LLparse and LRparse: Visual and Interactive Tools for Parsing

Stephen A. Blythe, Michael C. James, and Susan H. Rodger[1]

Department of Computer Science
Rensselaer Polytechnic Institute
Troy, NY 12180-3590
email: rodger@cs.rpi.edu

Abstract

This paper describes instructional tools, LLparse and LRparse, for visualizing and interacting with small examples of LL and LR parsing. These tools can be used to understand the process of constructing LL(1) and LR(1) parse tables through a series of steps in which users receive feedback on the correctness of each step before moving on to the next step. For example, in LRparse, the user initially enters an LR(1) grammar, calculates FIRST and FOLLOW sets, graphically constructs a deterministic finite automaton of item sets, and finally constructs the LR(1) parsing table. Upon completion of the constructed table, the user can observe a visualization of the parsing of input strings. These tools can be used to provide problem solving feedback in courses on automata theory or compiler design.

1 Introduction

Formal languages and automata theory provide the background necessary for understanding the design of programming languages and the construction of compilers, yet this important area is usually taught in a noninteractive learning environment using pencil and paper to solve problems. Most automata theory textbooks [2, 8, 9] represent the concepts of grammars and automata formally as n-tuples of sets. Solving problems using this notation results in tedious descriptions that are difficult to both debug and trace. In addition, most students view this theoretical class as uninteresting compared to their other computer science classes for two reasons: they receive no immediate feedback when solving problems, and they do not see the concepts applied to real problems until later in the curriculum.

In courses with programming, students receive immediate feedback as they compile, run, and debug programs, allowing them to gauge whether a program is correct or not in real time. To make automata theory more interesting, a number of tools [3, 6, 7, 10, 11] have been developed that provide visual feedback on experimenting with automata. For example, FLAP [10] is a tool for drawing a graphical representation of an automaton and then simulating the running of the automaton. Using FLAP, students receive immediate feedback on the format correctness after constructing an automaton and on the logic correctness while simulating the automaton. FLAP and other existing tools provide a mechanism for experimenting with automata; however, the range of these tools covers only a few of the topics discussed in an automata theory course.

In order to fully understand any theoretical area such as automata theory, one has to see how the concepts are applied and use them in real problems. Automata theory forms the underlying basis of the parsing methods used in compilers. Thus, most course textbooks contain LL and/or LR parsing as the main application. These are real methods used by compiler designers. However, constructing LL and LR parse tables by hand is tedious and prone to errors. Once a table is constructed, the table can be used to parse strings to determine if they are in the language of the original grammar. The parsing process is useful in observing how the table is used, but is difficult to trace by hand since the parsing stack is constantly changing.

In this paper, we address both the lack of immediate feedback and the need for experimenting with applications by developing tools that allow students to experiment with the application of finite automata and pushdown automata in parsing. We have developed instructional tools, *LLparse* and *LRparse*, that guide a user through the steps in constructing the LL(1) and LR(1) parse tables for small examples, and then show visualizations of parsing user specified input strings. Further-

[1]Supported in part by a Fellowship from the Lilly Foundation and a Rensselaer CIUE Development Grant for Educational Innovation.

more, our tool allows the user to construct parse tables for grammars that are not LL or LR in order to determine where the process fails. Our tools can be used to enhance undergraduate classes in automata theory or compiler design.

In Section 2 we briefly describe LL and LR parsing and the process of constructing LL and LR parse tables. We describe our tools in Section 3, and in Section 4 we give an example using LRparse. In Section 5 we describe the usage of the tools in education, and in Section 6 we give concluding remarks.

2 Overview of LL and LR Parsing

Parsing is the process of determining whether a string of symbols can be generated from a set of production rules called a grammar. Although algorithms exist to perform this task for any context-free grammar and any given string, they are generally computationally expensive. However, efficient parsers can be constructed to parse LL(1) and LR(1) grammars. In this section we briefly describe LL and LR parsing and the construction of LL(1) and LR(1) parse tables. See [1] for more details.

LL parsing is a top-down method of parsing that begins with the start symbol of a grammar, uses lookaheads in the input string to determine which grammar rules to apply, and derives a leftmost derivation of the input string. In LL(1) parsing, only one lookahead is needed. To simplify the implementation of parsing, a table can be constructed containing the information of which rule to use for a particular variable and lookahead.

Three steps are used in constructing an LL(1) parse table. The first step is to find the FIRST sets for various portions of the right hand sides of rules. The FIRST set of a given rule contains all the symbols that are the first symbol of a string that the rule could derive. The second step is to find the FOLLOW sets for all variables in the grammar. The FOLLOW set of a given variable is comprised of all the symbols of the grammar that can immediately follow that variable in the derivation of a string. The third step consists of analyzing each rule in the grammar with the information in the FIRST and FOLLOW sets to fill in the appropriate rules in the parse table.

LR parsing is a bottom-up method of parsing that starts with the input string, replaces right-hand sides of rules by left-hand sides until the start symbol of the grammar has been derived, thus generating a rightmost derivation in reverse order. Since right hand sides of rules usually contain multiple symbols, rules are represented internally as items. An item has a marker in one position of the right hand side of the rule indicating that symbols to the left of the marker have already been recognized, and symbols to the right of the marker must still be recognized in order to match this rule. An LR(1) parser groups the items into states such that an item is in a state if and only if the item can match the current input to the parser. These states, referred to as item sets, and the transitions between them can be represented by a deterministic finite automaton (DFA). All of this information can be stored in a table to facilitate parsing. Rows represent states of the parser, and each row's corresponding columns indicate a course of action to be taken based on the grammar symbol that identifies each column.

There are four steps in computing an LR(1) parse table. As with LL(1) parsing, the first two steps are computing FIRST and FOLLOW sets. The third step is computing the DFA and calculating the item sets for each of its states. In the final step, the parse table is constructed from the states and transitions in the DFA.

3 The tools: LLparse and LRparse

LLparse and LRparse [4, 5] are two interactive and visual instructional tools for constructing LL(1) and LR(1) parse tables from appropriate grammars, and for using these constructed tables to parse strings. Naturally, there are size limitations due to what can visually be displayed. These tools allow reasonable size examples for experimenting with and understanding these methods. LLparse and LRparse are written in C++ and use the X Window System[2]. In this section we describe the graphical user interface and the interactive error checking.

3.1 The User Interface

For the most part, the interface is common between the two tools. Both tools consist of a series of windows representing the steps in building a parse table. At a given window, the user cannot proceed to the next window until the current step is correctly completed.

In the initial window of LLparse, the user must enter an LL(1) grammar that contains at most fifteen rules. After such a grammar is entered successfully, a second window pops up with a table of blank FIRST sets of appropriate strings that need to be filled in. Upon successful entry of these sets, a similar window pops up for the FOLLOW sets of variables, which the user must continue to enter until they are correct.

At this point in LLparse, a window appears containing an empty LL(1) parse table with the correct number of labeled columns and rows. Upon successful entry of the parse table, a parsing window appears. The parsing window allows the user to enter an input string and

[2]The X Window System is a trademark of the Massachusetts Institute of Technology.

start an animation that visualizes the parsing of this string step by step, showing the current stack contents and explaining which entries in the table are being used.

In LRparse, the first three windows encountered are the same as LLparse, except that an LR(1) grammar with at most fifteen rules must be entered in the first window. The fourth window popped up in LRparse requires the entry of a DFA representing the states in the parsing process. This window is a modified version of the DFA window in FLAP [10]. By clicking mouse buttons, the user can graphically draw states and labeled arcs representing a DFA. The number of states in the DFA is limited to at most twenty-five. The item sets can be generated for each state by clicking on the state and entering the set of items in a small window corresponding to the state. Upon successful creation of the correct DFA and item sets, a window representing the LR(1) parsing table appears with the correct number of labeled rows and columns. Once the table is successfully filled in, the final parsing window appears. This window allows example strings to be visually parsed using the constructed LR(1) parsing table.

3.2 Interactive Error Checking

The interactive error checking is performed when a user clicks on a **Done** button in the current window during a run of the program. At this point in time, the information given by the user is parsed and checked against the correct responses that have been internally generated.

In the initial window of either tool, the user must type in an LL(1) or LR(1) grammar. If the grammar is of the correct form, internal programs automatically calculate solutions for the remaining steps. If the grammar is not of this form, the user is notified and has two options. The user can either change the grammar or continue with the current grammar up to the point of constructing the parse table in order to determine why the grammar is not LL(1) or LR(1). For such grammars, the user can observe that at least one slot in the parse table will have multiple entries.

In the FIRST sets, FOLLOW sets and Parse Table windows, each user entry is checked against the automatically generated solutions. User entries containing an incorrect answer are highlighted for the user to correct. The user must continue to change incorrect answers until the correct answer is entered in order to continue. As an alternative, the user can select the **Show** option to fill in the correct answers and then move on. The error checking in these windows is a simple matter as all the input is in textual form.

The method for checking the correctness of a DFA is considerably more complex because it is not entirely based on textual responses. Here, the user's DFA must be compared to an internal representation of the DFA. The internal DFA's states are not likely to be labeled identically as those of the user's DFA. Thus a graph traversal algorithm is used to compare the two versions of the DFA. As each node of the DFA is reached in the traversal, its associated item set is compared to the corresponding internally generated item set. Each state is also checked to see that the number of transitions leaving it is correct and that each of those transitions is on a valid symbol. If any of these or various other conditions fail, the state is highlighted and an explanatory error message is displayed.

4 An Example Assignment

Here is a homework problem given in spring 1992.

1. Consider the following grammar:

$$S \rightarrow AcB \qquad A \rightarrow Aa \mid \lambda \qquad B \rightarrow bB \mid \lambda$$

   (a) Compute FIRST and FOLLOW for all the variables in the grammar.

   (b) Construct the DFA that models how an LR parser works.

   (c) Construct the LR(1) parse table.

   (d) Show a trace of the string 'aacbbb'. Show both symbols and state numbers on the stack.

This problem could be solved by using LRparse in the following manner. Upon starting the tool, the grammar window in Figure 1 initially would appear empty, and the user would type in the grammar. After selecting **Done**, the FIRST sets window appears. In Figure 2 the user has been informed that the FIRST set for S is incorrect. It should be "ca". Upon correcting the mistake and selecting **Done**, a similar FOLLOW set window appears for the user to fill in. Next, a blank Build window appears that is similar to the DFA part of FLAP. The user can construct the corresponding DFA that models the symbols on top of the stack. In Figure 3 the user has been informed that there is an incorrect transition from state q6, the highlighted state. The transition from state q6 to state q7 should be labeled by the symbol "B". Item sets must be entered for each state. Figure 4 shows the item sets for state q0. This window pops up after selecting the state, and can stay up as long as the user desires. After completing the correct DFA, selecting done brings up the parse table window. Figure 5 shows the parse table for the grammar in Figure 1. There is a mistake in row 3 that the user needs to correct before continuing. The entry in row 3, column b should be "s6" (shift and move to state 6). The rows in the parse window correspond to the states in the DFA. Row 3 corresponds to state q3.

By selecting row 3, the item sets for state q3 can be displayed and removed when desired.

When the parse table is correct, the final window displayed is the parsing window. Figure 6 shows part of the trace of the string *aacbbb*. Informative messages are displayed at the bottom of the window, telling what type of operation (reduce, shift, accept, or error) has just been performed.

There are several additional features available to the user: printing (or writing to file) the results of all steps, reading in a grammar, and receiving online help.

5 Usage of Tools in Education

The tools LLparse and LRparse can be used to enrich undergraduate classes on automata theory or compiler design. Both parsing methods can be modeled using a pushdown automata. Furthermore, the LR parsing process contains the construction of a DFA that models the contents of the top of the parsing stack. In a compiler design course, these tools provide examples of both top-down and bottom-up parsers. More importantly, students can experiment with the tools in order to understand the parsing methods. In both classes, the tools can be used to study LL(1) and LR(1) grammars and students can discover why certain grammars are not LL(1) or LR(1).

These tools have been used at Rensselaer in the sophomore course Fundamental Structures of Computer Science, in which half of this course is spent on automata theory. During class, the professor ran LLparse and LRparse on an X terminal, projecting the image on an overhead projector. The students were shown how to use these tools as the class provided input and the professor typed in their suggested solutions, going through all the steps in constructing the tables, and then watching the animations of parsing strings, using the constructed tables. Grammars studied during class were saved in files by the professor and made accessible to students, so they could recreate the examples done in class. Additional problems were assigned and students were encouraged to use these tools for working the problems.

Using the tools, assigned problems can be worked on until they are understood, as one cannot move to the next step until the current step has the correct solution. When students can't solve a particular problem, they will know right away to come and ask the instructor for help. Previously, without these tools, students turned in written assignments with incorrect answers, and didn't find out what they missed until the assignments were graded and handed back, usually a week or more later. Using the tools, students can work on extra problems and receive immediate feedback, which will especially benefit those students with weaker mathematical skills.

6 Conclusions and Future Work

We have developed two tools, LLparse and LRparse, to aid students in constructing and testing LL(1) and LR(1) parse tables. These tools can be used with courses on automata theory or compiler design to provide an interactive class environment and a chance for students to receive feedback on solving problems. We are currently working on developing additional tools with the long range goal of designing a formal languages and automata theory course in which the majority of the concepts can be experimented with using tools.

References

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

[2] A. Aho and J. Ullman, *Foundations of Computer Science*, Computer Science Press, 1992.

[3] J. Barwise and J. Etchemendy, Turing's World, Kinko's Academic Courseware Exchange, Santa Barbara, CA, 1986.

[4] S. Blythe, LLparse and LRparse: Interactive LL(1) and LR(1) Parsing Tutorials, Master's Project, Rensselaer Polytechnic Institute, May 1993.

[5] M. James, A Software Tool to Aid in Understanding LL Parsing, Master's Project, Rensselaer Polytechnic Institute, September 1992.

[6] D. Hannay, Hypercard Automata Simulation: Finite State, Pushdown and Turing Machines, *SIGCSE Bulletin*, 24, 2, p. 55-58, June 1992.

[7] M. C. Lee, An Abstract Machine Simulator, *Third International Conference on Computer Assisted Learning*, p. 129-141, 1990.

[8] H. Lewis, and C. Papadimitriou, *Elements of the Theory of Computation*, Prentice-Hall, 1981.

[9] P. Linz, *An Introduction to Formal Languages and Automata*, D. C. Heath and Company, 1990.

[10] M. LoSacco, and S. H. Rodger, FLAP: A Tool for Drawing and Simulating Automata, *ED-MEDIA 93,* p. 310-317, June 1993.

[11] K. Sutner, Implementing Finite State Machines, *DIMACS Workshop on Computational Support for Discrete Mathematics*, 1992.

Figure 1: Entered LR(1) grammar

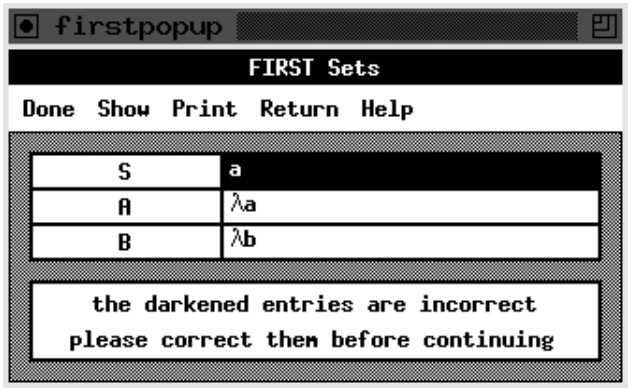

Figure 2: FIRST sets with error

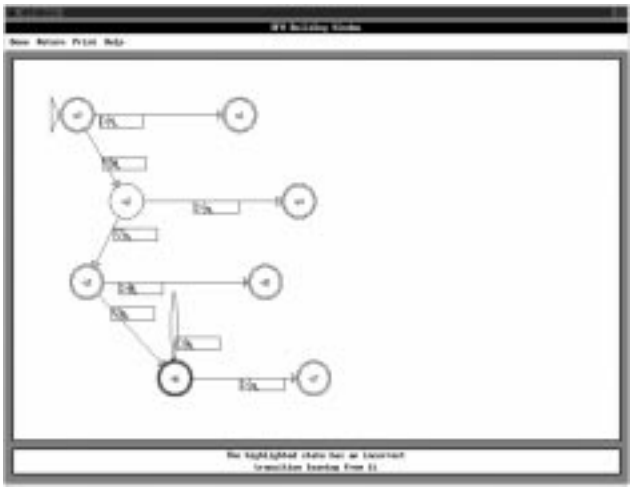

Figure 3: DFA Building window with error

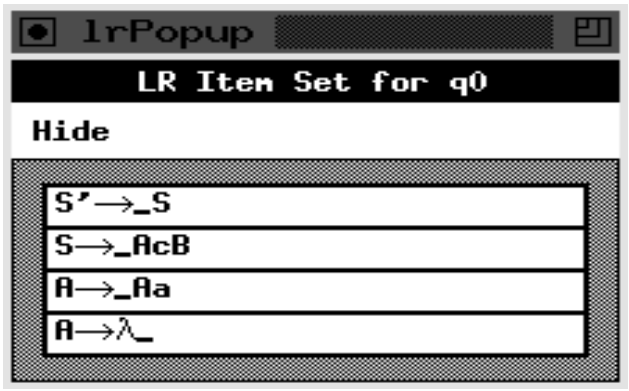

Figure 4: LR item set for state q0

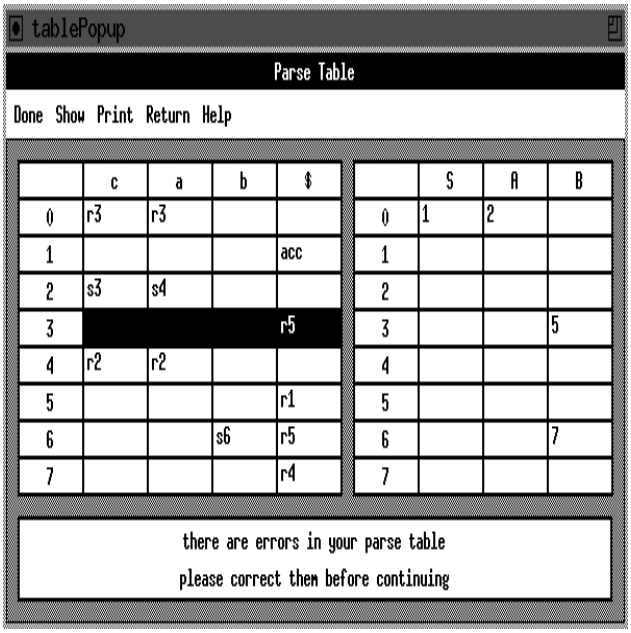

Figure 5: LR(1) parse table with error

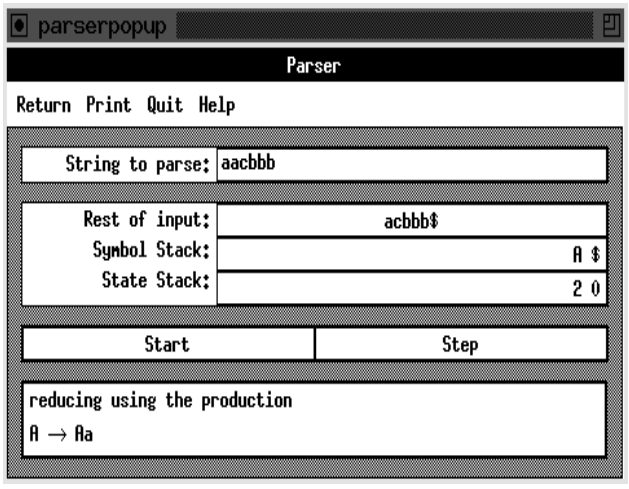

Figure 6: Parsing string *aacbbb*