Data type Modeling with DFA and NFA as a Lexical Analysis Generator

Nisreen L. Abdulnabi[1], Hawar B. Ahmad[2]
[1]Department of Economics and Administration, Duhok University, Duhok, Kurdistan Region – Iraq
[2]Department of Computer Science, Nawroz University, Duhok, Kurdistan Region – Iraq

ABSTRACT
Lexical analysis helps the interactivity and visualization for active learning that can improve difficult concepts in automata. This study gives an implementation of two frequently used model, NFA for combination of Real and Integer data type and DFA for Double Data Type in Java this chosen model will be implemented using JFLAP. The model will also be tested using JFLAP that will accept at least FIVE (5) inputs and rejected FIVE (5) inputs. These two models are some of the different lexical analyzer generators that have been implemented for different purposes in finite automata.
Keywords: (DFA), (NFA), JFLAP, Lexical Analysis.

1. Introduction

JFLAP (Java Formal Languages and Automata Package) can be defined as an interactive educational software written in Java for experimenting with topics in the computer science area of formal languages and automata theory, JFLAP gives an opportunity to create and simulate structures, such as experiment with proofs , programming a finite state machine, and, converting a deterministic finite automaton (DFA) and a Non-deterministic finite automaton (NFA) (B. Bhowmik & A. Kumar, 2010).
Real number data type had been used in many programming languages for the data types float, double as well as long which are used for storing decimal numbers with different length. Meanwhile, integer data type is used to store only non-decimal number and it is adopted in every programming language such as Java, C# and C++ (H. Luo, 2012). In this example, I would combine these two data types into one single data type by using non deterministic automaton. Maybe this designed could be adopted in other programming languages so that every type of number could be adopted by just using single data types without having to interchange it. An NFA is similar to DFA in a way that it receives input and move from one transition state to another transition state. However, unlike DFA, it can route to two possible transitions based on similar single input and transition state could be performed without receiving any input (∑) which it is known as *transitions.*

Academic Journal of Nawroz University (AJNU) Volume 8, No 4 (2019).
Regular research paper : Published 31 Dec 2019
Corresponding author's e-mail : hawar_duhoki88@yahoo.com

In this simulation, the NFA can be described based on 5 elements (K, ∑, f, S and Z) (A. Mateescu & A. Salomaa, 1995).

However, double data type in java represented in integer or decimal numbers and all the decimal numbers have a Decimal Point**.** Moreover**,** Decimal point goes between units and tenths. Unit is the first number from the left of decimal point and tenths is the first number from the right of decimal point. In fact, the right and left side of decimal point could be numbers from (0-9) and these numbers can be repeated as the user wants. In addition, double data type in java does not accept any letters or more than one decimal point. Furthermore, our project is represented how the double data type works in java (Z. Gejun, S. Yuqiang, 2009). We apply DFA model to do the decimal numbers procedures which is used in java programs. DFA is a data processing technique that allows for the detection of scaling behaviors in observational time series. Our DFA has a five-tuples, (K, ∑, f, S and Z).

2. RELATEDWORKS

Lexical analyzer is extensively used in a lot of area of research. These are the many of the researched that had been done. It is a very challenging approach in query matching on XML stream as the query data is enormous and requires a lot of processing time. Syntactic Twig-Query Matching (STQM) that used the concept of parser and lexical analyzer is able to process the queries on XML and returning the results immediately and continuously (C.-P. Chou, K.-F. Jea and H.-H. Liao, 2011, p. 993–1007) . Mongolian language is adhesive and contains huge amount of

doi : 10.25007/ajnu.v8n4a488

dictionary. In order to identify the words in speech recognition, a lexical analyzer is required. A Mongolian lexical analyzer with the usages of dictionaries and NFA methods is proposed to improve the speed of analyzing the language (S. Loglo, Sarula and HuaShabao, 2010). GLAP model had been proposed to reduce the analysis of time complexity and the design of lexical analyzer (B. Bhowmik & A. Kumar, 2010). In this model, it focuses on a very restricted sub-set of the entire dictionary in least cost (B. Bhowmik & A. Kumar, 2010). Lexical analyzer translates lexemes into token via Lex which communicates with parser for serving token requests. After that, it removes the comments and skips over white spaces. It also will monitor the current line number so that the parser can identify incoming error (G. Dodig Crnkovic and M. Burgin,2012).(HE Yan-xiang, WU Chun-xiang,, 2010) in this study describes learning software that visualizes the different transformations; on the other hand, it used a fixed instance and consequently it amounted to a "canned demo" only. JFLAP (Alfred V.Aho, Ravi Sethi, 2007) is interactive learning software meant to focus on automata theory. It visualizes the main algorithms but only partly automates the construction and instead guides the students during the algorithms, warning them about any errors. The GaniFA applet (Torben Ægidius Mogensen, 2009) focused on the compiler construction rather than aiming at automata theory. The HaLeX library (William M. Waite, Assad Jarrahian, 2006) provides some functions to represent , manipulate REs and FAs and Haskell data types . on the other hand, its focus is on a particular formalization of the algorithms, instead of visualization, and it provides a built-in dot graph and output that is not appropriate for interactive visualization. jFAST (T. M. White and T. P. Way, 2006)did not support lexical specifications using REs and did not visualize any of the transformations between the different representations. it only allowed the simulation of different types of finite state machines (including DFAs and NFAs) and interactive construction.

3. Implementation of DFA /NFA in JFLAP

3.1 DFA for Double Data Type in Java by JFLAP

The DFA/NFA is implemented using JFLAP. The implementation is done step by step as follows:

A. DFA Definition

DFA is a data processing technique that allows for the detection of scaling behaviors in observational time series. Figure 1 shows DFA Models in JFLAP.

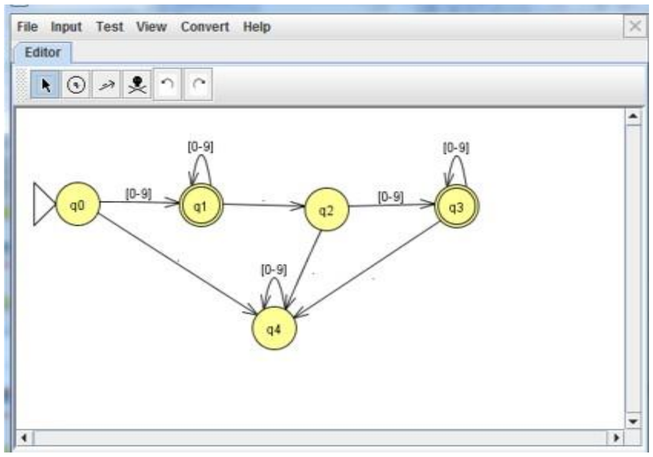


Figure 1- DFA Models in JFLAP

Our DFA has a five-tuples:

M = (Q, ∑, □□, q0,q1, q3)

Where

Q is a finite set of states,

∑ is a finite set of input symbols; q0 is a start state,

q1, q3 is a set of final state,

□□ Is the transition function, which is mapped

Q = {q0,q1,q2,q3,q4 },

∑= {.,0,1,2,3,4,5,6,7,8,9}

*q*0 = start, F = {{q1}, {q4} }

B. Model Details

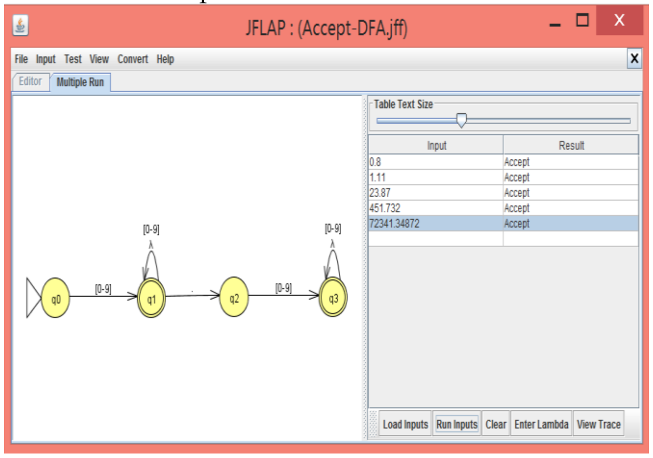
In this part we are going to explain the procedures of this model:

i. Accepted (DFA) model

We have taken five scenarios which include all situations of double data type that accepted by java programming. Figure 2 shows the cases that have been accepted. In general, the cases which we test are achieving the condition of the DFA Model which is:

(a) It has one decimal point.

There are numbers of integer value from (0-9) in right and left decimal point.


JFLAP : (Accept-DFA.jff)

Input	Result
0.8	Accept
1.11	Accept
23.87	Accept
451.732	Accept
72341.34872	Accept

Figure 2- The Cases That Accepted by DFA Modal

The explanations of all cases that accepted:

1. Case 1 the input is (1.8)

The system is in *q*0 state initially. Then it will detect the first integer which is 1 after that go to q1 in (dot) will go to state q2 although 8 is accepted because it from the range (0-9) then it goes to state q3 which is the final state.

2. Case 2 the input is (1.11)

At first *q*0 is state initially. Then it will detect the first integer which is 1 after that go to state q1, and when (dot) is coming the transition goes to state q2. By 1 and 5 goes to state q3, this is the final state.

3. Case 3 the input is (23.87)

The system is in *q*0 state initially. Then it will detect the first integer which is 2 after that go to state q1, and when 3 is coming it remains in state q1 in (dot) will go to state q2 . By 8 the transition goes to state q3, and then stay in state q3 due to 7 is accepted because it from the range (0-9) which q3 is the final state.

4. Case 4 the input is (451.732)

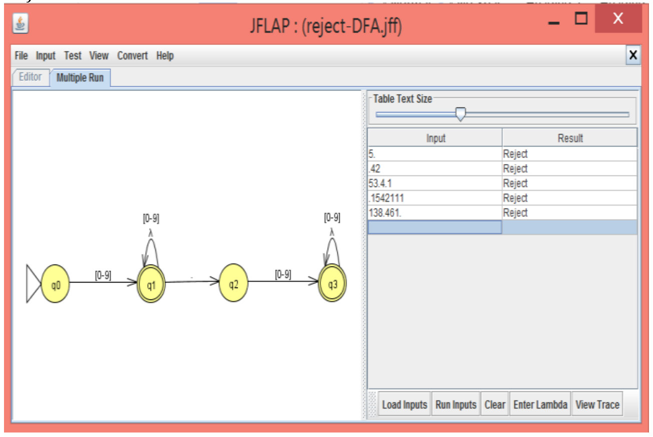
First of all, *q*0 state initially. The system will detect the first integer which is 4 after that go to state q1, and when 5 and 1 is coming it remains in state q1 because this state has loop, in (dot) will go to state q2. By 7 the transition goes to state q3, and then stay in state q3 due to 3, 2 is accepted because it from the range (0-9), which q3 is the final state.

5. Case 5 the input is (72341.34872)

Firstly, q0 state initially. The system will receive the first integer which is 7 after that go to state q1, and when 2, 3, 4, 1 are coming it remains in state q1 because this state has loop, in (dot) will go to state q2. By 3 the transition goes to state q3, and then stay in state q3 due to 4,8,7,2 are accepted because it from the range (0-9) , which q3 is the final state.

ii. **Rejected (DFA) model**

We have taken five scenarios which include some cases of double data type that not accepted by java programming. Figure 3 shows the cases that have been rejected.


JFLAP : (reject-DFA.jff)

Input	Result
5.	Reject
.42	Reject
53.4.1	Reject
.1542111	Reject
138.461.	Reject

Figure 3- The Cases That Rejected by DFA Moda

In general , from figure 3 we can notice that the system reject some cases due to there are more than one dot or when start from dot that is because it does not achieve the condition of decimal numbers in the DFA Model.

The explanations of all cases that has been rejected from DFA Model:

1. Case 1 the input is (5.)

The system is in *q*0 state initially. Then it will detect the first integer which is 5 after that go to q1 in (dot) will go to state q2 due to state q2 is not final state so the systems reject that.

2. Case 2 the input is (.42)

The system is in *q*0 state initially. Then it will not detect the first input which is dot because there is not transition from q0 that can be detected; therefore it is rejected.

3. Case 3 the input is (53.4.1)

At first *q*0 is state initially. Then it will detect the first integer which is 5 after that go to state q1, and when 3 is coming it remains in state q1 because this state has loop, in (dot) will go to state q2 and when 4 is come, the transition goes to state q3. Finally (dot) is coming the situation is stopped.

4. Case 4 the input is (138.461.)

First of all, *q0* state initially. The system will detect the first integer which is 1 after that go to state q1, and when 3 and 8 is coming it remains in state q1 because this state has loop, in (dot) will go to state q2. By 4, 6, 1 the transition goes to state q3, due to there is not transition from q3 so dot is not inter this model.

5. Case 5 the input is (.1542111)

Firstly, q0 state initially. Then it will not detect the first input which is dot because there is not transition from q0 that can be detected; therefore it is rejected.

3.2 NFA for combination of Real and Integer data type by JFLAP

a. In this simulation, the NFA can be describing based on 5 elements (K, ∑, f, S and Z) such as Figure 4:

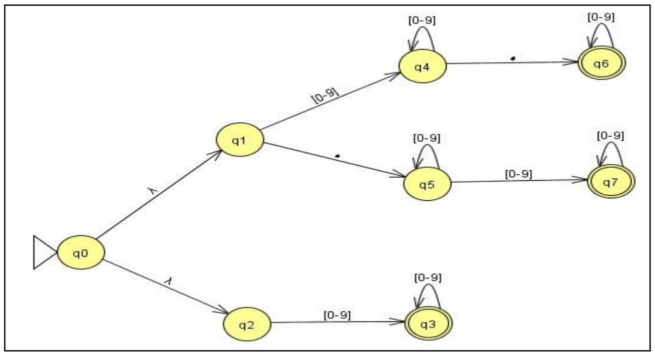


Figure 4: NFA for the combination of real and integer number

K : Finite symbol sets (*Q*)

o The NFA model consists of 8 states

o o *Q = {q0, q1, q2, q3, q4, q5, q6, q7 }*

∑: Alphabets

o ∑= {[0-9], . }

F: Transition functions (Q

Q × (S ∪ {Z}) → subsets of *Q* is a transition function

Example : Q (q1, 5) → q3 As Figure 5 shows.

S: initial state (q0)

The initial state is q0

Z: Accepting state or final state Z= {q3, q6, q7}

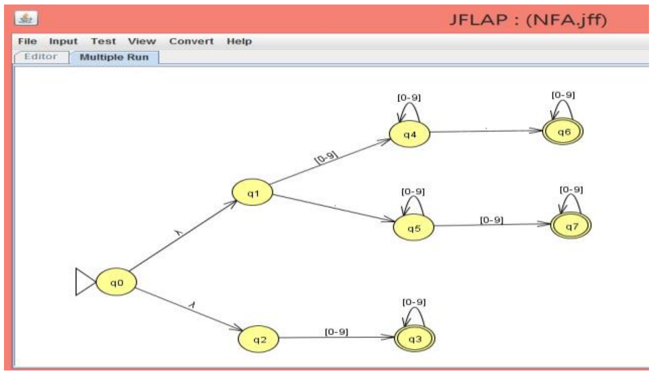


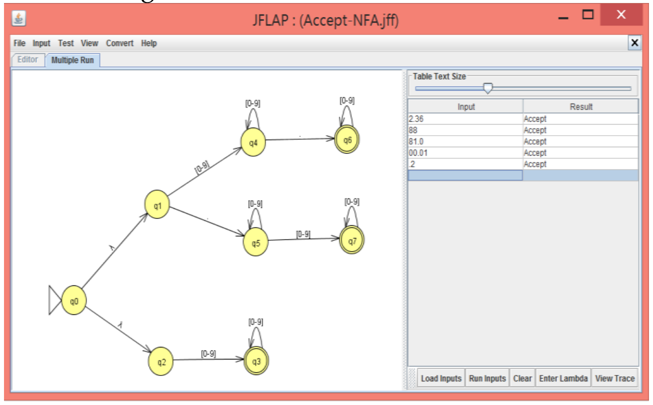
Figure 5- NFA Models in JFLAP

B. Model Details

This section would explain the concept of the NFA and what are the accepted and rejected values for the NFA. Generally, the regular expression for this model is [0-9][0-9]* | ([0-9][0-9]*.[0-9]) | (.[0-9][0-9]*) which the model would accept any input that satisfy this regular expression and reject any input that violate this regular expression. The following are the test cases for accepting test and rejecting test.

1) Accepting test

In this test, there are 5 inputs which are tested with positive results. The 5 inputs values are different style from each other in order to test the accepting states of the NFA. Figure 6 are the test values:



Input	Result
2.36	Accept
88	Accept
81.0	Accept
00.01	Accept
.2	Accept

Figure 6: The cases that accepted in JFLAP

The explanations of all cases that accepted:

1) Case 1 the input is (2.36)

Before input the value of 2.36, the value is separated into few input values (tokens). The tokens are "2", ".", "3" and "6".

- Token 1 = 2
  - o *(q0, 0) → {q3 or q4}*
- Token 2 = .
  - o *(q4, .) → {q6}*
  - o *(q3, .) → rejected*
- Token 3 = 3
  - o *(q6, 3) → {q6}*

- Token 4 = 6
    - *(q6,.) → {q6}*

Since token 4 is the last input and q6 is the accepting state, the input value 2.36 is accepted.

2) Case 2 the input is (84)

Before input the value of 84, the value is separated into few input values (tokens). The tokens are "8" and "4".

- Token 1 = 8
    - *(q0, 8) → {q3 or q4}*
- Token 2 = 4
    - *(q3, 4) → {q3}*
    - *(q4, 4) → {q4}*

Since the token 2 is the last input and q4 is the accepting state, the input value 84 is accepted. Meanwhile, the q3 is not accepting state. So, the transition ends there without accepting it.

3) Case 3 the input is (81.0)

Before input the value of 81.0, the value is separated into few input values (tokens). The tokens are "8", "1", "." and "0".

- Token 1 = 8
    - *(q0, 8) → {q3 or q4}*
- Token 2 = 1
    - *(q3, 1) → {q3}*
    - *(q4, 1) → {q4}*
- Token 3 = .
    - *(q3, .) → {q6}*
    - *(q4, .) → rejected*
- Token 4 = 0
    - *(q6,0) → {q6}*

Since the token 4 is the last input and q6 is an accepting state/ final state, the **input value 81.0 is accepted**.

4) Case 4 the input is (00.01)

Before input the value of 00.01, the value is separated into few input values (tokens). The tokens are "0", "0", ".", "0" and "1".

- Token 1 = 0
    - *(q0, 0) → {q3 or q4}*
- Token 2 = 0
    - *(q3, 0) → {q3}*
    - *(q4, 0) → {q4}*
- Token 3 = .
    - *(q4, .) → {q6}*
    - *(q3,) → rejected*
- Token 4 = 0
    - *(q6, 0) → {q6}*
- Token 5 = 1
    - *(q6, 1) → {q6}*

Since token 5 is the last token and q6 is the accepting state, the input value 00.01 is accepted.

5) Case 5 the input is (.2)

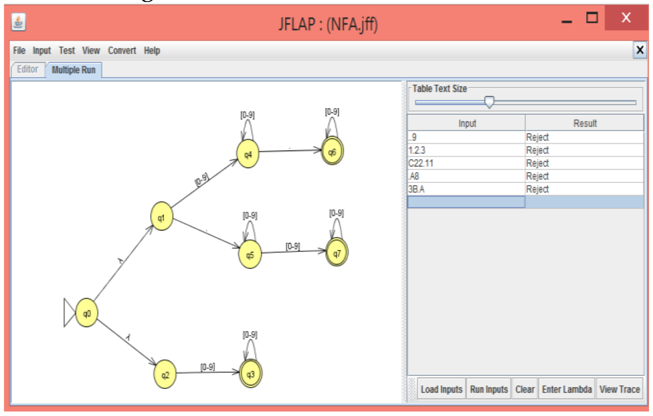
Before input the value of .2, the value is separated into few input values (tokens). The tokens are "." and "2".

- Token 1 = .
    - *(q0, .) → {q5}*
- Token 2 = 2
    - *(q0, 9) → {q5 or q7}*

Since token 2 is the last token and q7 is the accepting state, the input value .2 is accepted.

2) Rejecting test in NFA

In this test, there are 5 inputs which are tested with negative results. The 5 inputs values are different style from each other in order to test the accepting states of the NFA. Figure 7 are the test values:



Input	Result
.9	Reject
1.2.3	Reject
C22.11	Reject
.A8	Reject
3B.A	Reject

Figure 7: The cases that rejected in JFLAP

The explanations of all cases that rejected:

1) Case 1 the input is (..9)

Before input the value of...9, the value is separated into few input values (tokens). The tokens are ".", ".", "9".

a. Token 1 =.
    - *(q0, .) → {q5}*

b. Token 2 =.
    i. *(q5,) → rejected*

Although token 2 is not the last token but since all the possible transition is rejected, then the input value...12 is rejected.

2) Case 2 the input is (1.2.3)

Before input the value of 1.2.3, the value is separated into few input values (tokens). The tokens are "1", "2", "3".

a. Token 1 = 1
    - *(q0, 0) → {q3 or q4}*

b. Token 2 = .
    - *(q4, .) → {q6}*
    i. *(q3, .) → rejected*

c. Token 3 = 2
   - *(q6, 0) → {q6}*

d. Token 4 = .
   i. *(q6, .) → rejected*

Although token 4 is not the last token but since all the possible transition is rejected, then the input value 1.2.3 is rejected.

3) Case 3 the input is (C22.11)

Before input the value of C22.11, the value is separated into few input values (tokens). The tokens are "C", "2", "2", ".", "1" and "1".

a. Token 1 = C
   i. *(q0, C) → rejected*

The input value for token 1 did not meet any transition value. So the token 1 is already rejected which cause the following token won't be executed. Therefore, the input value C22.11 is rejected.

4) Case 4 the input is (.A8)

Before input the value of .A8, the value is separated into few input values (tokens). The tokens are ".", "A" and "A".

a. Token 1 = .
   - *(q0, .) → {q5}*

b. Token 2 = A
   i. *(q5,A) → rejected*
   ii. *(q7,A) → rejected*

c. Token 3 = 8
   - *(q5,8) → {q5}*
   - *(q5,8) → {q7}*

Since both possible transitions are rejected, then the input value .A8 is rejected.

5) Case 5 the input is (3B.A)

Before input the value of 3B.A, the value is separated into few input values (tokens). The tokens are "3", "B", "A".

a. Token 1 = 3
   - *(q0, 0) → {q3 or q4}*

b. Token 2 = B
   i. *(q4,B) → rejected*

Although token 2 is not the last token but since all the possible transition is rejected, then the input value 3B.A is rejected.

4. CONCLUSION

JFLAP is a usable and valuable software tool that assists in teaching and learning about finite state machines, a concept some students find particularly challenging and initially impenetrable. Teachers and students found the software to be easy to learn and use, allowing the focus to be on the concepts rather than the software classrooms in the coming year. We have shown how visualization and interaction can be integrated into a DFA and NFA using JFLAP as a tool. Such tools provide a visual picture, making it easier to see relationships between objects, and provide interaction, allowing the student to experiment with the picture and receive immediate feedback.

REFERENCES

C.-P. Chou, K.-F. Jea and H.-H. Liao, "A syntactic approach to twig-query matching on XML streams," The Journal of Systems and Software, vol. 84, p. 993–1007, 2011.

H. Luo, "Research of Using Finite Automaton in the Modeling of Lexical Analyzer," in International Conference on Information Management, Innovation Management and Industrial Engineering, 2012.

S. Loglo, Sarula and HuaShabao, "Research on Mongolian Lexical Analyzer Based on NF A," in IEEE, 2010.

Z. Gejun, S. Yuqiang, Y. Ruimin and G. Yuwan, "A Simplification Algorithm of Regular Grammar Production," in International Conference on Information Science and Engineering, 2009.

Bhowmik, A. Kumar, A. K. Jha and R. Kumar Agrawal, "A New Approach of Complier Design in Context of Lexical Analyzer and Parser Generation for NextGen Languages," International Journal of Computer Applications, vol. 6, no. 11, pp. 21-25, 2010.

Mateescu, A. Salomaa, K. Salomaa and S. Yu, "Lexical Analysis with a Simple Finite-Fuzzy-Automaton Model," Journal of Universal Computer Science, vol. 1, no. 5, pp. 292-311, 1995.

William M. Waite, Assad Jarrahian, Michele H. Jackson, Amer Diwan, 2006. Design and Implementation of a Modern Compiler Course, ACM 1595930558/06/0006.

Braune, S. Diehl, A. Kerren, and R. Wilhelm.Animation of the generation and computation of finite automata for learning software. In Proc. 4th Intl. Workshop Implementing Automata, LNCS 2214, pp. 39–47. Springer, 2001.

R. Cavalcante, T. Finley, and S. H. Rodger. A visual and interactive automata theory course with JFLAP 4.0. In Proc. SIGCSE'04, pp. 140–144. ACM Press, 2004.

S. Diehl, A. Kerren, and T. Weller. Visual exploration of generation algorithms for finite automata on the web. In Proc. 5th Intl. Conf. Implementation and Application of Automata, LNCS 2088, pp. 327–328. Springer, 2000.

J. Saraiva. HaLeX: A Haskell Library to Model, Manipulate and Animate Regular Languages. In Proc. ACM Workshop on Functional and Declarative Programming in Education, pp. 133–140. University of Kiel Technical Report 0210, 2002.

T. M. White and T. P. Way. jFAST: A Java finite automata simulator. In Proc. SIGCSE'06, pp. 384–388. ACM Press, 2006.