A Collection of Tools for Making Automata Theory and Formal Languages Come Alive

Anna O. Bilska, Kenneth H. Leider, Magdalena Procopiuc, Octavian Procopiuc,
Susan H. Rodger, Jason R. Salemme and Edwin Tsang
Duke University, Durham, NC
rodger@cs.duke.edu

Abstract

We present a collection of new and enhanced tools for experimenting with concepts in formal languages and automata theory. New tools, written in Java, include JFLAP for creating and simulating finite automata, pushdown automata and Turing machines; Pâté for parsing restricted and unrestricted grammars and transforming context-free grammars to Chomsky Normal Form; and PumpLemma for proving specific languages are not regular. Enhancements to previous tools LLparse and LRparse, instructional tools for parsing LL(1) and LR(1) grammars, include parsing LL(2) grammars, displaying parse trees, and parsing any context-free grammar with conflict resolution.

1 Introduction

The majority of computer science courses have a hands-on approach, since they have a natural programming component. In the introductory courses, students learn a programming language and write programs to experiment with the language. In many later courses, students work on programming projects related to the topic, such as writing components of an operating system, or using libraries to solve scientific computing problems. Theoretical computer science courses, algorithms and formal languages, are an exception, traditionally taught with no programming assignments, but rather students write homework assignments using pencil and paper and do not receive feedback until the assignments are graded.

With the development of algorithm animation tools such as AACE [5], Xtango [11], and Zeus [4], algorithms courses are demonstrating animations of algorithms and data structures during lectures [8], which the students can recreate outside of lectures. For example, inserting elements into a red-black tree and watching the changes (rotations) to the data structure at one's own pace is helpful in learning the algorithm. In [1], a study showed that students learned algorithms better if they had hands-on experience. In this study, students wrote the code for the animations in order to learn the algorithms.

The formal language and automata theory course has lagged behind in integrating hands-on materials into lectures and assignments. In this paper, we describe a collection of tools we have developed for experimenting with many concepts in this course. JFLAP is for experimenting with automata, pushdown automata and Turing machines; LLparse and LRparse are for experimenting with top-down and bottom-up parsing; Pâté is both a brute force parser for restricted and unrestricted grammars and a grammar transformer from a context-free grammar to CNF; and PumpLemma is a tool for experimenting with the pumping lemma. LLparse and LRparse are written in C++ and X Windows and the remaining tools are written in Java.

In Section 2 we describe recent enhancements we have made to the tools LLparse and LRparse, and in Section 3 we describe new tools we have developed. In Section 4 we describe the use of these tools in teaching. In Section 5 we describe additional automata tools developed by others, and in Section 6 conclude and discuss future work.

2 Enhancements to LLparse and LRparse

LLparse and LRparse [3] are instructional tools for constructing LL(1) and SLR(1) parse tables through a series of steps, and then animating the stack in parsing input strings. In LLparse, the user enters a grammar and can proceed if the grammar is an LL(1) grammar. The user then enters FIRST sets of variables and then enters FOLLOW sets of variables. If a set is incorrect, the set is highlighted and the user must either change the set or select *Show* to see the answer. Finally, the user fills in an LL(1) parse table with the appropriate entries. Once correct, a parsing window appears. The user can enter any input string and then step through its parsing.

Enhancements to LLparse include handling LL(2) gram-

*This material is based upon work supported by the National Science Foundation's Division of Undergraduate Education through grants DUE-9596002 and DUE-9555084.

mars and presenting the parse tree when parsing a string. When a user enters a grammar, he/she can select to construct either an LL(1) or LL(2) parse table In constructing the LL(2) parse table, the user enters FIRST sets, FIRST2 sets (for 2 lookaheads), FOLLOW sets and FOLLOW2 sets. The LL(2) parse table shown will automatically compress columns where only 1 lookahead instead of 2 are needed, reducing the size of the table. A partial LL(2) table from LLparse is shown in Figure 1. In the parsing window, the user enters an input string and steps through the parsing process. In addition to a stack trace, when a variable is replaced by its right hand side, the right hand side is added to the picture of the parse tree.

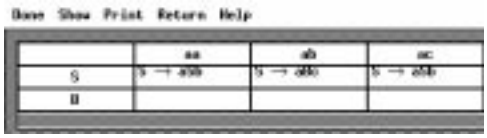

Figure 1: Partial LL(2) Table

In LRparse, the user enters an LR(1) grammar, followed by FIRST and FOLLOW sets. Then in a drawing window, the user draws a transition diagram of a FA that models the stack. For each state in the FA, the user enters the item set (marked rules). As in all windows, the user must enter the correct information before proceeding. There is a *Show* button if the user wants to see the answer. Next, the user fills in the LR(1) parse table. Finally, a parsing window appears and the user can step through the parsing of an input string.

Enhancements to LRparse include parsing any context-free grammar with conflict resolution and showing the parse tree when parsing an input string. If a grammar is not LR(1), there will be multiple items in at least one entry of the parse table. The first item in the entry will be selected in parsing. Thus the user can experiment with choosing different items as the first item. In the parsing window, the user enters an input string and steps through the parsing process. In addition to a stack trace, when a rule is reduced, the reduction is shown by joining the right hand side of the rule in the parse tree.

3 New Java Tools for Formal Languages

We have developed a set of tools written in Java for experimenting with concepts in formal languages and automata theory. JFLAP (Java Formal Languages and Automata Package) allows one to construct automata, pushdown automata and Turing machines, and run traces on input strings. Pâté (Parsing And Transforming Engine) contains two parts for experimenting with grammars, a brute force parser that shows the derivation and parse tree for an input string, and a transformer for converting a context-free grammar to CNF through several steps. PumpLemma allows one to experiment with the regular pumping lemma. We describe these tools in the following sections.

3.1 JFLAP, New Version of FLAP

JFLAP[10] is a Java implementation of FLAP[7]. In JFLAP, one can graphically construct a transition diagram for nondeterministic versions of finite automata, pushdown automata, and 1-tape and 2-tape Turing machines. Figure 2 shows a 1-tape Turing machine in JFLAP that adds unary numbers. Once constructed, an input string is entered and either fast or step run is selected. In fast run, a message quickly responds indicating the acceptance of the string. In step run mode, all current configurations (there is more than one if the machine is nondeterministic) are shown at each step and the user must control the trace by freezing or killing configurations if there are more than 15.

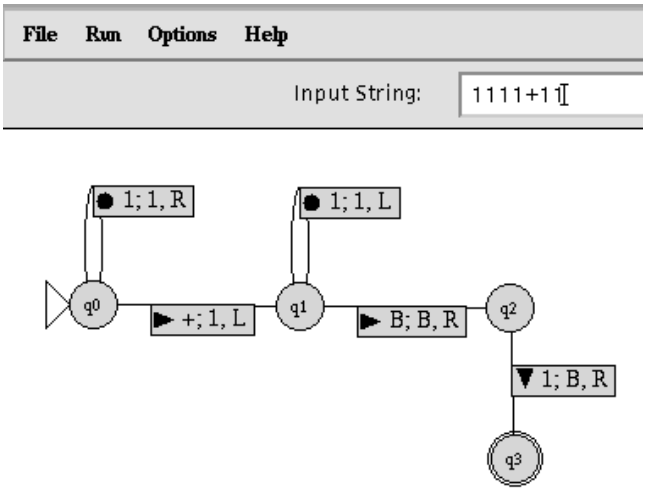

Figure 2: Turing Machine Example

3.2 Pâté - Brute Force Parser

The brute force parser part of Pâté is an exhaustive search parser for restricted (regular and context-free) and unrestricted grammars. Given a grammar and an input string, the parser builds a derivation tree (not displayed) of all possible derivations in a breadth-first manner. Each node in the tree contains a sentential form and the production number used to get from its parent node to itself. The start symbol S is the root of the tree. A derivation of the input string is found when the input string appears as a sentential form (a node). To speed up the creation of the derivation tree, the user can choose to allow only one such node for each sentential form in the tree. Thus, each new sentential form generated is first looked up to see if the sentential form already exists, and if so it is not added. For restricted grammars, additional pruning of nodes is accomplished by eliminating nodes whose prefix, suffix or substring of terminals do not match in the input string. Once a derivation is found, a message indicates the acceptance of the string and the size of the derivation tree. The user can choose to display the actual derivation in textual format or in the form of a parse tree (for restricted

grammars only). If all nodes in the derivation tree are exhausted and the string is not found, a message indicates that the string is not in the language of the grammar.

Pâté is an instructional tool to experiment with small grammars and small input strings, and works well for most assignments given to beginning students. Obviously, for some strings and grammars, the parsing may take a long time. For large sizes of the derivation tree, messages appear indicating to the user the size of the tree and asking if they want to continue. The user can also pause the parsing at any time for the same information.

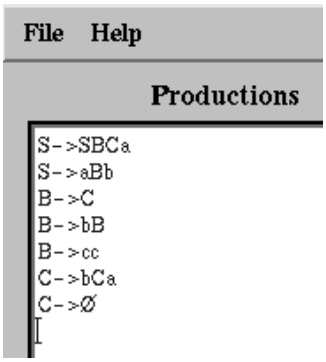

Figure 3: Grammar in Pâté

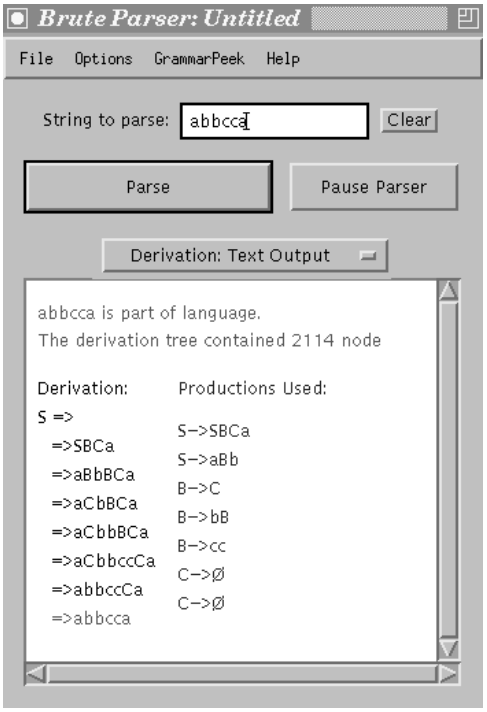

Figure 4: Derivation of $abbcca$

We give an example of Pâté parsing a string in a grammar. Figure 3 shows a portion of the initial Pâté window. Either a restricted or unrestricted grammar is entered. In this case, a context-free grammar is entered ($\emptyset$ represents the empty string). There are two buttons at the bottom of the window (not shown) for selecting either *Parser* or *Transform Grammar*, and a message window to indicate if the grammar is in the correct format. The *Parser* button is selected and the parsing window in Figure 4 appears (except the string and window are blank). The user enters an input string at the top and selects *Parse*. While the parser is busy, animations of fractals appear on the *Parse* button. When the parser completes, the derivation and information are displayed in the bottom half of the window. Figure 4 shows the derivation for the string $abbcca$. The derivation tree was quite large, containing 2114 nodes or sentential forms. Alternatively for restricted grammars only, one can select graphical output and the parse tree is shown. The parse tree for $abbcca$ is shown in Figure 5.

3.3 Pâté - Grammar Transformer

The grammar transformer part of Pâté is an instructional tool for converting a context-free grammar to Chomsky Normal Form (CNF) through a series of steps. At each step the user is requested to enter information and cannot proceed until the information is correct. There is a *Show* button to show the answer in case the user is stuck.

To use the transformation tool, the user enters a context-free grammar in the original Pâté window and selects *Transform Grammar*. The first step is to remove lambda productions. A window appears and the user first enters the variables that derive lambda productions. Once verified these are correct, the user enters the new grammar without lambda productions. No windows are removed, so the user can look at the grammar window to see the original grammar. The new grammar without lambda productions for the grammar in Figure 3 is shown in Figure 6. The user would click on the button *Next Transformation* at the bottom of the window (not shown in the figure) to move to the next transformation.

The next two steps remove unit productions and useless productions. Both of these steps include a window to draw a graph modeling how unit productions are connected (in the first case) and which variables can be replaced by other variables (in the second case). The final step is a window for constructing the CNF grammar. In this window, the user is informed of the format for additional variables, B(x) for a new variable deriving the single terminal x and D(#) (where # is an integer) for other new variables. Informative error messages tell the user which rules are typed incorrectly or have not yet been replaced. For some grammars, all steps in the conversion process may not be needed. A user is informed if a step does not apply and can skip the step (if a grammar does not have unit productions, none need to be removed).

3.4 PumpLemma

PumpLemma is a tool for experimenting with the regular pumping lemma. The user enters a non-regular language L

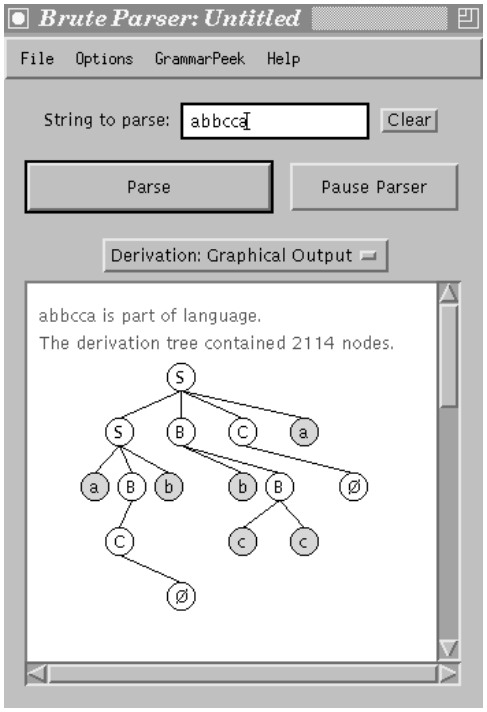

Figure 5: Parse Tree for $abbcca$

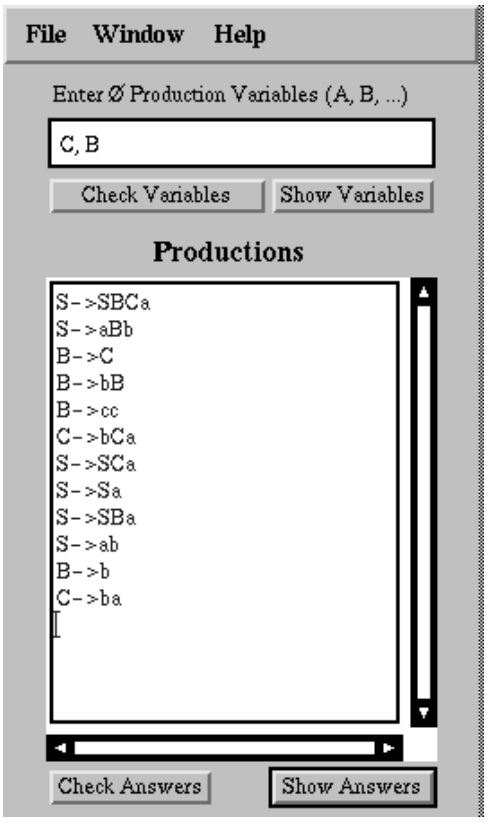

Figure 6: Removing Lambda Productions

Figure 7: PumpLemma for $a^{n/2}b^n$

and through steps, tries to prove that L is not regular. The user chooses the string w such that $w \in$ L and $|w| \geq m$. This string w should be partitioned into three parts: x, y and z such that $|xy| \leq m$, $|y| > 0$, and for all $i \geq 0$, xy^iz is in the language. If the string cannot be partitioned to meet these conditions, then the language L is not regular.

PumpLemma begins with the user typing in a language. Figure 7 shows the top portion of the PumpLemma window in which the user has typed in the language $a^{n/2}b^n$. Currently language selection is limited to ordered languages (i.e. $a^nb^n(ab)^n$, but not "the number of a's equals the number of b's"). When the user presses return, the language is parsed, and the user can continue if the language is legal. Next, the user can define the ranges of variables. For conventional purposes the only legal variables are n-s. If a language fails parsing due to poorly defined variables, an error message will appear.

The string w is entered in the same manner as the language, except the only legal exponents are combinations of m and integers. If the string is syntactically correct, it's length is then checked before proceeding. If the string is long enough, then the case list is automatically generated. In Figure 7 the user entered the string $w = a^{m/2}b^m$ and three cases were generated. The case list represents all the possible values for the y field. A well chosen string will yield fewer cases. (Note that in Figure 7, if the user had chosen $w = a^mb^{2m}$ there would be only one case.)

The user now selects a case (the case is highlighted) and fills in values for the substrings x, y and z. The contents of the substrings are stored for each case, so the user can switch between cases without losing information. The x, y, and z fields have their own set of exponents, ranging from h through l. After the y string is typed in, it is checked against all possible cases for a match. If it does not match any case, then the xyz cannot possibly equal w, and an error is generated. Otherwise, the case it matches becomes the current case. Figure 8 shows the bottom of the PumpLemma window for the example in Figure 7.

After the final substring field is entered the substrings are concatenated together and verified that they form w. The user then chooses i, the value to pump, and presses the run button and xy^iz is generated in the *Resulting String* box to the right. A message displays whether the resulting string is in the language. The case is colored red if the string is not in the language, and green if it is. The user then proceeds through all cases, trying to disprove them. When all cases are red, the language has been proven not to be regular.

Figure 8: Partitioning in PumpLemma

4 Use of Tools in Teaching

The tools described in this paper can be used in a Formal Languages and Automata Theory course, in lectures, labs and assignments. During lecture, the instructor can use a computer to demonstrate how to use the tools, and to solve problems with input from the class. Students can use the tools in labs or to work on assignments, and those students who want more practice can use the tools to reproduce examples illustrated in lecture or to create their own examples, receiving immediate feedback.

FLAP, LLparse and LRparse were used in CPS 140 at Duke in the spring of 1995 and 1996. Feedback from students in these courses was positive. They found the tools useful for testing out their answers. We plan to integrate the new tools into this course in the spring of 1997.

5 Related Work

In this section we describe several additional tools that have been developed for experimenting with other aspects of automata. Automata[12] developed on Mathematica allows one to experiment with finite automata (FA), in which one enters an FA in the textual formal notation, a 5-tuple, and can then automatically generate a list of strings in the language, convert an NFA to a DFA and many other useful operations. Hypercard Automata Simulation[6] allows one to enter an FA in the tabular format. Turing's World[2] allows one to experiment with many different forms of Turing Machines in a graphical format, including building Turing machine modules.

6 Conclusion and Future Work

We have developed a collection of instructional tools for experimenting with automata, grammars, and parsing for the formal languages course. Future plans include expanding PumpLemma to include the pumping lemma for context-free languages, and converting the LLparse and LRparse tools to Java. In addition, we plan to continue to develop new tools for experimenting with concepts in the formal languages area.

The tools in this paper are currently available on `http://www.cs.duke.edu/~rodger`

References

[1] A. Badre, C. Lewis, and J. Stasko, Empirically Evaluating the Use of Animations to Teach Algorithms, *Proceedings of the 1994 IEEE Symposium on Visual Languages*, p. 48-54, 1994.

[2] J. Barwise and J. Etchemedy, Turing's World, Stanford: CSLI Publications, New York: Cambridge University Press, 1993.

[3] S. Blythe, M. James, S. Rodger, LLparse and LRparse: Visual and Interactive Tools for Parsing, *Twenty-fifth SIGCSE Technical Symposium on Computer Science Education*, p. 208-212, 1994.

[4] M. Brown, ZEUS: A System for algorithm animation and multi-view editing. *Proceedings of the IEEE 1991 Workshop on Visual Languages*, p. 4-9, Kobe, Japan, Oct. 1991.

[5] P. Gloor, *AACE - Algorithm Animation for Computer Science Education*, IEEE Workshop on Visual Languages, p. 25-31, 1992.

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

[7] M. LoSacco, and S. Rodger, FLAP: A Tool for Drawing and Simulating Automata, *ED-MEDIA 93, World Conference on Educational Multimedia and Hypermedia*, p. 310-317, June 1993.

[8] S. Rodger, An Interactive Lecture Approach to Teaching Computer Science, *Proceedings of the Twenty-sixth SIGCSE Technical Symposium on Computer Science Education*, p.278-282, 1995.

[9] S. Rodger, Integrating Hands-On Work into the Formal Languages Course via Tools and Programming, *First International Workshop on Implementing Automata*, London, Ontario, 1996, (to appear).

[10] M. Procopiuc, O. Procopiuc, and S. Rodger, "Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP," *1996 Frontiers in Education Conference*, Salt Lake City, Utah, 1996, (to appear).

[11] J. Stasko, Tango: A Framework and System for Algorithm Animation, *IEEE Computer*, p.27-39, September 1990.

[12] K. Sutner, Implementing Finite State Machines, in *Computational Support for Discrete Mathematics, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Vol. 15, N. Dean and G. E. Shannon (ed.), American Mathematical Society, p. 347-363, 1992.