Visualization and Interaction in the Computer Science Formal Languages Course with JFLAP

Magdalena Procopiuc[1], Octavian Procopiuc and Susan H. Rodger[1]
Computer Science Department, Box 90129
Duke University
Durham, North Carolina 27708-0129
rodger@cs.duke.edu

ABSTRACT

The computer science formal languages course becomes a more traditional computer science course by integrating visual and interactive tools into the course, allowing students to gain hands-on experience with theoretical concepts. We explain how the tool JFLAP can be used in such a manner.

1 Introduction

The majority of undergraduate computer science courses have a natural programming component, yet there are a few required theoretical courses that are usually taught with no programming and no use of software. These theory courses are algorithms and formal languages. Since the development of tools for algorithm animation such as AACE [3], Xtango [9], Polka [10] and Zeus [2], the algorithms course is seeing the inclusion of animations into lectures [7, 8] and even a programming component with students developing animations [1] or experimenting with the analysis of existing programs.

Less common is the inclusion of software into the formal languages course. Recently the first Workshop on Implementing Automata [6] presented several tools for experimenting with automata. In this paper we address how such tools can enhance the formal languages course, changing it from a traditional mathematics course into a traditional "hands-on" computer science course. In particular, we describe the tool JFLAP and its use in such a course.

In Sections 2 and 3 we discuss the benefits of visualization and interaction in the formal languages course. In Section 4, we give an overview of JFLAP and in Section 5 we give an example of JFLAP to experiment with pushdown automata. In Section 6 we give concluding remarks and discuss future tools under development.

[1]Supported in part by the National Science Foundation's Division of Undergraduate Education through grants DUE-9596002 and DUE-9555084.

2 Visualization

A picture of a concept can be easier to understand then a textual representation. The material in an automata theory course has traditionally been presented in a textual manner, making it more likely that computer science students will have difficulty with the concepts. We present an example to illustrate the distinction between textual and visual presentations.

A description of a nondeterministic pushdown automaton (PDA) M can be described in a formal manner as a 7-tuple, $M = (Q, \Sigma, , , \delta, q_0, Z, F)$ where Q is a finite set of states, Σ is a finite set of tape symbols, , is a finite set of stack symbols, δ is a set of transitions represented by $\delta : Q \times \Sigma^* \times , ^* \rightarrow$ finite subsets of $Q \times , ^*$, q_0 is the start state ($q_0 \in Q$), Z is the start stack symbol ($Z \in ,$), and $F \subseteq Q$ is a set of final states.

An example of a specific PDA for those strings containing a's and b's with exactly twice as many a's as b's is given by describing each of the parts of the 7-tuple. In particular, M=($\{q_0, q_1\}$, $\{a, b\}$, $\{a, b, c, Z\}$, $\{(q_0, a, Z, q_0, cZ)$, $(q_0, a, a, q_0, \lambda)$, (q_0, b, cZ, q_0, aZ), (q_0, b, a, q_0, aaa), $(q_0, b, b, q_0, \lambda)$, (q_0, b, Z, q_0, aaZ), (q_0, b, cb, q_0, c), (q_0, a, c, q_0, b), $(q_0, \lambda, Z, q_1, Z)\}$, q_0, Z, $\{q_1\}$). Intentionally, there is a mistake in this description, can you find it easily?

Sutner [11] developed a tool called *automata* based on Mathematica for experimenting with finite automata (FA), in which one enters an FA in the formal notation above. Given an FA, one can automatically generate a list of strings in the language, convert an NFA to a DFA and many other useful operations. However, developing and entering an FA in this textual format is tedious and prone to errors.

A more organized representation is shown by the tool Hypercard Automata Simulation[4], in which one enters an FA in the tabular format. The table for the above PDA is given on the next page (the last three columns are not shown). The row headings represent states, the column headings represent input symbols and symbols to pop from the stack, and each entry represents the state to move to and the symbols to push on the stack.

In this case, it can still be difficult to determine relationships, since the table can be quite large.

	a,Z	a,a	b,cZ	b,a	b,b	b,Z
q_0	q_0,cZ	q_0,λ	q_0,aZ	q_0,aaa	q_0,λ	q_0,aaZ
q_1						

Although this tabular format looks organized, it is still tedious to examine relationships between states. A pictorial approach to representing this same PDA is shown in Figure 1. Here it is clear how states are related, which states are final states (double circles), which state is the start state (triangle adjacent to state), where loops exist, and other information. This approach, a transition diagram, is a preferable method for developing automata. [2] Even though this visual approach can be easier to understand than the textual approach, it also has some drawbacks. Drawn as a picture, it is tedious to determine correctness. We address this issue in the next section.

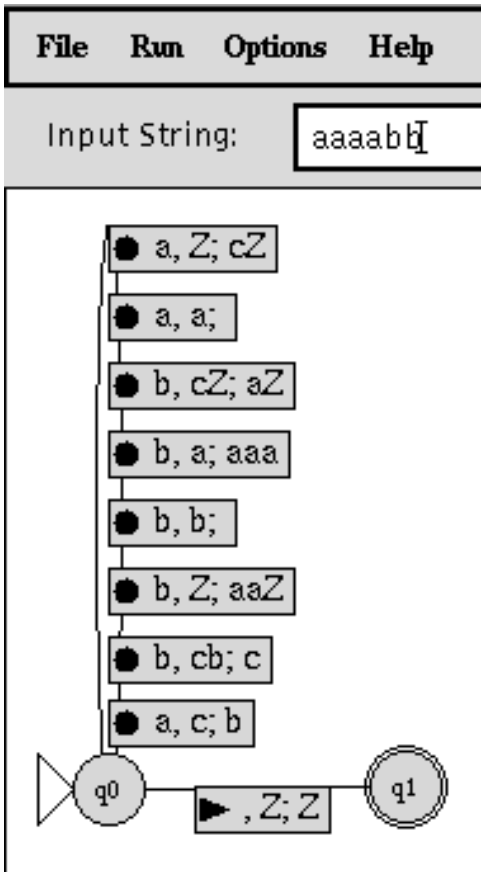

Figure 1: Pushdown Automaton

[2] Surprisingly, many textbooks on automata theory show finite automata in the transition diagram representation, but they do not use this representation for PDA or Turing machines, sticking to the more formal textual description.

3 Interaction

We have observed that the majority of automata drawn as transition diagrams using pencil and paper contain mistakes, since it is tedious to trace the run of input strings by hand. Furthermore, most students are not going to attempt additional exercises outside of assignments since they are unlikely to receive feedback. The automata drawings need to come alive, allowing for simulations on input strings so the designs can be tested and debugged.

Most computer science courses already have a natural interactive hands-on component, namely programming. Providing a tool for hands-on experience and receiving feedback in a formal languages course creates an atmosphere that resembles other computer science courses. Furthermore, combining experimentation with theory brings more meaning to the theory. Once a student has experimented with pushdown automata, the formal notation is going to be more familiar, making it easier to relate to formal proofs using such notation. For example, familiarity of PDA helps in understanding the proof that the class of languages defined by context-free grammars are equivalent to the class of languages defined by nondeterministic pushdown automata.

4 JFLAP - Interactive Tool for Automata

JFLAP (Java Formal Languages and Automata Package) is a Java implementation of FLAP [5], which allows one to graphically create a nondeterministic automaton, and then run the automaton on different input strings. As shown in Figure 2, the choices of automata available are finite automata, pushdown automata, and 1 and 2-tape Turing machines. A user can test an input string using either the fast run or step run. The fast run only shows the final result, acceptance of the string if there is some path that leads to a final state, and otherwise rejection. The step run allows one to step through the trace, seeing all the possible configurations at each step. If there are more than 12 configurations, the user must control the run by removing configurations that are unlikely to lead to acceptance.

5 An Example Using JFLAP

We provide an example of the PDA from Section 2, still containing an error. This PDA could either be handed out as a lab assignment to correct the error, or possibly might have been constructed by a student who is attempting to test and debug it. This PDA is suppose to represent the language L such that each string in L is composed of a's and b's, and there are twice as many a's as there are b's. For example, *aba* and *babaaa* are in L, but *abaa* and *bba* are not in L.

Figure 1 shows the incorrect PDA for L that has been created using JFLAP (the complete window is not

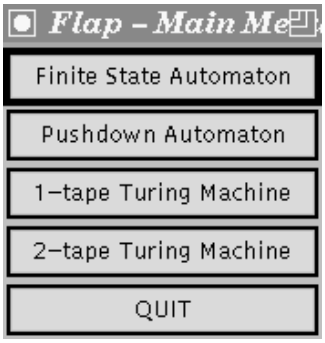

Figure 2: JFLAP main menu

shown), represented as a transition diagram. The labels on the arcs are in the form *A,B;C* where *A* is the string of symbols processed in the input string, *B* are the symbols popped from the stack and *C* are the symbols pushed onto the stack. For example *a,bc;de* means that *a* is the current symbol in the input string and is processed, *bc* is popped from the stack (with *b* on top of *c*), and *de* is pushed onto the stack (with *d* on top of *e*).

Students should create a set of test data, including strings that are in L and strings that are not in L, to determine correctness of the PDA. To test an input string in JFLAP, enter the string in the box labeled *Input String:* near the top of the window. To run the PDA on this string click on *Run*, and two options are shown, Step Run and Fast Run. Fast Run answers immediately whether or not the string was accepted. If it was, one can watch a trace of the path to acceptance. Step Run allows one to step through the execution, showing all possible configurations in the case of nondeterminism.

On this example, we ran the Fast Run on the test strings *aaaababb* (rejected it as it should), *bbaaaa* (all *b*'s before *a*'s, accepted), and *ababaa* (mixed *a*'s and *b*'s, accepted). Next we ran it on the string *aaaabb* and it was rejected, but it should have been accepted. At this point we ran it through the Step Run to determine where the error was.

Figure 3 shows the initial window for the step run. There is only one configuration at the start, containing the start state q_0, the complete input string *aaaabb* and an empty stack (Z is the bottom of stack marker). Not shown are the control buttons at the bottom of the window. Selecting *Step* results in showing all the configurations resulting from one step. If there are two many configurations (there is room for showing at most 12) then one must select certain configurations to remove (select the configurations and then select *Kill*). One can control this growth by freezing (*Freeze*) configurations (they cannot be expanded) and then thawing (*Thaw*) them at a later date. Selecting a specific configuration and *Trace* allows one to see a trace of the path from the start state to this configuration. Other buttons include *Help, Restart,* and *Quit.*

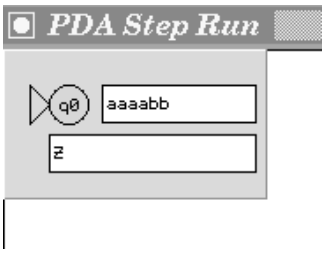

Figure 3: Initial Step Run Window

The result of selecting Step once is shown in Figure 4 (the complete run window is not shown since the rest of it is blank). This machine is nondeterministic as there are two possible configurations after one step. The correct configuration is the first one, one *a* has been processed in the input string and a *c* (representing that half a *b* is needed) is placed on the stack. The other configuration is in a final state, but none of the input has been processed. Figure 5 shows one step later. The second configuration has an X indicating that the previous configuration could not be expanded. This configuration will disappear in the next step. The first configuration is a continuation of the previous first configuration, a second *a* has been processed and the *c* on the stack has turned into a *b* representing that 2 *a*'s had been processed and a *b* is needed to match with them.

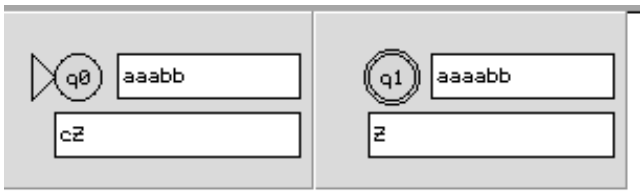

Figure 4: Step Run after 1 step

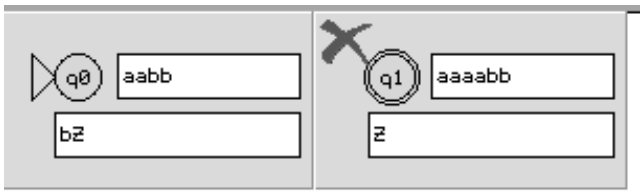

Figure 5: Step Run after 2 steps

Figure 6 shows that our previous first configuration could not be expanded. There is no transition to handle an *a* in the input string and a *b* as the top of the stack marker. We need to add such a transition to the PDA. At this point we quit the Step Run window and return

to the PDA building window. We add the rule $a, b; cb$ from state q_0 to state q_0. In this case, we need to leave the b on the stack, and add a c to indicate that we now need to see 1 1/2 b's to match the a count.

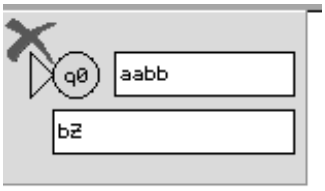

Figure 6: Step Run after 3 steps

With this correction, selecting the Fast Run for this string shows it is accepted, as illustrated in Figure 7. By selecting Yes, we can step through a trace that is similar to the step run, but shows only the configurations on the acceptance path.

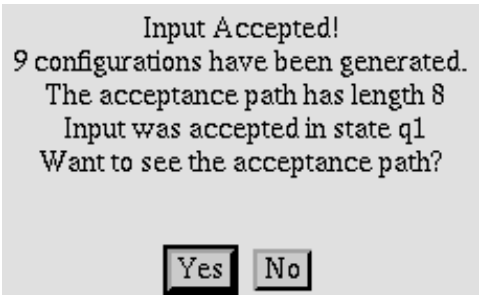

Figure 7: String Accepted

6 Conclusion and Future Work

We have shown how visualization and interaction can be integrated into a formal languages course, using JFLAP as an example. Such tools provide a visual picture, making it easier to see relationships between objects, and provides interaction, allowing the student to experiment with the picture and receive immediate feedback.

We are currently developing tools for parsing different types of grammars, for transforming grammars, and for exploring the pumping lemma. These tools will be available on the web address below.

7 Acknowledgements and Obtaining Tool

Many students from Rensselaer and Duke have contributed to FLAP and JFLAP. Dan Caugherty and Mark LoSacco created the original version of FLAP, Greg Badros made improvements to FLAP, Ben Hardekopf worked on the preliminary design of JFLAP, and Magdalena and Octavian Procopiuc implemented JFLAP.

JFLAP, FLAP, and other tools for formal languages and automata theory are available via anonymous ftp on `http://www.cs.duke.edu/~rodger/tools/tools.html`

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] 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.

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

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

[5] 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.

[6] Proceedings of the Workshop on Implementing Automata, London, Ontario, August 1996, to be published by Springer-Verlag.

[7] 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.

[8] S. Rodger, "Integrating Animations into Courses," *ACM SIGCSE/SIGCUE Conference on Integrating Technology in Computer Science Education,* Barcelona, Spain, 1996 (to appear).

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

[10] J. Stasko and E. Kraemer, A Methodology for Building Application-Specific Visualizations of Parallel Programs, *Journal of Parallel and Distributed Computing*, Vol. 18, p. 258-264, 1993.

[11] 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.