Integrating Hands-on Work into the Formal Languages Course via Tools and Programming

Susan H. Rodger[1]
Department of Computer Science
Duke University
Durham, NC 27708-0129
email: rodger@cs.duke.edu
http://www.cs.duke.edu/~rodger

Abstract

Integrating hands-on practice into the automata and formal languages course aids in transforming the course from a traditional mathematics course into a traditional computer science course, and makes the material more interesting from a teaching and learning perspective. The hands-on practice we integrate into our course comes in the form of interactive and visual tools and programming assignments. The tools we use are FLAP, a tool for constructing and simulating several types of nondeterministic automata, and LLparse and LRparse, tools for constructing parse tables and animating the parsing of strings. For the programming component, our students write an LR(1) parser for a simple programming language and use the tool Xtango to animate programs in this new language.

1 Introduction

Successful learning requires some form of feedback, and the sooner the feedback comes the more useful it is. When feedback is delayed, for example when written homework is not returned right away, the problems can be forgotten, or worse a student may not realize misunderstandings, yet continue to build on them. Immediate feedback is the most helpful, as it allows one to move forward confidently when correctness is realized, or to seek help from the instructor when misunderstandings are realized.

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

Traditionally, automata theory has been taught with a long turnaround time of feedback. For example, students write solutions for a finite automaton or a Turing machine by writing on paper formal notation for all parts of the machine, including a table of transitions. An improvement is writing, also on paper, a solution in the form of a transition diagram. This pictorial representation clearly shows relationships between states. However, in the same manner that a program is usually not correct the first time it is compiled and run, our experience has been that hand drawn automaton are often incorrect.

It has been shown in studies of algorithm courses that hands-on practice with animations aids in understanding. Animations of algorithms clearly excite students, but it is difficult to measure whether they help students in understanding. In [2], students watched animations to aid in learning algorithms, but an empirical study showed that the animations only slightly assisted student understanding. However, the study in [3] showed that allowing students to build their own animation led to a higher understanding of an algorithm.

This paper explains how we have integrated immediate feedback through hands-on practice into CPS 140, the undergraduate automata and formal languages course at Duke University. Immediate feedback is achieved through visual and interactive tools and programming assignments. Tools are used during lectures by the instructor [9] for explaining concepts, solving problems, and as an introduction to using the tool. Tools are used in labs and homework by students to solve problems. The tools we use include FLAP [8, 5], a tool for constructing and simulating several types of nondeterministic automata, and LLparse and LRparse [4], instructional tools for constructing parse tables and animating the parsing of strings. For the programming component, our students write an LR(1) parser for a simple programming language and use the tool Xtango [10] to animate programs in this language.

In Section 2 we describe the tool FLAP and describe how we have integrated it into the course CPS 140. In Section 3 we describe the tools LLparse and LRparse and their use in CPS 140. In Section 4 we describe the LR(1) parser programming assignment and the use of Xtango for animating programs. Section 5 describes the evaluation of these tools in the course CPS 140, and Section 6 gives concluding remarks.

2 FLAP

In this section we describe FLAP [8, 5], and several assignments and lectures using FLAP.

2.1 Overview of FLAP

FLAP is a visual tool for designing and simulating several variations of finite automata (FA), pushdown automata (PDA), Turing machines (TM), and two-tape Turing machines (TTM), including nondeterministic versions of these machines. In the building window of FLAP, one draws a graphical representation of the transition diagram. States are automatically created by clicking the mouse, and arcs between states are created by clicking and dragging the mouse. In the latter case, a transition label automatically appears that can be filled in. There are options for making a state final or nonfinal, moving a state or label, deleting a state or label, saving or retrieving machines, and identifying nondeterministic states.

When the drawing of the automaton is complete, one can simulate the automaton on any input. A user types in the input string and selects either fast simulation or a slower step-by-step simulation. In the fast simulation, a complete tree of configurations is generated with the root containing the start state, input string and any other information relevant to a particular automaton. This configuration tree is not displayed, but rather a message appears telling the user whether or not the input string was accepted (if some path in the tree leads to acceptance). If the string is accepted, the user has the option of stepping through an animation of the configurations on the path in the configuration tree that led to acceptance. For example, the animation for a PDA initially shows a picture of the starting configuration containing the start state, the input string and an empty stack. Each step shows the new state, the symbols remaining to be processed in the input string, and the current contents of the stack.

In the slow simulation, the user steps through the simulation at his or her own pace, seeing all the configurations generated on the current level of the configuration tree. This run starts by showing a picture of the starting configuration. Each step displays the next level of configurations generated in the tree, with a limit of 12 configurations displayed at any

time. If there are more than 12 current configurations at any level, the user has to control the display by selecting configurations and either removing them or freezing them. In the latter case they cannot be expanded until they are thawed. At any point in a simulation, a particular configuration can be selected and traced to see how this configuration was reached.

FLAP has been designed with a common interface so that once a student has studied one type of automaton, it is easy to construct another type. In addition, the definitions of these machines are general so that several variations of each machine can be examined. This general definition allows the tool to be used along with any automata theory textbook [6, 7].

2.2 Definitions of automata

FLAP recognizes a general definition of each type of automaton so that several variations of each can be studied.

Definition 3.1: A nondeterministic finite automaton M is represented by the 5-tuple $M = (Q, \Sigma, \delta, q_0, F)$, where Q is a finite set of states, Σ is a finite set of input symbols, δ is a set of transitions represented by $\delta : Q \times \Sigma^* \rightarrow 2^Q$, q_0 is the start state ($q_0 \in Q$), and F is a set of final states ($F \subseteq Q$).

The definition is general so that other variations of FA can be examined. One variation is a deterministic FA. Another variation restricts the number of input symbols that can be processed in a transition to one symbol. In all variations, a FA accepts an input string if there is a path from the start state to a final state that recognizes this input.

Definition 3.2: A nondeterministic pushdown automaton M is represented by the 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.

The definition is general so that variations of a PDA can be examined. Other variations include deterministic PDA, restricting the number of input symbols to one or zero (λ) symbols to be processed, and restricting the number of symbols popped on each transition to

be exactly one. There are two definitions of the acceptance of an input string, providing additional variations of PDA that can be studied. Acceptance is based on either reaching a final state or the stack becoming empty.

Definition 3.3: A nondeterministic Turing machine M is represented by the 7-tuple $M = (Q, \Sigma, , , \delta, B, q_0, F)$, where Q is a finite set of states, Σ is the input alphabet, , is the tape alphabet (with $\Sigma \subseteq , -\{B\}$), δ is a set of transitions represented by $\delta : Q \times , \rightarrow 2^{Q \times \Gamma \times \{R,L,S\}}$, B is a special symbol denoting a blank on the tape, q_0 is the start state ($q_0 \in Q$), and $F \subseteq Q$ is a set of final states. The symbols R, L and S denote directions Right, Left, and Stay.

Other variations of the TM are a deterministic TM, a two-tape TM (δ is represented by $\delta : Q \times , \times , \rightarrow 2^{Q \times \Gamma \times \Gamma \times \{R,L,S\} \times \{R,L,S\}}$) and a variation when the movement of the tape head is restricted to always moving, either R or L. In all cases, acceptance of an input string is based on reaching a final state.

In FLAP, the transition format for all machines is:

FA: $< in\_string >$

PDA: $< in\_string >$ **,** $< pop\_string >$ **;** $< push\_string >$

TM: $< in\_symbol >$ **;** $< write\_symbol >$ **,** $< dir\_symbol >$

where $< in\_string >$ is the (zero, one or more) symbols that must be present on the input tape if the transition is to be applied, $< pop\_string >$ is the symbols that must be present at the top of the stack which are to be popped off, $< push\_string >$ is the symbols that are to be pushed on the stack, $< in\_symbol >$ is a single symbol which must be the current symbol on the input tape if the transition is to be taken; $< write\_symbol >$ is the single symbol which is written to the current position on the input tape; and $< dir\_symbol > \in \{R, L, S\}$, depending on whether the read head should move Right, move Left, or Stay put. For $< pop\_string >$ and $< push\_string >$, the leftmost symbol corresponds to the top of the stack. The two-tape TM has two connected labels (one for each tape), both in the TM format above.

Figure 1: FLAP menu

2.3 Examples using FLAP

2.3.1 An NPDA Example

In this section we first show an example of constructing and simulating a nondeterministic PDA, and then give brief examples of other automata constructed using FLAP.

In the first example, a nondeterministic PDA is constructed for the language

$$\Sigma = \{a, b\},\ \mathrm{L}=\{a^n b^m \mid m > 0, m \leq n \leq 3m\}.$$

The menu for FLAP is shown in Figure 1. From this menu one selects the type of automaton to construct. Figure 2 shows the constructed PDA for this language. The input string *aaaaaaaabbb* has been entered in the input box, and acceptance by final state has been selected.

The fast run results for this PDA and input string results in acceptance as shown in the message in Figure 3. The total number of nodes in this configuration tree is 75, and the length of the path from the root (or starting configuration) to an acceptance configuration (final state) is 18. By selecting *yes*, an animation of configurations in the acceptance path is shown, starting with the start state (see Figure 4) and ending in the final state (see Figure 5). The top rectangle in each figure is the current symbols to process in the input string and the bottom rectangle is the stack with the top of the stack on the left side.

Alternatively, one runs an automaton in the step-by-step mode. In this case a Run Window appears showing the starting configuration. As the user steps through the simulation,

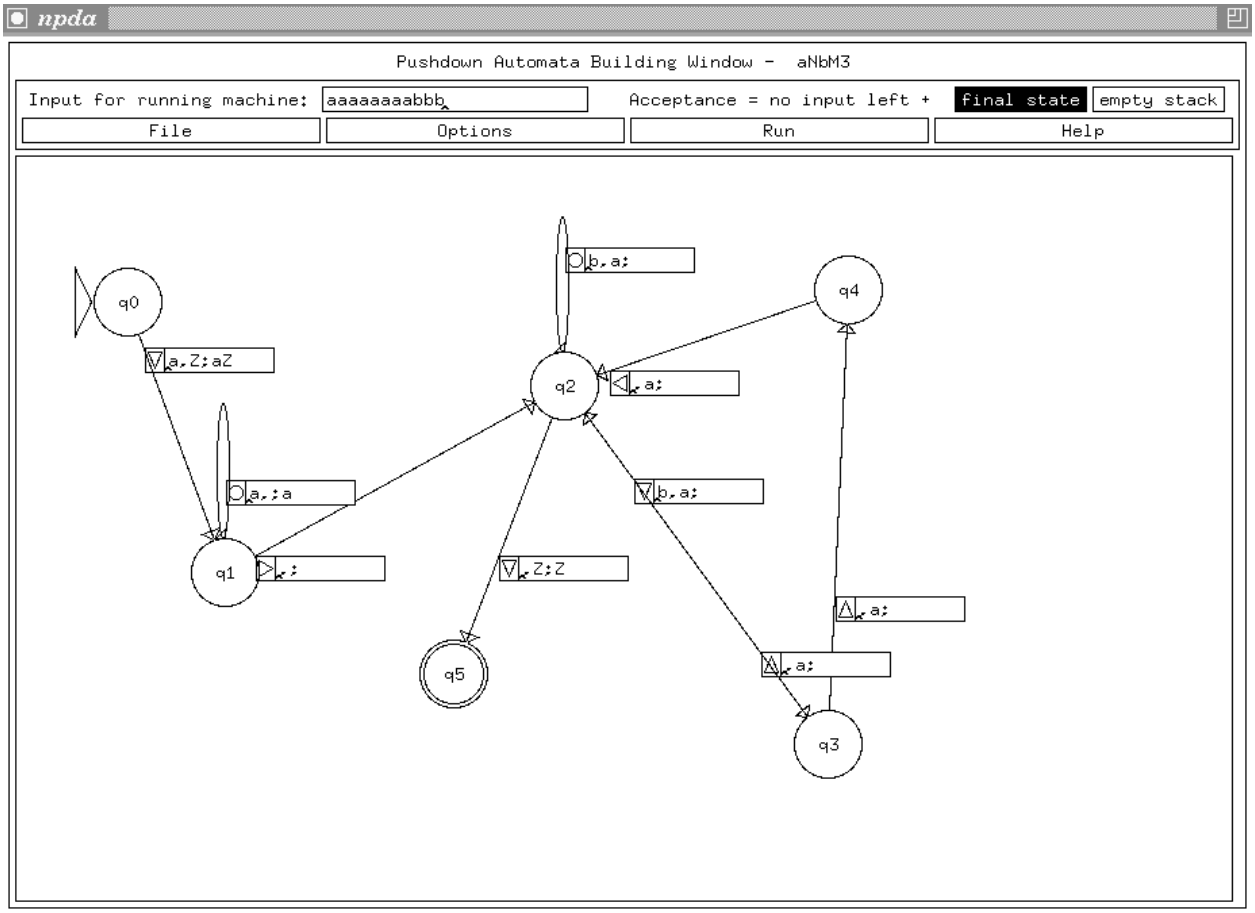

Figure 2: NPDA for $\{a^n b^m \mid n > 0, n \leq m \leq 3n\}$

all current configurations (up to 12) are shown. Figure 6 shows a snapshot of the run of the PDA in Figure 2 several steps into the trace. At this point there are 6 possible configurations.

2.3.2 Additional Examples

Below are a list of languages for which automata have been built using FLAP. FLAP is an instructional tool and is not meant to design automata with a large number of states.

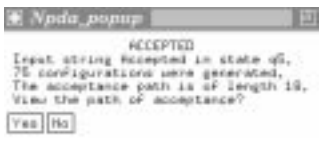

Figure 3: NPDA Acceptance Information

Figure 4: PDA Path Trace Starting Configuration

Figure 5: PDA Path Trace Acceptance Configuration

There is a limit to what can be physically drawn in the drawing window, plus it would be too tedious to construct large automata. The lists of languages below were given as lab assignments or homework assignments in CPS 140.

An assignment to build FA is listed below. For this assignment, the fourth one is really too large to use with FLAP.

1. $\Sigma = \{0, 1, 2\}$, L = $\{w \in \Sigma^* \mid w$ is a nonnegative integer base 3 $\}$. For example, 0, 1002, and 221 are in L, but 000, and 020 are not in L.

2. $\Sigma = \{a, b\}$, L = $\{w \in \Sigma^* \mid w$ has an even number of a's and an odd number of b's $\}$. For example, aabaa, and bbaba are in L, aabbbaba and bbab are not in L.

3. $\Sigma = \{a, b\}$, L = $\{w \in \Sigma^* \mid |w| \bmod 3 = 1\}$

4. $\Sigma = \{a, b\}$, L = $\{w \in \Sigma^* \mid |w| >= 4$ and every substring of four symbols has at least 2 b's$\}$. For example, aabbaba is in L (every substring of 4 consecutive symbols has 2

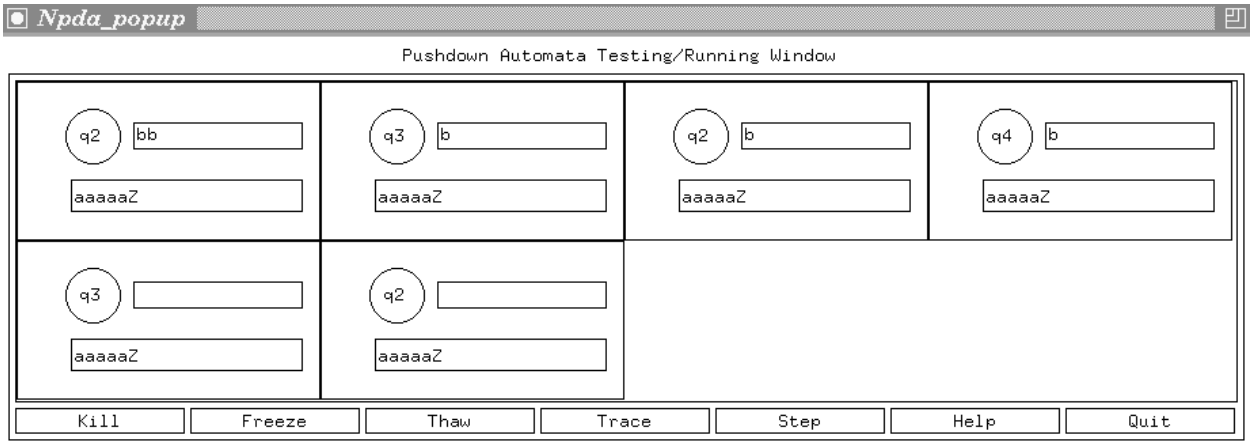

Figure 6: PDA Running Window

b's) and aaab is not in L. The string ababaa is not in L because the substring abaa does not contain 2 b's.

An assignment to build PDA's is listed below. One can select to accept by either final state or empty stack. Students seem to have difficulty with the nondeterministic machines (2 and 3 below).

1. $\Sigma = \{a, b\}$, L = $\{a^n b^{3n} \mid n > 0\}$. For example, abbb and aabbbbbb are in L. Accept by *Final State.*

2. $\Sigma = \{a, b\}$, L = $\{w \in \Sigma^* \mid w = w^R$ and w is of odd length $\}$. For example, bbabb and ababa are in L, aabbaa and bbbab are not in L. Accept by *Empty Stack.*

3. $\Sigma = \{a, b\}$, L = $\{a^n b^m \mid m > 0, m \leq n \leq 3m\}$. For example, aaabb and aaaaabb are in L, and aabbb and aaaaaaabb are not in L. Accept by *Final State.*

4. $\Sigma = \{a, b\}$, L = $\{w \in \Sigma^* \mid$ the number of b's is twice the number of a's $\}$. For example, bababb, and abb are in L, aabbbaba and bbab are not in L. Accept by *Final State.*

An assignment to build TM's is listed below. Again, students seem to have difficulty with the nondeterministic machine (3 below).

1. $\Sigma = \{a, b\}$, L = $\{a^n b^n \mid n > 0\}$.

2. $\Sigma = \{a, b\}$, L = $\{a^m b^n c^n \mid n > 0, m > n\}$.

3. $\Sigma = \{a, b\}$, L = $\{ww^R w \mid w \in \Sigma^*\}$

   For example, *aabbaaaab* is in L, where $w = aab$

An assignment to build two-tape TM's included the following language.

1. $\Sigma = \{a, b\}$, L = $\{a^m b^n c^n \mid n > 0, m > n\}$.

2.4 Examples using FLAP in Lecture

We use FLAP projected to a screen during lectures to solve problems and to aid in the discussion of proofs. In lectures, students are given assignments to build an automaton. After working for a few minutes on their own, the class as a whole tells the instructor which states and labels to construct in class, and the instructor constructs the automaton using FLAP. This is useful in both showing students how to use FLAP and showing students the thinking process in designing automata.

We examine proofs of converting three types of grammars into an NPDA, a context-free grammar in Greibach normal form, an LL(1) grammar, and an LR(1) grammar. For each of these, we work through a simple example constructing the NPDA and testing input strings. For the latter two, we proceed to talk in depth about the LL(1) and LR(1) parsing process. Since the PDA created is nondeterministic, we can run these machines on input strings in the slow step mode and talk about which lookahead to accept, and thus which configuration to try next. This aids in understanding both parsing processes.

3 LLparse and LRparse

In this section we give an overview of LLparse and LRparse and give an example using LRparse.

3.1 Overview of LLparse and LRparse

LLparse and LRparse [4] are interactive and visual instructional tools for constructing LL(1) and LR(1) parse tables [1] 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.

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.

For both tools, the first three windows encountered are the same. 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 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 [8]. 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) parse 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) parse table.

All the windows contain a subset of the following buttons. The *DONE* button is used to announce that a user has finished typing in input in a window. At that point, the user is informed whether or not the window's entries are correct and if not correct, the incorrect entries are highlighted. The correctness of the DFA is more complicated and thus just highlights one error at a time. The *SHOW* button automatically fills in the window with the correct entries and in the DFA window automatically constructs the DFA. This is useful for frustrated students. The *PRINT* button prints to paper or to a file all of the work done up to the current window, the *HELP* button provides online help, and the *QUIT* or *RETURN* button allow the user to return to the initial window to start over.

In both tools, after the grammar is entered, the parse table is immediately calculated but not shown. Thus, if it is determined that the grammar is not an LL(1) or LR(1) grammar, the user is informed and given the option to continue or change the grammar. The user can continue with the incorrect grammar up to the calculation of the parse table, at which point some position in the table will have multiple entries.

3.2 Example of LRparse

Here is an example using LRparse to construct an LRparse table for the following LR(1) grammar.

$$\begin{aligned} S &\rightarrow aSAb \mid c \\ A &\rightarrow cA \mid \lambda \end{aligned}$$

Upon starting LRparse, the grammar window in Figure 7 initially would appear empty, and the user would type in the grammar. The arrow and λ are typed using "CTRL ." (note the ">" is on the same keypad as the ".") and "CTRL l". This is explained in the online help that is provided. After selecting **Done**, the FIRST sets window appears. In Figure 8

Figure 7: Enter Grammar in LRparse

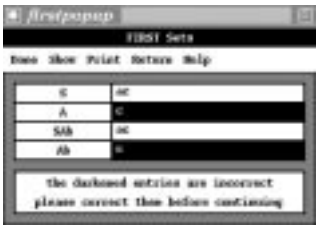

Figure 8: First Sets in LRparse with a Mistake

the user has been informed that the FIRST sets for A and Ab are incorrect. The FIRST set for A should be "cλ", and the FIRST set for Ab should be "cb". 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. The DFA differs from the DFA in FLAP in that an item set must be entered for each state. A small window pops up after selecting a state, and one enters the marked rules for that state. This window can stay up or be hidden. After completing the correct DFA, selecting done brings up the parse table window. Figure 9 shows the parse table for the grammar in Figure 7. By selecting row 3, the item sets for state q3 can be displayed and removed when desired.

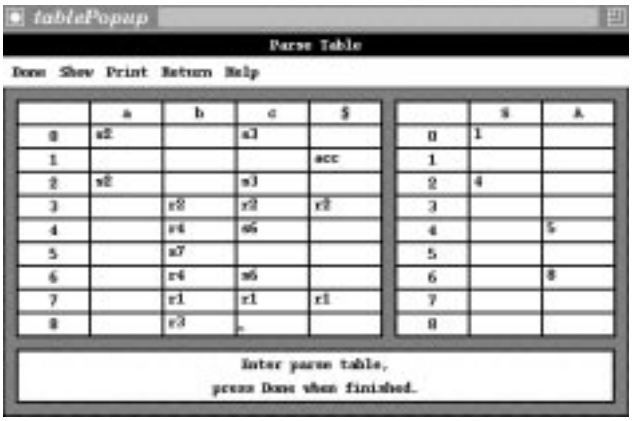

Figure 9: LRparse Table

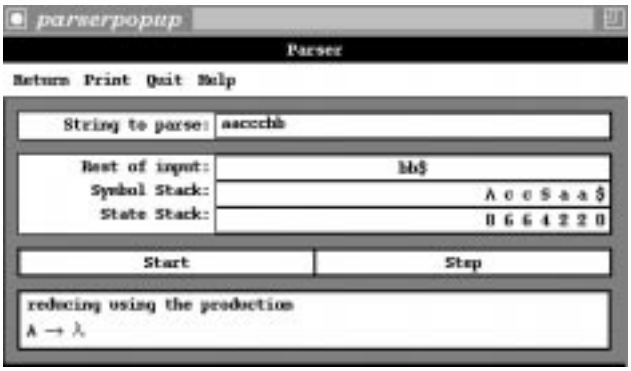

Figure 10: LRparse Parsing Example

When the parse table is correct, the final window displayed is the parsing window. Figure 10 shows part of the trace of the string *aacccbb*. 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.

3.3 Using LLparse and LRparse in a Course

We use LLparse and LRparse during lecture to increase the interaction in the classroom. After a short lecture on a topic, we use the tools to work through a problem. For example, after describing how to calculate FIRST sets, students work through an example at their

desk, and then we try out someone's solution using the tool, which tells us if there is a mistake or not. After creating parse tables, students suggest a string and we step through the parsing of the string.

Students use LLparse and LRparse outside of lecture in a homework assignment and also use them heavily to study for exams to make sure they understand the algorithms for constructing the parse tables.

4 Interpreter Programming Assignment

Writing an interpreter is beneficial in many ways. Students gain experience in seeing how material from the automata theory course is useful in the real world, they see the usefulness of material from their previous data structures course, and they gain additional programming experience. In CPS 140, students write an LR(1) parser in three phases: the scanner, the parser, and the syntax tree. In the third phase, the tool Xtango [10] is used to animate the running of a program in the new language.

Each phase is built from scratch. The new programming language is very simple and the focus is on how FA's and PDA's are used in the parsing process. Tools such as lex and yacc are not used, but they are discussed at the end of the semester so students are aware of them and know to use them if they were creating another language.

In the first phase, students write a scanner from scratch and set up a symbol table (or hash table). Their output is a list of tokens and their type. In the second phase, the parse table is given to students (in a file) since the parse table can be quite large (in the example language below it is 41 rows and 24 columns). Students focus on the LR(1) parsing process. They must create a stack, process operations (shift, reduce, accept and error), and print out the rules found (in reverse order) if a program is syntactically correct. In the third phase, a syntax tree is created and then the program is *interpreted* by walking through the syntax tree.

4.1 An Example Language and the use of Xtango

Every semester a different simple programming language is created. In the spring 1996 semester, students wrote an interpreter for the RIMLAN programming language, a simple language for moving robots around a room that contains six types of statements.

Statement	Meaning
begin i j *stmts* halt	program definition - defines starting room of height i and width j
robot v a b ;	draw a robot v at position (a, b)
obstacle a b ;	draw an obstacle at position (a, b)
add a to v ;	add statement
move v d a ;	move the robot v, a spaces in direction d
$v = a$;	an assignment statement
do *stmts* until $a > b$;	Execute *stmts*, if $a \leq b$ then repeat

where v is a variable, a and b are either variables or integers, i and j are integers, d is a direction (north, south, east or west) and *stmts* represents 1 or more valid statements.

Below is a sample RIMLAN program that creates a robot named bob, two obstacles, and then makes bob circle the perimeter of the rectangle formed by the two obstacles ten times.

```
begin 30 40
  robot bob 2 4 ;
  obstacle 4 5 ;
  obstacle 7 12 ;
  horizontal = 4 ;
  vertical = 8 ;
  move bob east 6 ;
  j = 1 ;
  do
     move bob north vertical ;
     move bob west horizontal ;
     move bob south vertical ;
     move bob east horizontal ;
     add 1 to j ;
  until j > 10 ;
halt
```

Visualizing the running of RIMLAN programs is helpful to see if robots have crashed into obstacles or other robots. We use the tool Xtango to create simple animations. In particular, we use the Xtango animator (an interpreter) which allows one to create simple objects such as rectangles, circles, points and lines. Each object created has a tag associated with it, and can be referred to in order to move the object. Thus, it is very simple to create obstacles (squares) and robots (circles) and then move the robots.

5 Evaluation

We have been teaching automata theory for seven years (five years at Rensselaer Polytechnic Institute and the past two years at Duke University). The first two years the programming assignment was part of the course, but no tools were used and the programming assignment did not have an animation component. In the remaining years, the tools have been integrated into lectures and assignments, and this year FLAP was also used in a separate computer lab.

FLAP has undergone changes and improvements over the years. Students have always been enthusiastic about using it to design and run automata. The majority of students using FLAP in CPS 140 in the spring of 1996 stated in evaluations that they found FLAP very easy to use and were impressed with it. These students used FLAP in three lab periods. The first lab was to design FA, the second lab to design PDA's and the third lab to design TM's and two-tape TM's. After the first lab, about half the students still preferred to write the FA on paper first so they could write comments beside states, but then they all found it very beneficial to be able to run the programs.

Comments after the first lab using FLAP include the following.

- "The flap program was easy to use. It took all of two seconds to learn."
- "The benefit of using FLAP is that you can easily test your design. This benefit outweighs the hassle of having to use a program to design the DFA. By using this program you can be sure your tests are conducted correctly and they don't take any time to run."
- "I found myself creating the basic structure for most on paper, and then testing them thoroughly via FLAP. FLAP was a lot faster (and more accurate) at testing the FA's

than pencil and paper."

- "All in all, I was actually impressed with FLAP, and I found it to be easier to think with than the pencil-and-paper method, and easier to understand the finished product than with hand-drawn FA."

Comments after the third lab using FLAP include the following.

- "Flap was very easy to use. I think the fact that the Turing Machines are more complicated to test on paper makes flap a very good tool."
- "I still used paper to lay the groundwork for my TM's. Then, I transferred them to Flap and modified them when they didn't work. So, it is good that Flap is there to catch your errors."
- "Creating Turing machines was actually kind of fun. It required a little more thought. FLAP definitely came in handy here to test all the different cases that you could get. Made things a lot more efficient."
- "FLAP is getting to be more worthwhile as the complexity of the things we are modelling increases. The trace function is incredibly useful, especially the two taped TM."
- "I now find it much easier to create FA on FLAP than by hand."
- "I like FLAP more as I get used to it. I find it to be very valuable when designing complex machines. It makes things much easier to test."

6 Conclusion and Future Work

We have transformed the Formal Languages and Automata Theory course from a course with a slow and small amount of feedback into a course using visual and interactive tools, increasing the amount and quickness of the feedback. The tool FLAP allows one to create and simulate several types of automata, and the tools LLparse and LRparse allow one to create parse tables and parse arbitrary input strings. These tools plus a three part programming assignment writing an LR(1) parser greatly enhance the amount of hands-on work in this course. Students are very positive about these tools and they enjoy being able to test out their designs of automata using FLAP, and checking their answers for constructing a parse table for a particular grammar using LLparse or LRparse. We have not converted all

assignments into this format. We still give some written homework assignments, but we are currently investigating how to create tools for all of our assignments.

Current work on these tools include creating a Java version of FLAP called JFLAP and adding parse trees to LLparse and LRparse. We are also investigating tools for experimenting with grammars. These tools are available via anonymous ftp from our web site http://www.cs.duke.edu/~rodger

7 Acknowledgements

Many students from Rensselaer and Duke have worked on designing and developing FLAP, LLparse, and LRparse including Dan Caugherty, Mark LoSacco, Greg Badros, Magda Procopiuc, Octavian Procopiuc, Mike James, Steve Blythe, Ugur Dogrosou, and Edwin Tsang.

References

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

[2] A. Badre, C. Lewis, and J. Stasko, Do algorithm animations assist learning? An empirical study and analysis. *INTERCHI 93 Conference Proceedings: Human Factors in Computing Systems*, p 61-66, ACM Press, April 1993.

[3] 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.

[4] 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.

[5] D. Caugherty, and S. H. Rodger, NPDA: A Tool for Visualizing and Simulating Nondeterministic Pushdown Automata, 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. 365-377, 1994..

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

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

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

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

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