# 将自动机理论转化为一门实践课程\*

Susan H. Rodger

Computer Science

Duke University

Durham, NC 27708

Bart Bressler

Computer Science

Duke University

Durham, NC 27708

Thomas Finley

Computer Science

Cornell University

Ithaca, NY 14853

Stephen Reading

Computer Science

Duke University

Durham, NC 27708

rodger@cs.duke.edu

**摘要**

We present a hands-on approach to problem solving in the

formal languages and automata theory course. Using the

tool JFLAP, students can solve a wide range of problems

that are tedious to solve using pencil and paper. In combi

nation with the more traditional theory problems, students

study a wider-range of problems on a topic. Thus, students

explore the formal languages and automata concepts compu

tationally and visually with JFLAP, and theoretically with

out JFLAP. In addition, we present a new feature in JFLAP,

Turing machine building blocks. One can now build com

plex Turing machines by using other Turing machines as

components or building blocks.

**Categories and Subject Descriptors**

F.4.3 [**Theory of Computation**]: Mathematical Logic and

Formal Languages Formal Languages; D.1.7 [**Software**]: Pro

gramming Techniques Visual Programming

**General Terms**

Theory

**Keywords**

JFLAP, automata, pushdown automata, Turing machine,

grammar, SLR parsing, LL parsing, L-system

**1. INTRODUCTION**

Traditionally, the formal languages and automata (FLA)

courses have assigned pencil and paper homework exercises

of two types: proofs and construction exercises. The second

of these types of problems are limited to small examples.

Even on a moderate-size example of constructing an au

tomaton with eight states, students are unlikely to do much

*∗*The work of all four authors was supported in part by

the National Science Foundation through grant NSF DUE

CCLI-EMD 0442513.

Permission to make digital or hard copies of all or part of this work for

personal or classroom use is granted without fee provided that copies are

not made or distributed for profifit or commercial advantage and that copies

bear this notice and the full citation on the fifirst page. To copy otherwise, to

republish, to post on servers or to redistribute to lists, requires prior specifific

permission and/or a fee.

*SIGCSE’06* March 1–5, 2006, Houston, Texas, USA.

Copyright 2006 ACM 1-59593-259-3/06/0003 ...$5.00.

testing as it is tedious to trace by hand. Grading such prob

lems is similarly slow and error prone.

We describe a hands-on approach to the FLA course that

allows students to explore many of the FLA concepts com

putationally and visually using the tool JFLAP. We are *not*

advocating to remove the proof type of exercises from the

course, but rather to supplement them with hands-on explo

rations of related topics. For example, consider the problem

of proving that if a language *L* is regular, then so is the lan

guage *LR* (the language with all strings from L reversed).

This is a common proof-type problem given in this course.

For some students, before proving this, it might be helpful

to visualize an example fifirst. They start with some regular

language L, build a deterministic fifinite automaton (DFA)

for it, and then convert this DFA into a DFA for *LR*. They

must create test data for both DFA to convince themselves

that the DFA are correct. This approach relates the FLA

course more in line with the majority of their computer sci

ence courses which are hands on and involve constructing,

debugging and testing.

Others have taken similar hands-on approaches to the

FLA course, but focus on a smaller number of topics. Tur

ing’s World[1] allows one to create and experiment with

Turing machines and automata. The focus is on Turing ma

chines and submachines. Taylor[8] uses the software *Deus Ex*

*Machina* letting users experiment with Turing machines, fifi-

nite automata, pushdown automata, and several other types

of automata. Forlan[7] is a toolset used in conjunction with

Standard ML for creating and experimenting with fifinite au

tomata, regular expressions and grammars. Language Emu

lator[9] is a toolkit for a number of forms of regular lan

guages including Moore and Mealy machines, and many

types of translations between the forms. Grinder[4] has de

veloped the FSA Simulator for experimenting with fifinite

state automata. It is part of Webworks[3], an extensive hy

pertextbook under development that will cover many topics

in automata theory. It incorporates text, sound, pictures, il

lustrations, slide shows, video clips and active learning mod

els.

In this paper we present an overview of JFLAP and then

give several examples of how it can be used computationally

and visually to explore FLA concepts in depth. We then

present new features of JFLAP including the ability to build

more interesting Turing machines with building blocks. Tur

ing machines built can be named and reused as a component

in another Turing machine. We conclude with an evaluation

of JFLAP’s use around the world and a description of future

work.**2. AN OVERVIEW OF JFLAP**

JFLAP[5, 6, 2] is an instructional tool for creating and

experimenting with several types of nondeterministic au

tomata, grammars, regular expressions, L-systems, and ex

perimenting with the conversion from one structure to an

other. With JFLAP one can build a fifinite automaton (FA),

a pushdown automaton (PDA), or a multi-tape Turing ma

chine (TM) and observe its simulation on several inputs.

One can enter a regular grammar, a context-free grammar

(CFG), or an unrestricted grammar and observe the brute

force parsing of strings in this grammar with the result

shown either as a derivation or a parse tree.

JFLAP allows the conversion from one form to another.

One can convert an NFA to a DFA to a minimal state DFA,

convert between NFA and regular grammars, or convert be

tween NFA and regular expressions. One can convert a

nondeterministic PDA (NPDA) to a CFG or a CFG to an

NPDA. One can convert a CFG to Chomsky Normal Form,

along the way removing *λ*-productions, unit productions and

useless productions. One can convert a CFG to either an

LL(1) or SLR(1) parse table and then parse strings in the

language. Finally, one can create an L-system, a difffferent

type of grammar that can be used for modeling the growth

of plants and fractals.

**3. PROBLEM SOLVING WITH JFLAP**

The previous section described the construction and test

ing of automata and grammars in JFLAP, and the conver

sion from one form to another. That in itself allows for users

to build and test automata more easily than can be created

using pencil and paper.

We now describe several other types of problem solving

with JFLAP that are tedious to do with pencil and paper.

**3.1 Comparison of Finite Automata**

Given two difffferent FA, determine if they are equivalent

and if not, then show that they do not accept the same lan

guage. A student can either be given the two FA in fifiles, or

can build them with JFLAP. The student must determine

a good set of test data and then run simulations on the in

put strings. The multiple run window allows for the testing

of multiple inputs simultaneously. Alternatively, they can

minimize the two FA and compare their results. Finally,

JFLAP’s *Compare Equivalence* will announce if the two FA

are equivalent. If the two FA are not equivalent, a student

needs to determine one string that is accepted in one FA

and not in the other.

**3.2 Comparison of Regular Expressions**

Given two regular expressions, determine if they are equiv

alent or not. In JFLAP one cannot test strings for a regular

expression. However, one can convert a regular expression

to an equivalent FA and then run a series of test strings

similar to the comparison of two FA.

**3.3 Working Backwards - DFA to NFA**

This problem shows the understanding of how an NFA is

transformed into a DFA, but works backwards. The stu

dents are given a DFA from JFLAP that was transformed

from an NFA. The DFA has each state labeled with the num

bers of the states from the NFA. The problem is to create

the original NFA. There is an assumption that the original

NFA did not have any *λ*-transitions. Once the original NFA

has been created, students can use the *Compare Equivalence*

option with the DFA to determine if they have created the

correct NFA.

**3.4 Creating Automata Based on Properties**

JFLAP can be used to construct examples that illustrate

the properties of languages. For example, given two au

tomata, build an automaton that represents the union of

the two. The two automata may already be constructed.

Using the *Combine Automata* option, both automaton are

placed in the same window, one of them losing its start state

status as only one state can retain this status. The user can

then connect and modify them. In this case, a new start

state is created and *λ*-productions are added from the new

start state to each of the previous start states. The user can

then test the new automaton on multiple inputs.

A more complicated example is to consider the property

called SwapFirstLast(L), which takes the fifirst letter of each

string in L and swaps it with the last letter of that string.

Students are to show that if L is regular, than SwapFirst

Last(L) is regular. Students would approach this problem

in two ways. First, using JFLAP they would construct a

simple DFA M with language L, and then convert it to the

DFA M2 for the language SwapFirstLast(L). Second, with

out JFLAP they would formally prove SwapFirstLast(L) is

regular.

**3.5 Determining Distinguishable States**

One of the transformations in JFLAP is converting a DFA

to a minimal state DFA. The algorithm in JFLAP assumes

initially that all the fifinal states are indistinguishable and all

the nonfifinal states are indistinguishable, grouping indistin

guishable states into two sets, one for fifinal states and one for

non-fifinal states. The algorithm then attempts to determine

if some of the states in a set are distinguishable, thus split

ting a set into two or more sets. If a user suspects that two

states in the same set are distinguishable, they can modify

the DFA to make each of these states a start state (at dif

ferent times) and determine a string that is accepted by one

of the modifified DFAs and not the other. If there is such a

string, then these states are distinguishable and need to be

placed into separate sets.

**3.6 Exploring with Nondeterminism**

Most students are used to thinking sequentially. When

given a problem that can only be solved nondeterministi

cally, many students struggle. The problem of determining

if a string is a palindrome can be solved by a nondeter

ministic PDA (NPDA), and not by a deterministic PDA

(DPDA). Students taking the sequential approach to this

problem want to fifind the middle and then determine if the

right and left half match up. But this approach does not

work with a DPDA. With JFLAP, students can build an

NPDA for this problem and then observe how the nondeter

minism works. When running the simulation with JFLAP,

each current confifiguration is shown. For a valid input string,

one of those confifigurations reaches the middle of the string

and the simulation begins matching the left and right halves,

continuing to acceptance.

**3.7 Exponential Growth in Grammars**

JFLAP can show the exponential growth in grammars,

and the result of transforming the grammar. For example,students are given the grammar on the left that contains a

*λ*-production and asked to transform it into an equivalent

grammar with no *lambda*-productions or unit-productions,

the resulting CFG shown on the right. They are then asked

to compare brute-force parsing of the two grammars. For

input *aaababaabbb*, the grammar on the left takes a long time

to accept, generating 13286 nodes in the derivation tree. The

grammar on the right accepts quickly after generating 335

nodes in the derivation tree.

S *→* aB S *→* aB

S *→* Ba S *→* Ba

B *→* aBb S *→* a

B *→* BB B *→* aBb

B *→* bBa B *→* BB

B *→ λ*

B *→* bBa

B *→* ab

B *→* ba

Similar examples can be shown with unrestricted gram

mars. Those grammars with more items on the left side of

a production will proceed more quickly in parsing.

**3.8 Determining the Language of a Grammar**

JFLAP can be used in determining the language of a

given CFG. In one approach, the CFG can be tested on

multiple inputs. In another approach, the user can divide

the problem into smaller components. For each variable,

enter its productions to determine the capability of that

variable (replacing other variables with temporary termi

nals). For example, consider the grammar below with seven

productions. The user can enter all the B productions in

a new grammar window, with a small *s* for *S* (otherwise

derivations are not possible). The user then derives strings

*{b, ab, bs, aabs, absa, . . .}* and determines that *B*

*∗*

*⇒*

*a∗b*(*λ* +

*S*)*a∗*. Entering only S productions with a special termi

nal to represent the variable B, the user can determine that

*S*

*∗*

*⇒*

*a∗bBa∗*. Putting them together, *S*

*∗*

*⇒*

*a∗ba∗b*(*λ* + *S*)*a∗*

or (*a∗ba∗b*)*∗a∗*. The user can then test the language by de

veloping a set of test strings.

S *→* aS S *→* Sa S *→* bB B *→* aB

B *→* Ba B *→* bS B *→* b

**3.9 In Depth Study of FOLLOW sets**

One of the early steps in LL or SLR parsing is to compute

the FOLLOW set for each variable in the grammar. The

FOLLOW set of a variable is the set of all terminals that

can follow this variable in some derivation. Students are

given an algorithm for computing the FOLLOW sets, and

many can follow the algorithm, but it is not clear that they

really understand the meaning of the FOLLOW sets. For

this problem, students are given a grammar and are asked

to compute the FIRST and FOLLOW sets for the variables

in the grammar. Then they are asked to show the sentential

form in a derivation for each symbol in a FOLLOW set that

shows that terminal immediately following the correspond

ing variable. They can solve this problem partially using

JFLAP by parsing strings with the brute-force parser. A

derivation is shown for each string and one can observe the

sentential forms in the derivation. Several strings may be

needed and a few sentential forms may not be found due to

the order JFLAP replaces productions when given a choice.

**3.10 Parsing Algorithms: Two Approaches**

We show how JFLAP can be used to extensively study

SLR parsing from two approaches. A similar approach ap

plies to LL parsing. Given an SLR(1) grammar, the fifirst

approach is to convert the grammar to an NPDA using the

SLR parsing method. The resulting NPDA has three states

and is likely to be nondeterministic. Students then run the

NPDA on several inputs, making the run deterministic by

choosing the lookahead each time and freezing confifigura

tions that are not chosen. Next students view the parsing

from a second approach. They build an SLR(1) parse table

for the grammar and step through the parsing of the same

inputs with the same lookaheads.

**3.11 Parsing grammars that are not SLR(1)**

With JFLAP one can build an SLR(1) parse table even

if there is a conflflict in the table. For each entry that has

a conflflict, the user chooses one of them. Then the user

can proceed and parse strings using the parse table. Not

all strings in the grammar can be parsed using the choices

chosen. Here is a problem given to students to test their

understanding of the parse table. Given a CFG that is not

SLR(1) and a given input string, fifind the correct choices for

conflflicts in the parse table so the string can be parsed.

**3.12 Running a Universal Turing machine**

With JFLAP’s 3-tape Turing machine, we have constructed

a Universal Turing machine that has 34 states. Using the

Universal Turing machine, a student can encode a simple

Turing machine with a few transitions, each encoded tran

sition will be a string of 0’s and 1’s of about length 15. A

student can then enter an input consisting of the encoded

machine followed by an encoded input string and observe

the simulation.

**3.13 Comparison of one-tape and two-tape TM**

Students are given the language *anbncn* and asked to build

a one-tape TM in JFLAP for this language, and then to

build a two-tape TM in JFLAP for this language. They

then compare the running of several input strings of difffferent

lengths on each TM. In this example, an effiffifficient one-tape

TM will run in *O*(*n*2) time and an effiffifficient two-tape TM

will run in *O*(*n*) time.

**4. NEW FEATURE: BUILDING BLOCKS**

A new feature of JFLAP is Turing machine building blocks.

A building block is a Turing machine with a specifific pur

pose that can be used as a component in building Turing

machines. One can build a complex Turing machine more

easily using building blocks than states.

**4.1 Creation of Building Blocks**

To create a building block, create a Turing machine using

states and transitions, and save it in a fifile. The Turing ma

chine can then be read in as a building block by selecting

the *Building Block Creator*. The building block appears as

a box and can be connected with transitions. We provide

special transitions for hooking up building blocks more eas

ily. Building blocks can also be formed using a combination

of states and building blocks.

With building blocks one wants to start with a simple

foundation. We list simple Turing machines that can form**Figure 1: Building Block for Rnot b**

a library with which to build more complicated Turing ma

chines. These building blocks can be provided for students

to use or they can build some or all of them.

R

Move right once

R a

move right once, keep

moving right until an *a*

Rnot a

move right once, keep

moving right until not an *a*

a

write an a (don’t move)

start starting block

halt halting block

Each of these represent a simple Turing machine. There

are analogous machines for moving left L, L a and Lnot a

and analogous machines for other symbols of the alphabet.

Building blocks can be connected using standard Turing

machine transitions. In JFLAP the standard TM transition

*a*; *b, R* means to read the symbol *a*, write the symbol *b* and

move right. We have created special symbols for transitions

and a new type of transition. The symbol *∼* means to ignore

a read or write. The transition *∼*;*∼, R* means to ignore the

symbol to read, ignore the symbol to write and move right.

The symbol !*x* used in the read position means to match any

terminal that is not the terminal *x*. For example, Figure 1 is

a Turing machine for moving right once and then continuing

to move right until there is a symbol that is not a *b*. We

have named this Turing machine Rnot b and will use it as

a building block.

One may want to connect two building blocks in one of two

ways. First, one may want to connect them so they execute

in sequence, connecting them with *∼*;*∼, S* (S means stay

put). Second, one may want to move to a second building

block depending on the current symbol after processing the

fifirst building block. For example *a*; *∼, S* means if *a* is the

current symbol on the tape then enter this building block

and do not move on the tape head. To further simplify the

connection between two building blocks, we have created

a *Block Transition Creator* that only shows the read and

assumes the write is *∼* and the move is *S*. To reduce the

duplication of code that is similar except for one symbol, we

allow the notation *a*1*, a*2*,...an}v*. This means if one of the

*ai* is read, any occurence of *v* later is replaced by *ai*.

Figure 2 shows a Turing machine built solely with build

ing blocks to represent the transducer *f*(*w*) = *w0* such that

*w0* has all the *a*’s from *w* listed fifirst, followed by all the *b*’s

in *w*. For example, *f*(*babba*) = *aabbb*. The Turing machine

starts in a start building block, which represents a simple

Turing machine of one state that is a start state and a fifinal

state. It then repeatedly moves right fifinding the fifirst *a* past

a *b*, replaces it with a *b* and then replaces the leftmost *b*

with an *a*. When all the *a*’s are to the left of all the *b*’s, the

tape head moves to the leftmost symbol and enters the halt

building block. In this Turing machine, all the transitions

were created with the *Block Transition Creator*. For exam

ple, the *start* block has an *a* transition to the *Rnot a* block.

This *a* transition means “if there is an *a* on the tape head,

do not write on the tape and do not move the tape head,

but go to the *Rnot a* block for the next instruction.”

Building block machines can be quite complicated. Tur

ing machines built with building blocks can be named and

saved in a fifile and used as a building block. Once a building

block is read in using the Building Block Creator, a copy

of its defifinition is stored in the new Turing machine. If the

same building block is read in a second time, then it’s old

defifinition in the fifile is used. Once a building block is part of

a Turing machine, it can be modifified. When the *Attribute*

*Editor* is selected, one can click on a block and then select

the option *Edit Block*. The Turing machine for that block

appears in a new tab and can be modifified. Be cautious: if

there are multiple uses of this block in a TM, modifying one

creates a new defifinition only for that block. It is best to

build and test a block before using it in Turing machines so

that it does not need to be modifified later.

The default name of a building block is the name of the

fifile when the building block is read in. This name can be

changed with the *Set Name* option.

**4.2 Simulation with Building Blocks**

There are fifive options for the simulation of input strings

with Turing machines containing building blocks. One op

tion, *Step*, provides a trace through the Turing machine one

state at a time. When the trace enters a building block, the

building block is highlighted as long as the trace is in a state

within the building block. Selecting the *Focus* option will

automatically change the view to display the transition di

agram for the current building block, with the current state

highlighted. Selecting the *Defocus* option changes the dis

play back to the original Turing machine with its building

block highlighted.

A second option, *Step by Building Block* displays the tran

sition diagram of the original Turing machine and each block

is considered one step in the simulation. Thus, the Turing

machine moves quickly through a trace. If a block represents

“Move right until a blank is seen,” then *in one step* the tape

head moves to the right to a blank. There are three fast run

options. The *Fast Run* takes one input string and gives the

result of acceptance or not without a trace. *Multiple Run*

and *Multiple Run (Transducer)* return the result of several

strings without providing a trace.

**4.3 Other New Features**

There are other new features of JFLAP. One is the abil

ity to change the name of a state for all types of automata

in JFLAP. The default names for states for all types of au

tomata are *qX* where *X* is the number of the state starting

with 0 . Figure 1 shows the default naming of states *q*0, *q*1

and *q*2. Figure 3 shows a Turing machine in which the four

states have been renamed to *start*, *1*, *2*, and *3*. This Turing

machine is a transducer for *f*(*w*) = *w0* in which *w* must start

with an *a* and have at least one *b*. The output is a string

of *b*’s equal in length to the fifirst group of *a*’s. Another new

feature is that Turing machines that are transducers can be

run with multiple inputs. Figure 4 shows the simulation of

several input strings and the output of those strings. The

fourth input string is invalid as it does not contain a *b*.**Figure 2: Turing machine to put** *a***’s fifirst**

**Figure 3: Change fifirst group of** *a***’s to** *b***’s**

**Figure 4: Multiple input for TM transducer**

**5. JFLAP’S USE AROUND THE WORLD**

Since January 2003, JFLAP has been downloaded over

25,000 times in over 120 countries. The type of user was 29%

undergraduate student, 18% graduate student, and 16% fac

ulty. JFLAP was required use by 33% and not required by

36%. The type of use of JFLAP was 48% as a resource, 25%

for homework, 15% as lecture and 12% as lab. The reason

for using JFLAP was 46% taking a course, 14% teaching a

course, and 6% research. These statistics do not add up to

100% as some users elected not to respond.

**6. CONCLUSIONS AND FUTURE WORK**

We are continuing to develop JFLAP with additional al

gorithms and ways to use JFLAP in the FLA course. We

recently started a two-year study to evaluate JFLAP’s ef

fectiveness as a learning tool. A dozen universities are using

JFLAP and participating in the study. A two-day JFLAP

workshop was held in June 2005 and we received feedback

on the use of JFLAP and now have many ideas for improve

ments that we plan to implement.

**7. REFERENCES**

[1] J. Barwise and J. Etchemendy. *Turing’s World 3.0 for*

*the Macintosh*. CSLI, Cambridge University Press,

1993.

[2] R. Cavalcante, T. Finley, and S. H. Rodger. A visual

and interactive automata theory course with jflflap 4.0.

In *Thirty-fififth SIGCSE Technical Symposium on*

*Computer Science Education*, pages 140–144. SIGCSE,

March 2004.

[3] J. Cogliati, F. Goosey, M. Grinder, B. Pascoe, R. Ross,

and C. Williams. Realizing the promise of visualization

in the theory of computing. *JERIC*, to appear, 2006.

[4] M. T. Grinder. A preliminary empirical evaluation of

the effffectiveness of a fifinite state automaton animator.

In *Thirty-fourth SIGCSE Technical Symposium on*

*Computer Science Education*, pages 157–161. SIGCSE,

February 2003.

[5] S. H. Rodger. Jflflap web site, 2005. www.jflflap.org.

[6] S. H. Rodger and T. W. Finley. *JFLAP - An*

*Interactive Formal Languages and Automata Package*.

Jones and Bartlett, Sudbury, MA, 2006.

[7] A. Stoughton. Experimenting with formal languages. In

*Thirty-sixth SIGCSE Technical Symposium on*

*Computer Science Education*, page 566. SIGCSE,

February 2005.

[8] R. Taylor. *Models of Computation and Formal*

*Languages*. Oxford University Press, New York, 1998.

[9] L. F. M. Vieira, M. A. M. Vieira, and N. J. Vieira.

Language emulator, a helpful toolkit in the learning

process of computer theory. In *Thirty-fififth SIGCSE*

*Technical Symposium on Computer Science Education*,

pages 135–139. SIGCSE, March 2004.