 1994), D. Joyce, Ed., vol. 26_1 of *SIGCSE Bulletin*, ACM Press, pp. 208–212.

[5] Donnelly, C., and Stallman, R. M. BISON — the YACC-compatible parser generator. Tech. rep., Free Software Foundation, 675 Mass Ave, Cambridge, MA 02139, USA, Tel: (617) 876-3296, 1988.

[6] Hudson, S. E., Flannery, F., Ananian, C. S., Wang, D., and Appel, A. W. CUP parser generator for Java, Mar. 1998.

[7] Johnson, S. C. YACC — Yet another compiler - compiler. Computing Science Technical Report No. 32, Bell Laboratories, Murray Hill, N.J., 1975.

[8] Lovato, M. E., and Kleyn, M. F. Parser visualizations for developing grammars with yacc. In *Proceedings of the 26th Technical Symposium on Computer Science Education* (New York, NY, USA, Mar. 1995), vol. 27 of *SIGCSE Bulletin*, ACM Press, pp. 345–349.

[9] Shoup, D. Visualizing lalr generated parsers. Masters project report, Deparment of Computer Science, Clemson University, Clemson, SC, Aug. 1999.

[10] White, E., Deddens, L., and Ruby, J. Software visualization of LR parsing and synthesized attribute evaluation. Technical Report TR98-01, George Mason University, Computer Science, Apr. 6, 1998.