Improving the Flexibility of JFLAP
String Symbols, Explicit Formal Definitions, and Code Reconstruction*

Julian Z. Genkins
Computer Science Department
Duke University
Durham, NC 27708
jzg@duke.edu

Abstract: Java Formal Languages and Automata Package (JFLAP) is an educational tool designed to assist in the teaching of formal languages and automata (FLA) theory. The package consists of modules spanning different language classes including regular, context-free, and recursively enumerable languages. JFLAP is used internationally in a wide variety of courses at the university level. The current version of JFLAP, v7.0, obscures the formal definitions which underlie automata and grammars, limiting the contents of associated symbol sets by enforcing restrictive naming conventions. Included in these conventions is the mandate that every character represents an individual symbol. This means that JFLAP can only be utilized to represent simple, symbolic languages. In an effort to improve the flexibility of the JFLAP software, the program now comes equipped with a formal definition package which allows for custom creation and explicit representation of formal definitions. Symbols can now be defined by multiple characters, a string, which allows the creation of more complex, literal languages. To enhance feedback and clarity, a set of concrete, context-specific rules now governs the limitations placed on adjustments made to these alphabets. Finally, as a byproduct of these extensions, the code base of JFLAP has been restructured and unified to better accommodate such global changes. In order to gauge the efficacy of these changes, a small assessment was done at the end of the project. Overall, the results of the survey indicated that the changes are valuable, worthwhile, and the above issues were ameliorated to some degree.

*This work was supported by the National Science Foundation through grant NSF TUES 1044191
1. Introduction

Formal Languages and Automata (FLA) theory is a set of concepts which includes grammars, machines, and other formal language-related principles. (Chakraborty 2011) FLA theory forms the theoretical foundation of computer science (CS), with applications spanning across graph theory, compilers, coding languages, string operations, and more. (Linz) As a result of this evident relevance to CS, a course in FLA theory is commonly a cornerstone of undergraduate or graduate level CS education.

However, FLA theory is a challenge to present clearly as a result of its nature as a collection of abstract or theoretical concepts. The teaching of FLA theory hinges on the construction of models – grammars and automata - which act as mechanisms for the formal definition and analysis of language. These constructs are then transformed into graphical representations or written forms to enable hands-on interaction. Because of their complexity, hand-written presentations of these structures can be tedious to create and frustrating to test, serving as an additional obstacle in FLA theory education.

To ameliorate this issue, educators have sought to apply the principles of software visualization (SV) and algorithm visualization (AV) to the teaching of FLA theory. These efforts have yielded a large collection of FLA theory-oriented SV educational programs. Such tools were in development as early at 1963, and slowly transitioned from FLA scripting languages to static graphical representations to interactive programs with dynamic parts. (Chakraborty 2011) JFLAP is one example of such a tool. It is a comprehensive suite of modules which address many topics canonical to a FLA course and provides an interactive interface for creating, studying, and testing those concepts.

The goal of this work was to study SV, FLA theory, and the JFLAP software in an effort to further enhance the latter’s capability and effectiveness as an education tool. Initial exploration resulted in the hypothesis that JFLAP’s scope and flexibility could be extended to encompass more complex languages through a few small, influential changes. These adjustments primarily targeted previous, implicit JFLAP conventions and included allowing string-based symbols and representing explicit formal definitions, with the goal of producing a more powerful, dynamic piece of software that could better enhance a user’s experience learning FLA theory. Once implemented, the effectiveness of these changes was evaluated in a small-scale survey of students already engaged in an FLA theory course. The results of this assessment corroborate the initial supposition that such additions to the program would be welcome improvements and helpful tools for the future.

Genkins 1
2. FLA Theory

FLA theory outlines a collection of formally defined structures which can be divided into three primary categories – languages, grammars, and automata. A given formal language can be described equivalently using each of these base models. These categories can be further subdivided by the complexity class of the represented language. FLA theory provides precise, formal descriptions of language which are foundational to code parsers, compilers, and other language-based tools.

2.1 Languages

A formal language is a method of precisely defining how individual symbols may be combined into strings such that those strings form the contents of that language. Each language is limited to a finite set of symbols known as an alphabet. A string in the language is defined as an ordered concatenation of these symbols. Languages introduce finite string operations such as concatenation or reversing, infinite combinatorial operations such as the star closure (Kleene star), and the concept of an empty string.

2.2 Grammar

Grammars are one of multiple concrete mechanisms used to represent a language. Colloquially, they provide a finite set of rules and a finite set of symbols which govern the composition and scope of a language. The formal definition of a grammar \( G \) is the 4-tuple \( G = (V,T,S,P) \). \( T \), the set of terminals, provides the symbols which are concatenated to form a string \( w \in L(G) \) (the language of \( G \)). \( P \) is the aforementioned set of production rules which directs how symbols in the grammar can be combined. Variables, \( V \) are intermediate symbols not found in \( L(G) \). They exist only in the sentential (intermediate) forms of a string in a derivation.

The derivation of a string using \( G \) begins with the start variable, \( S \), applies a sequence of production rules, and terminates once the resulting string is composed entirely of terminals. Any string that can be derived through this process is considered to be in \( L(G) \). Two grammars are equivalent if they generate the same language, i.e. if they derive an identical set of unique strings.

There are many different classes of grammars, as delineated by the complexity of the languages they define and/or by specific restrictions placed on the structure of productions. For example, a context-free grammar (CFG) must have only a single variable on the left-hand side of the production but can have any concatenation of terminals and variables on the right-hand side. The class of CFG derives a specific set of languages, known aptly as context-free languages (CFL).
2.3 Automata

Automata are theoretical machines which provide another mechanism for describing the contents of a given language. Each Automaton can read a string input composed of symbols from an input alphabet. The machine will then process the input by moving through states as dictated by a transition function, analogous to the production rules of a grammar. Whether a transition function can be applied to the current state of the automaton depends on the next symbol in the input as well as other machine-specific conditions. Once the machine halts, which can occur for a variety of reasons, it will provide output based on the input. (Linz)

The form of this output varies among the different families of machines. Some are transducers, which output a string, and others are acceptors, which output a binary accept/reject. Another meter for dividing up the population of theoretical machines is by the complexity of the language of the machine, \( L(M) \). This separates automata into such categories as: finite state automata, pushdown automata (PDA), or Turing machines (TM).

2.4 Regular Expressions

One other way to describe a language is via regular expressions (RE). This method applies only to regular languages. REs are defined via the combination of strings of symbols limited by some symbol alphabet, parenthesis (or some other variety of grouping pair), and the following operators:

\[
\begin{align*}
+ & = \text{Union of strings} \\
\cdot & = \text{Concatenation of strings} \\
* & = \text{Star closure (Kleene star)}
\end{align*}
\]

An example of the language \( L \) produced by a regular expression \( r \) on the alphabet \( \Sigma = \{a, b, c\} \) is:

\[
\begin{align*}
r & = ((a \cdot b) + c)^* \\
L(r) & = \{\lambda, ab, c, abc, cc, abab, abcab \ldots\}
\end{align*}
\]

In the definition above, \( \lambda \) represents the empty string. If two REs derive the same language, then they are considered equivalent. JFLAP provides a symbol interface for REs which allows the user to input the expression as a string and then either test input on the RE or transform it into an equivalent automaton (a deterministic finite acceptor). Although REs comprise a small component of FLA theory, they provide an important example of how patterns translate into language and an alternative way to view language derivation.
3. Software Visualization

The ability to transform abstract languages into concrete formal definitions is one of the fundamental reasons for constructing automata or grammar. Therefore representing these structures effectively is essential in designing a piece of software dedicated to FLA theory. The academic theory outlining the study of methodologies used to produce these graphical or tangible constructs is known as Software Visualization (SV).

SV is defined as “the use of the crafts of typography, graphic design, animation, and cinematography with modern human-computer interaction and computer graphics technology to facilitate both the human understanding and effective use of computer software” (Stasko 1997). The theory of SV focuses on the balancing of four major design categories – scope, content, form, and method.

Scope governs the range of programs or algorithms that a given platform can handle. Content describes what data is presented in the code, and how much of that translates into the program itself. Form dictates how this content is presented, including medium, presentation style, and granularity or degree of detail. Finally, method designates how a program is run, whether it is automatic, interactive, or a combination of the two. (Stasko 1997) Seeking an ideal equilibrium between these concepts produces the core questions of Software Visualization, namely:

1. Is the complexity of the software compensated for by its effective explanation of the target material?
2. How can one use insight gained while working on the code to actively improve the design of the software?
3. How can one gauge the effectiveness of a piece of visualization software in an objective fashion?

3.1 Education through Software Visualization

One of the primary applications of SV is to create interactive educational software which employs SV strategies to enhance learning. Educators generally agree on the potential of software visualization to improve the computer science education experience. However, experimental studies designed to prove or analyze the merit of education through SV have not reinforced this belief. (Naps 2002) This conclusion can neither be confirmed nor denied, as there is a dearth of reliable information on measuring the effectiveness of software visualization tools.
Other studies have shown that the effectiveness of SV software is linked to a mandating of user interaction within a given tool. (Naps 2002) Active usage of the software seems to enable students to construct their own understandings of concepts. This observation has been corroborated in other meta-studies of SV subcomponents, such as information visualization (InfoVis) and algorithm visualization (AV). (Stasko 1997, Hundhausen 2001, Stasko 2007) Specific to JFLAP, another study done in 2009 across 14 universities showed that the majority of students who use the program to study FLA theory felt more engaged in the course and their overall learning experience was more enjoyable. (Rodger 2009)

The growing popularity of SV in education is undeniable when one considers such sources as AlgoViz.org, a relatively new database of many popular pieces of educational software focusing on CS. The tools present on AlgoViz span CS education, treating topics from linear structures to computational geometry to FLA theory. (AlgoViz.org)

Overall, any assessment of SV effectiveness relies heavily on conjecture and opinion as there is no standard for what qualifies as effective methods or degrees of user interaction. Instead individual educators often apply their own subjective rubric to determine if the chosen method of interaction is best. The combined experiences of educators who believe they are using SV software effectively can be reduced to a few generally applicable approaches:

1. Provide a medium through which a user can interpret or interact with visualizations
2. Vary the form of visualizations such that the platform caters to a range of different knowledge levels
3. Give access to differing perspectives or representations of the same algorithm or diagram
4. Enable flexible control of execution such that a user may experiment with different steps, speeds, or methods of moving through the same algorithm.
5. Keep the user interested through active testing or application of their knowledge
6. Be forgiving to the user, ensure that tedium is minimized and mistakes are rectifiable
7. Keep the interface uncluttered and simple, giving access to more complicated tools without confusing simpler processes. (Stasko 2002) (SV book p 146)

These considerations provide a general framework for the effective synthesis of SV education tools. They also serve as a standard by which an SV designer or educator can judge if the method of visualization and degree of interaction in their tool are adequate and appropriate. (Yi 2007) (Hundhausen 2001)
4. JFLAP: A Short History

JFLAP, or Java Formal Languages and Automata Package, is a SV educational tool for FLA theory and related concepts. It seeks to explore FLA theory through a unified platform, addressing most FLA topics with an organized, systematic approach toward visual representation of automata and grammars that helps to reduce tedium and prevent human error.

4.1 Motivation

In the past, teaching of FLA theory has relied on using paper and pencil as a means transforming abstract concepts, proofs, and constructions into their concrete visual representations. These methods proved frustrating and tedious for students and teachers alike, as testing of automata or grammars required trial-and-error iteration with no feedback between attempts. Without an interactive, reusable platform, canonical FLA theory courses lacked expedience and yielded poor results.

Efforts to cope with these issues have been made in the past two decades, typically through small-scale programs focusing on specific sub-components of FLA theory. Of these tools, most focus on a particular concept such as finite automata (Stallman 1994) or Turing machines (Barwise 1993), although there are a couple of examples which strive to span multiple FLA theory topics (Taylor 1998). Some programs sought to construct simple programming languages through which one could construct and compile an automaton via a rigidly formatted document. One example of this is Finite Automaton Description Language (Figure 4.1) which allowed for the description of a finite automaton to be written in a textbook-like fashion and then rapidly compiled into the image on the right. (Chakraborty 2011) Alternative approaches, such as a web-based textbook (Ross 2008), seek to explore improved ease of platform as a mechanism for encouraging use, decreasing the hassle required to retrieve and install a piece of software.

Figure 4.1 – Describing Automata:
To the right is a definition of a deterministic finite automaton using the specification of the Finite Automaton Description Language. That script is then interpreted into a graphical representation on the far right. (Chakraborty 2011)
Although each new tool for automaton simulation and grammar exploration are consistently welcomed by the scientific community, this variety in platforms makes it difficult to use SV across a comprehensive FLA curriculum. One must overcome the barriers of familiarity before truly taking advantage of what each separate program has to offer. Thus the twofold goal of JFLAP is:

1. To mitigate the harrowing effects of “pen and paper” methodologies through easily reproducible and testable representations.
2. Increasing the accessibility to software visualization techniques through a unified platform for concepts spanning FLA theory.

4.2 Development History

JFLAP found its origin in work done at Rensselaer Polytechnic Institute in the early 1990s under the guidance of Professor Susan Rodger. First written in C++ and XWindows, it began as a simple tool for constructing NPDAs. Called “FLAP” at the time, the tool grew rapidly. In the subsequent years it incorporated other automata such as finite automata and Turing machines, grammars, and various algorithms including LR parsing, non-deterministic to deterministic finite automata conversions and conversions of grammar to automata.

After Professor Rodger moved to Duke University, work continued on the JFLAP lineage with the addition of LL and LR parse trees, L-systems, a pumping lemma module, and brute force parsing. In the late 90’s and early 2000’s FLAP became JFLAP after being rewritten in Java and then in Swing, the native Java GUI package. This transformation retained much of the same functionality as previously and even included the addition of Moore and Mealy machines. In 2006, Professor Rodger and a student, Thomas Finley, co-authored the first book centered on JFLAP, a workbook guiding students through the program. (Rodger 2006) In the recent past, JFLAP has been improved further, focusing on its value as a teaching tool in the hopes of building an entire FLA curriculum around the program. Batch testing, export of images, and automaton magnification are just a few of these education oriented improvements which make JFLAP much more tractable in the classroom or lecture environment. JFLAP has a website and associated tutorial page (JFLAP.com), containing helpful explanations on how to use certain components of the program as well as many example files which students can download. One can also acquire JFLAP itself from this website.
5. Current Functionality

The current distributed version of JFLAP, version 7.0, is nominally composed of eleven modules as seen from the main menu upon starting the program. The definition and capabilities of each are outlined below.

5.1 Representing Automata

In discussing different FLA concepts, it is essential to have a well-defined, intuitive method for visually representing the abstract structure. For automata we use transition graphs, in which the vertices represent states and labeled edges as transitions. (Linz) The labels on the vertices correspond to the names of the states, while each edge label denotes a given transition function. The representation of this function is dependent on the family of automata being drawn. *(Figure 5.1)*

The start state is designated by an incoming arrow or triangle not originating at any vertex. Accepters have a set of one or more final states, each designated graphically by a second circle drawn inside that state. Beginning at the start state, running an input string on the graph involves traversing from state to state along transitions based on the current location in the input until termination. If that state happens to be a final state, then the string is accepted.

One can create a transition graph of an automaton with JFLAP, including states and transitions with associated symbols as labels, in a process not unlike drawing the machine on paper. The software is much more forgiving though, as it provides mechanisms to undo and redo edits along with providing some feedback to guide the testing process.

*Figure 5.1 – A Transition Graph*

This image is an example of a transition graph from JFLAP. It exemplifies the way that transition diagrams are generally drawn. \( q_0 \) is the start state, and \( q_2 \) is the final state. Specifically, the automaton represented is a finite state accepter.
5.2 Finite Accepters

Formally: \( M = (Q, \Sigma, \delta, q_0, F) \)

\[
\begin{align*}
    Q &= \text{Finite set of states} \\
    \Sigma &= \text{Finite Input alphabet} \\
    \delta: Q \times \Sigma &\to Q = \text{Transition function} \\
    q_0 &\in Q \text{ is the initial state} \\
    F &\subseteq Q \text{ is a set of final states} \\
    **\text{for NFAs}, \delta: Q \times (\Sigma \cup \{\lambda\}) &\to 2^Q
\end{align*}
\]

This module focuses on deterministic and non-deterministic Finite Accepters (DFA and NFA respectively). DFAs accept a string \( w \) by scanning moving from left to right in the string, scanning one symbol at a time. If \( |w| = n \), then a DFA will accept or reject a string in worst cast \( O(n) \) runtime. The set of strings accepted by a DFA is denoted by \( L(M) \) and is guaranteed to be a regular language;

NFAs differ slightly from DFAs due to the fact that the range of \( \delta \) is the powerset \( 2^Q \), \( \lambda \) is allowed as an input symbol, and the set \( \delta(q, a) \) can be empty, meaning that there may not be a transition defined for this configuration. Thus non-determinism is a product of the fact that, in any given state, there may be multiple transition functions with the same associated input symbol and state but different end states.

Through JFLAP, the user may run a variety of different algorithms and tests on the FA. These include stepping through input, minimizing a DFA, converting an NFA to a DFA, converting the DFA to a grammar, adding a trap state, and applying various graph layout algorithms to the states. It is also possible to add notes to the automaton canvas as a way of marking various components or adding simple reminders of functionality.

5.3 Mealy Machine

Formally: \( M = (Q, \Sigma, \Lambda, T, G, q_0) \)

\[
\begin{align*}
    Q &= \text{Finite set of internal states} \\
    \Sigma &= \text{Finite Input alphabet} \\
    \Lambda &= \text{Finite Output alphabet} \\
    T: S \times \Sigma &\to S = \text{Transition function} \\
    G: S \times \Sigma &\to \Lambda = \text{Output function} \\
    q_0 &\in Q = \text{Initial state}
\end{align*}
\]

Mealy machines are finite state transducers which produce output based on the output function \( G \), which maps an output symbol to a given input symbol-state pairing. Some definitions combine the set of transition function and output functions into a single set governed by the hybrid function \( T: S \times \Sigma \to S \times \Lambda \). The Mealy

Genkins 9
machine graph is drawn exactly like that of an FA except that output symbols are written on each transition label along with input symbols, mirroring the alternate definition for $T$ mentioned previously. The JFLAP interface simply allows for a step-by-step and fast run of input which presents the output of a specific, user-defined string input.

### 5.4 Moore Machine

Formally: $M = (Q, \Sigma, \Lambda, T, G, q_0)$

Like Mealy machines, Moore machines are transducers and therefore produce output. Unlike Mealy machines, the output function of a Moore Machine maps states to output symbols. For each step through an input, the output generated depends on the symbol associated with the next state. Figure 5.2 above shows how Moore states are represented in JFLAP v7.0 and will continue to be in future versions.

### 5.5 Nondeterministic Pushdown Automaton

Formally: $M = (Q, \Sigma, \Gamma, \delta, q_0, z, F)$

Pushdown automata, or PDAs, are an example of an automaton which uses memory. Each PDA has a stack, an infinitely large FIFO (first-in-first-out) queue of symbols from the stack alphabet. This stack is analogous to a computer’s memory. Transition functions have both a pop and push action associated with them, thus editing memory as a string is read. The current configuration, called “instantaneous configuration,” of a PDA is composed of a 3-tuple, $(q, w, \beta) \in Q \times \Sigma^* \times \Gamma^*$, $q$ is the current state, $w$ is the remaining unread portion of the input string, and $\beta$ is the contents of the stack.
Like FAs, PDAs can be divided into two categories – non-deterministic (NPDA) and deterministic. For NPDAs, the transition function becomes:

\[ \delta: Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow \{Q \times \Gamma^+, Q \times \Gamma^+, \ldots\} \]

In other words, each \((q, a, b) \in Q \times \Sigma \times \Gamma\) can have a finite set of \(Q \times w, w \in \Gamma^+\) 2-tuples to which they map. A PDA accepts or rejects a string if, upon termination, the instantaneous configuration is in state \(q_0\). An alternate definition for accepting is to accept by empty stack. In this case, if the \(|\beta| = |w| = 0\) in the instantaneous description of the PDA at any point, then the input string is accepted.

The collection of strings accepted by an NPDA comprises the less restricted family of context-free languages (CFL) which, as a result of the PDA memory stack, can contain more complex derivations than regular languages. It follows that the grammar into which an NPDA can be converted is a context-free grammar.

### 5.6 Turing Machine

Formally: \(M = (Q, \Sigma, \Gamma, \delta, q_0, B, F)\)

- \(Q = \text{Finite set of internal states}\)
- \(\Sigma = \text{Finite Input alphabet}\)
- \(\Gamma = \text{Finite Tape alphabet}\)
- \(\delta: Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R\} = \text{Transition function}\)
- \(q_0 \in Q = \text{Initial State}\)
- \(B = \square \in \Gamma = \text{Blank symbol}\)
- \(F \subseteq Q = \text{set of final/halt states}\)

A Turing machine (TM) is an automaton with temporary storage in the form of a tape. The tape is an array of cells that extends infinitely in both directions, with each cell containing a single symbol. The machine uses a read-write head to note which cell is currently being adjusted. The read-write head is capable of reading symbols from the tape, writing symbols to the tape, and moving right or left on the tape. The TM transition function maps a current state and tape symbol to a next-state, tape symbol to write, and a read-write head movement.

Turing machines may act as transducers or accepters. As transducers, the input is written to the tape, the TM is run, and the output is set as the string of symbols to the right of the resting position of the read-write head. As an acceptor, if the TM halts on a final state, meaning there are no transition functions that can be applied to the current configuration, then the input string is considered part of the \(L(M)\).
There are many varieties of Turing machines including TM’s using a stay option, semi-infinite tapes, multi-track and multi-tape TM’s, or even multi-dimensional TM’s. JFLAP only treats multi-tape TMs but also always has a stay command available therefore incorporating this variation as well. Multi-tape TMs have a finite number of tapes with an independent read-write head on each. Multi-tape TMs can enhance the algorithmic runtime of basic TMs but do not add any computing power to the overall mechanism. Any multi-tape TM can be represented by a single-tape Turing machine. In fact, as per Alan Turing’s yet unproven hypothesis, “Any computation that can be carried out by mechanical means can be performed by some Turing machine,” (Linz) specifically referring to a single-tape TM.

JFLAP also allows the construction of Turing machines using blocks. A block is represented by a special state in the TM transition diagram, and corresponds to a previously created TM. The block allows one to design a Turing machine with a special purpose and then use that sub-machine as a block in a larger machine. This process is analogous to the construction of macros, and the implementation of those macros in higher level scripts to enable more complex code with minimal duplication. Thus implementing blocks in a TM promotes good design and enables creation of even more complex machines.

Like the other automata, the JFLAP TM module allows the user to design TMs and run input as either a transducer or an acceptor. A Turing machine can be converted into an unrestricted grammar, the least restricted family of grammars and therefore the most potentially complex.

### 5.7 Grammar

Formally: \( G = (V, T, S, P) \)

- \( V = \text{Finite set of variables} \)
- \( T = \text{Finite set of terminal} \)
- \( S \in V = \text{start variable} \)
- \( P = \text{Finite set of productions} \)
- \( V, T \) are non-empty and disjoint.

As discussed in the introductory section on FLA theory, grammars are an alternative way to formally represent languages without using automata. By applying the production rules of a grammar to its constituent symbol alphabets, one can generate many different families of languages with unique properties.

Production rules are of the form \( a_1 \rightarrow a_2 \) where \( a_1 \in (V \cup T)^+ \), \( a_2 \in (V \cup T)^* \). \( a_1 \) is called the left hand side (LHS), \( a_2 \) is the right hand side (RHS) of the production. Rules come in many distinct varieties based on the...
structure of their LHS and RHS. Specific configurations of these sub-components determine the family of which \( L(G) \) is a part of. One example of this is a right-linear grammar where, for every production rule \( p \in P \), \( p: (V \rightarrow T^* \cdot V) \) Any grammar that holds to this restriction derives a regular language, the same kind of language accepted by an FA.

Grammars are used to parse input strings. If a grammar can derive a string \( w \), then \( w \) can be parsed by that grammar. The derivation for \( w \) is the finite sequence of productions that, when applied, result in the generation of \( w \). This sequence must begin with a start production, a production with only \( S \) on the LHS. There are many different parsing procedures, each with various strengths in the context of different complexities of language. JFLAP contains four unique parsing algorithms, brute force parsing, CYK parsing, LL parsing, and SLR(1) parsing.

Grammars come in many varieties based on restrictions placed on the productions. These restrictions give certain grammars special characteristics that make them easier or harder to parse with. Typically, different kinds of grammars are associated with different automata from which they can be generated via automata→grammar conversions. JFLAP contains content that deals with the following grammars:

<table>
<thead>
<tr>
<th>Grammar</th>
<th>Associated Automaton</th>
<th>Restrictions</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Regular Grammar</td>
<td>Finite Acceptor</td>
<td>( A \rightarrow (T^* \cdot V) \mid A \in V )</td>
<td>Regular language</td>
</tr>
<tr>
<td>Context-Free Grammar</td>
<td>Pushdown Automata</td>
<td>( A \rightarrow (V \cup T)^* \mid A \in V )</td>
<td>Context-Free Language</td>
</tr>
<tr>
<td>Unrestricted Grammar</td>
<td>Turing Machine</td>
<td>( u \rightarrow v \mid u \in (V \cup T)^+ )</td>
<td>Recursively enumerable Language</td>
</tr>
<tr>
<td>Chomsky Normal (CNF) form</td>
<td>Unrelated</td>
<td>( A \rightarrow BC \mid A, B, C \in V ) or ( A \rightarrow a \mid A \in V, a \in T )</td>
<td>Context-Free Language</td>
</tr>
<tr>
<td>Greibach Normal (GNF) form*</td>
<td>Unrelated</td>
<td>( A \rightarrow ax \mid A \in V, a \in T, x \in V^* )</td>
<td>Context-Free Language</td>
</tr>
</tbody>
</table>

*JFLAP v7.0 does not address GNF.

Figure 5.3 – Grammar Families – The table above shows a selection of different families of grammar, the type of automaton to which they are associated, the restrictions placed on productions in that family, and the family of languages they represent.

CNF and GNF are examples of grammar families determined not by the language they derive but rather the specific format of productions. Any CFG can be converted to CNF or GNF form. The former is accomplished in JFLAP along with other grammar transformation algorithms such as the removal of useless productions, lambda productions, and unit productions, all of which are prerequisites to the CNF conversion algorithm.
There is also a correlation between grammar families and varieties of automata as shown in Figure 5.3 FLA theory includes algorithms to convert grammars into these corresponding classes of automata. JFLAP provides wholly interactive visualizations of these conversions for regular grammar to DFA and CFG to PDA. The latter can be executed via both an LL and LR conversion algorithm.

5.8 Regular Expressions

JFLAP also has a module dedicated to RE’s. As mentioned previously, these provide another mechanism for defining a regular language by combining strings of symbols in an input alphabet $\Sigma$, parenthesis, and the operators $+$, $\cdot$, and $\ast$.

A formal definition is achieved through applying these operations, defining $\Sigma$, and holding to the three rules listed below (Linz):

1. $\emptyset, \lambda, \text{ and } a \in \Sigma$ are all regular expressions, called primitive regular expressions.
2. If $r_1$ and $r_2$ are regular expressions, so are $r_1 + r_2$, $r_1 \cdot r_2$, $r_1^\ast$ and $(r_1)$
3. A String is a regular expression if and only if it can be derived from the primitive regular expressions by a finite number of applications of the rules mentioned in (2).

REs holding to these restrictions can be used to describe any regular language. They can also be converted to and from DFA’s and be used to create regular grammars. All of the algorithms associated with these transformation or derivations are included in JFLAP.

5.8 Other components

JFLAP also includes three other modules - L-Systems, Regular Pumping Lemma, and Context-Free Pumping Lemma. The first may ultimately be incorporated into the new system explained in the remainder of this document, but as of now they will all remain in their current state. One important note is that they will all fit into the new underlying structure of the JFLAP code, but with minimal external, visual modifications.
6. Issues with JFLAP v7.0

The extent of JFLAP’s usage worldwide is a testament to the value of the software and the degree to which it has accomplished the basic goals stated above. But as with any piece of software, it can always be improved. The current iteration of JFLAP has two major limitations which make both improving and using the software more difficult and less effective respectively.

6.1 Implicit Formal Definitions

First, in working with the various modules, the user is never exposed to a concrete textbook-like representation of a formal definition. JFLAP 7.0, as with other previous versions, functions from the basis of an implicit formal definition and a series of hard-coded conventions. Should one construct an automaton in the current version (Figure 3), there is no literal representation of the input alphabet. Instead the alphabet is dynamically constructed - as new transitions utilize different symbols, those symbols are added to the input alphabet for the purposes of all future algorithms.

### Figure 6.1 – Formal Definition Representation in v7.0

This image shows the automaton editing window in JFLAP v7.0. There is no clear written representation of the formal definition of this PDA, rather it is implicit in the structure. Specifically, the definition would be:

\[
M = (Q, \Sigma, \Gamma, \delta, S, z, F)
\]

\[
Q = \{q_0, q_1, q_2, q_3\}
\]

\[
\Sigma = \{a, b\}
\]

\[
\Gamma = \{A, B, Z\}
\]

\[
\delta:
\]

\[
(q_0, a, Z) \rightarrow (q_1, AZ)
\]

\[
(q_1, a, A) \rightarrow (q_1, AA)
\]

\[
(q_1, b, A) \rightarrow (q_2, \lambda)
\]

\[
(q_2, b, A) \rightarrow (q_2, \lambda)
\]

\[
(q_2, \lambda, Z) \rightarrow (q_3, \lambda)
\]

\[
S = q_0
\]

\[
z \in \Gamma = Z
\]

\[
F \subseteq Q = \{q_3\}
\]

This definition, but without needing to preemptively derive some of the components, the meaning of this automaton as formally defining a language could be unclear to the user.
As explained in previous discussion of FLA concepts and mechanisms, the components of each formal definition determine what the machine or grammar is capable of and how it will be used. In the current version of JFLAP these components are somewhat obscured as a result of aforementioned conventions. In holding to the philosophies of software visualization, components of the formal definition, such as the terminal and variable sets of a grammar or the start state of a DFA, should be clearly represented in multiple ways with a flexible, interactive interface.

6.2 Symbols as Characters

Another drawback of the software is JFLAP’s universal interpretation of all symbols as individual characters. By convention, JFLAP alphabets are composed of alpha-numeric characters only. Grammars apply further restrictions, where a variable is identified as a majuscule character and a terminal could be any other character that the user could input. This limits the structures created through JFLAP to the description of simple, symbolic languages with a finite set of potential symbols.

As a result, JFLAP does not allow exploration of FLA theory using more complex, real-world languages such as simple programming languages. Preventing literal language implementations of such familiar topics hinders how much a student’s intuition can be leveraged to help them gain a deeper understanding of the goals and applications of FLA theory.

6.3 Unified Code Base

As a result of nearly a decade of Java code, written by a long lineage of students, the JFLAP code base has become a patchwork of separate components only superficially unified within the API. This lack of integration has resulted in the perpetuation of disparate design and coding styles which make the implementation widespread changes or paradigm shifts difficult.

The code for different modules, and therefore the modules themselves, should share as many common code elements as possible. Shared structure in the underlying design will be reflected in the user interface and is intended to maximize GUI components shared between modules. This confluence of structure is what increases the efficiency of teaching FLA theory using JFLAP, since similarities between the different interfaces design reduces the challenge of becoming familiar with the platform as a whole.
Thus a reevaluation and partial rewrite of the code is eminent and necessary, especially in light of the global adjustments made to implement explicit formal definitions. This should also make the JFLAP API far more useful in open source FLA-related projects.

6.4 Undo Mechanism

In JFLAP v7.0, undoing and redoing is only possible in automata and is limited by a greedy use of memory. This means there is no method for undoing edits to the productions of a Grammar, adjustments to regular expressions, or edits in L-systems. And in automata, a user can only go so far back in their history of edits. This limitation exists because, once an undoable action occurs, the current undo keeper simply stores a clone of the former state of the automaton. This takes time and exhausts memory. Therefore, in order to retain efficiency in the execution of the rest of the program, there is a default thirty action limit to what can be held in the undo and redo queues.

Additionally, the adjustments made to transform the current system of implicit formal definition into something more explicit and complex would benefit from a more flexible and efficient undo/redo system. Inherent in increasing the potential complexity of a formal definition is the fact that there is also more room for error. In line with the tenants of SV, an intuitive undo/redo system could help to compensate for some of the increased tedium and frustration.

6.4 Improving Documentation

As JFLAP has expanded to include increasing amounts of material, the various guides, wikis, and help system have not grown to mirror this change. A lack of documentation on newer JFLAP components has left the program with a variety of useful yet unused tools.

This same issue is readily apparent in the code. When JFLAP was rewritten into Swing, the students responsible for this transition left detailed, clear documentation. Yet subsequent updates to the functionality of classes and methods have lacked such rigorous note-taking. This makes comprehension of the code difficult and tedious at times, and perpetuates the aforementioned disparities in the code base as classes that cannot be understand are difficult to use in extensions. On top of this, there are instances of unused classes and methods which make the application of the JFLAP infrastructure challenging, as a developer risks incorporating a defunct component into their code.
6.5 Feedback

One of the primary tenets of software visualization and education is providing valuable feedback upon which the user can form inferences, recognize problems, and execute intelligent solutions that require application of previous knowledge. Without this feedback, users are often forced to resort to arbitrary efforts that, while still reaching a correct answer, contribute minimally to their understanding of the problem. As a combined product of the aforementioned issues, user error feedback and automated responses in JFLAP are inconsistent in their helpfulness. In general, JFLAP could benefit from a simple, unified system of presenting errors to the user that would provide them with valuable feedback and be easy to implement.
7. Research Goals

The primary goals of this project are split between two different yet intrinsically linked efforts. These can be summarized as the implementation of a new string-based Symbol paradigm and general housekeeping, updating, and modification of the code base.

7.1 String-based Symbol Paradigm

1. Transforming the idea of a symbol from a single 8-bit character to an array of characters, i.e. a string, thereby allowing for the design and exploration of literal languages in JFLAP. This shift must preserve backwards compatibility and previous functionality.

2. Describe and present a clearly defined set of explicit rules which govern the restrictions imposed by JFLAP on the addition of symbols to alphabets in a given formal definition.

3. Provide an interactive, intuitive interface through which users can create symbols, add them to alphabets, modify those alphabets dynamically, and see how it affects the formal definition.

7.2 Interface Modification and Code Housekeeping

1. Add more intuitive, user friendly controls to the Grammar and Automaton editing interfaces, including an action based undo/redo mechanism to create a more flexible and efficient history of editing actions.

2. Improve the error feedback upon this creation of symbols as well as throughout the program, such that users receive valuable information to rectify their mistakes.

3. Restructure the JFLAP code base to unify the many disparate modules under a single API and simultaneously improve the internal documentation for the API such that it may be more effectively reused in future JFLAP projects and in open source programs which use the JFLAP library.

7.3 Assessment

1. Perform preliminary testing and assessment of the new version of JFLAP in order to confirm the value of the new additions and to evaluate the degree to which the aforementioned goals were achieve.
8. String-based Symbol Paradigm

This goal of this work has been to ameliorate these issues through additions to and modifications of JFLAP. To this point, this includes a new, explicit formal definition system and a large-scale refactoring of the JFLAP v7.0 code into a pure, model-view-controller design backed by an action driven undo/redo infrastructure.

8.1 From Characters to Symbols

The first step in this project was to eliminate the mandate that individual characters correspond to a single symbol. Implementing symbols capable of having multi-character representation allows for larger automata, more complex languages represented by those automata, and the implementation of non-symbolic languages with literal, relatable applications.

8.1.1 Symbol API

In JFLAP v8.0, a Symbol class has been created which allows the concatenation of multiple characters to represent a single symbol. Each individual character is no longer assumed to be an independent symbol. Instead a Symbol object now exits, containing a Java string object. Symbols serve as building blocks for SymbolStrings, ordered lists of symbols which correspond to the idea of an input string (in Automata) or a derived string (in grammar). This system allows complex, literal symbols, yet does not inhibit backwards compatibility as a Symbol of length one is essentially a character. Therefore it is still possible to define single-character symbolic languages as in previous versions of JFLAP.

Figure 8.1 – The JFLAP v8.0 Symbol Paradigm Shift - The figure above shows the general trends in the movement from characters to string symbols and how that change will affect other classes in JFLAP v8.0.
8.1.2 Alphabets and Symbol Sets

In order to have a language, one must first have a clearly defined finite set of symbols to constitute the vocabulary of that language. Specifically, languages require an alphabet. Alphabets are finite sets containing all symbols needed to describe a formal language. In FLA theory, the English dictionary is analogous to the alphabet of the English language. Alphabets are a specific subset of formal definition components.

In JFLAP v8.0, before a symbol can be used in language functions of a given FLA structure, they must first exist in the alphabet. This enforces the idea of a language as a combination of symbols and rules. When symbols are removed or modified, those changes are reflected across any object using the adjust symbol.

8.1.3 Explicit Formal Definitions

In the new version of JFLAP, each formally defined object, namely automata, grammars, and regular expressions, now have an explicit formal definition. A new package within the code provides an abstract framework for applying a formal definition to each of the aforementioned structures. Each formal definition is comprised of the appropriate n-tuple of formal definition components. Each component is capable of existing independent of a formal definition, but acquires slightly different capabilities once it has become included into a specific definition. (Figure 8.2)

The approach to formal definitions is very literal. From this structure, you can retrieve a clear sense of what constitutes a given automaton or grammar simply from a glance at which components are added to them. Formal definitions also have the option to be checked for completeness now, which iterates across the components of the definition and checks if each is sufficiently defined, i.e. $\Sigma$ is not empty in an FSA or the start variable $S$ has been set in a grammar.

8.1.4 Meta-definitions and Default Definitions

One risk with changing the core of a program like JFLAP is the potential for preventing backwards compatibility, or accidentally removing a function already in the software. JFLAP now carries with it a default definition which is roughly the definition that would result from the application of previous conventions. The symbols in the alphabets of this definition span the set of individual characters considered valid in previous versions of JFLAP. The default definitions corresponding to each different module are all held inside of a single meta-definition and through this object the definitions are applied to their associated contexts.

$$Ex.\ default\ Input\ Alphabet\ \Sigma = \{a, b, c, d, e \ldots, z, 1, 2, 3, \ldots 0\}$$
A meta-definition can be defined formally as an n-tuple, where n is the number of unique, individual formal definitions which the meta-definition contains. For a meta-definition $D$, a grammar $G$, a DFA $M_1$, and a PDA $M_2$, $D$ might look like: $D = (G, M_1, M_2)$ It is important to note that a meta-definition may only contain one formal definition of a given variety at a given time, i.e. only one DFA or one regular expression.

Figure 8.2 - Formal Definition API - Shown above is the basic formal definition API that now underlies JFLAP. The diagram specifically highlights how the hierarchy is design in the case of a grammar. Note the insatiable components highlighted in red. Those are the classes which constitute the formal definition of a Grammar.
A new module has also been added which allows for the creation and modification of these meta-definitions. It gives the user the ability to import various definitions used in the past, store those as a complete meta-definition, and then import them into specific modules in the future. One can also make additions to the default JFLAP definition. Once saved, a meta-definition can be imported in JFLAP as a custom default definition and have its subcomponents automatically applied to each module of JFLAP. This means that the user can now transform the entire program into an effective tool for working with a set of definitions that they designed in as complex or simple a way as possible.

8.2 Symbol Rules

In an effort to make any restrictions on the user explicit, there is now a rule hierarchy encompassing set of rules that govern the addition, modification, and removal of symbols for a given alphabet. Explicit presentation of these rules allows the user to understand exactly why a symbol is valid or invalid upon attempted addition to an alphabet. These rules prevent ambiguity when tokenizing a user-input string into a symbol string.

8.2.1 Rule Paradigm

Every component in the new explicit formal definition system has been designed to stand alone. Therefore, rules are applied directly to alphabets rather than to the overarching formal definition. Each alphabet carries an adjustable list of rules which are checked upon each addition to, modification of, or removal from the alphabet. New restrictions are added to alphabets upon being added to a given formal definition. This means that there are certain limitations intrinsic to a given type of alphabet, and there are also restrictions imposed by the formal definition containers. This rule paradigm ensures that alphabets can now exist outside of a formal definition and still be affected by the proper limitations.

JFLAP now has an explicit “rulebook” which contains a set of alphabet rules. This rulebook can be explored by the user via a simple interface so that they can understand how and why alphabets interact with each other and their associated formal definitions. Each rule also provides unique feedback based on the situation which can then immediately be transformed into an error report in the case of a bad symbol.
Figure 8.3 - The Rule API – The UML diagram above gives a quick overview of the different rules now in place in JFLAP, and how they are linked together. The goal is to make it easy to understand what these rules do, how they relate to each other, in which alphabets they belong. Additionally, the RuleBook, shown here in the upper right, is a database of every rule which can be accessed by the user for a more in depth description of the system.
8.2.2 Default Rules

The following rules are used in the default situation, in every case of symbol editing unless another rule specifically supersedes this argument.

1. **Self-Identical Rule:** No symbol may be identical to any other symbol in the same alphabet.
   Algorithmically, this rule also ensures that the parsing of a string of $n$ characters into a symbol string on alphabet $\Sigma$ will have the worst case runtime:
   $$O(m, n) = m \cdot n, \quad m = |\Sigma|$$

2. **Disallowed Character Rule:** No symbol may contain a disallowed character.
   This includes such symbols as:
   - $\Box \rightarrow$ Turing Machine blank symbol
   - $! \rightarrow$ null string in LL(1) parsing module
   - $\lambda$ or $\epsilon \rightarrow$ the empty string symbol depending on current mode
   - $\sim \rightarrow$ special symbol reserved for Turing machine transitions
   - $\$ \rightarrow$ end of string marker in LL(1) parsing algorithm

8.2.3 Definition Specific Rules

For each module in which formal definitions and alphabets are utilized there is a unique set of rules that deal with the addition of new symbols to a given alphabet in the context of that formal definition. This ensures that symbols remain well defined, even in the presence of other alphabets or more complex formal definitions.

1. **Turing Machine Rule:** Any symbol adjustments (add, remove, modify) made in the Tape alphabet ($\Gamma'$) will be applied to the Input Alphabet ($\Sigma$)
   
   **Purpose:** This simply follows the definition of a Turing machine - $\Sigma \subseteq \Gamma - \{\Box\}$

2. **Grammar Rule:** A variable may not be identical to any symbol in the terminal alphabet and vice versa.
   
   **Purpose:** Similar to the Self-Identical Rule, this rule simply ensures that any pair of variables and terminals in the same grammar is differentiable.

8.2.4 Grammars – Using Delimiter-Pairs

When creating a grammar you can now use a delimiter pair, such as `(‘ and ‘)`, with variables. The options for grouping characters are limited to a specific set of pairs in JFLAP which do not conflict with any special
characters in proofs, constructions, or other JFLAP features. Should you choose to use a given grouping pair, those symbols immediately become disallowed (see Rule 2 in Default Rules) for the terminal set. A grouping pair further differentiates variables and terminals from a visual perspective, and circumvents the grammar rule above, since variables will be required to have a grouping pair.

When you choose to apply a grouping pair, the following two rules are added to the variable and terminal alphabets as described:

1. **Each variable must have only one open delimiter character as its first character and only one close delimiter character as its last character**

   Using ‘(’, ‘)’ as the delimiter pair, NOUN and N(OU)N are invalid, (NOUN) is valid.

2. **A symbol using a delimiter pair may not be added to the variable set if there is already a terminal corresponding to the internal string of that symbol (and vice-versa).**

   For a grammar, \( G = \{ V, T, P, S \} \), if \( \text{blue} \in T \), then \( \text{(blue)} \) is ineligible as a variable. However, \( \text{(BLUE)} \) could be added to \( V \) as the string comparison is case sensitive.

Although this makes the addition of variables somewhat tedious, it is an essential component to the new system. This is because, when an FSA, PDA, or TM are transformed into grammar, the result often has a tremendous number of productions corresponding to the number of states in the automata. Using delimiter pairs allows for a direct mapping of state names to variable names and ensures that the resulting grammar is easier to understand in the context of the Automata which produced it. The user can note how the proof executed since the basing of variables on state names is preserved. They will also be able to explore how different components of the automata formal definition translate into the grammar formal definition, such as alphabets, state states, end states, etc.

**8.3 Formal Definition Interface**

Arguably the most important element of the new formal definition system is its implementation as an interface within the already established JFLAP GUI. The goal of the interface is to promote user interaction, to help them understand the connection between the formal definition and the representation on screen, and finally to provide as much information as possible without overwhelming the user. Additionally, each component is structured to stand alone. Like formal definition components themselves, their GUI equivalents can be used outside the formal definition interface. This stand-alone approach allows the different components to be applied in a variety of environments.
8.3.1 User-Defined Mode

Automatically using the default definition can be toggled off and user-defined mode activated. In this mode, the user is prompted with a definition creation dialog immediately when they begin to construct an automaton or grammar. This dialog enforces the completion of alphabet-related components prior to the continued editing of the formal definition. This includes components such as alphabets, the bottom of stack symbol (PDA) and the start variable (grammar). (Figure 8.4)

In user-defined mode, students will need to use foresight and planning when designing an alphabet so that they include all necessary symbols. This results in the interactive application and reinforcement of knowledge, processes which are missing from JFLAP v7.0 and earlier. However, alphabets are not immutable. If mistakes are made or symbols are overlooked in the initial construction then the alphabet can be adjusted via the formal definition panel outlined below.

8.3.2 Formal Definition Panel

The formal definition panel is a new panel found at the bottom of every window representing a formally defined object. It contains the name of the object and a representation of every alphabet in the current formal definition. Each alphabet is a horizontal toolbar with buttons corresponding to every symbol in that alphabet. When clicked, these buttons add the associated symbol to whatever text field is currently active. This makes it much easier to deal with alphabets containing longer symbols, for the user can simply click those symbols into any location in which the symbol’s string representation is needed.

This panel also allows for the dynamic adjustment alphabets in the formal definition via a right-click menu. This menu gives three options – add, modify, and remove the symbol. Because of this, mistakes made while creating an alphabet through user-defined mode are never permanent.

Finally, the panel provides a method for setting the special symbol of an alphabet when applicable through the same right-click popup menu. This applies to the bottom of stack symbol in the stack alphabet of PDAs and the start symbol in the variable alphabet of Grammars. Depending on the implementation of TM’s, the blank tape symbol may also serve as a special symbol in the tape alphabet. In Figure 8.4, the start variable had been set to (NOUN) and is indicated as such by the light yellow background color.
Figure 8.4 - Building a Grammar in User-Defined Mode - The top figure is the definition creation interface. In User-defined mode, this window will appear each time you create a new formally defined object. The bottom figure shows how the user-defined alphabets facilitate complex structures, such as this English language grammar.
8.3.3 Definition Creation Dialogue

As mentioned above, and shown in Figure 8.4, a special dialogue appears to design alphabets prior to the construction of the automaton or grammar. This definition creation dialogue consists of a definition panel as described combined with an improved ability to edit the alphabets. This includes a copy and paste button for the contents of entire alphabets, a clear button, a text field to input directly into the selected alphabet, and an import menu for retrieving alphabets from files.

This import system can be used as an alternative to changing the overarching default definition. The import menu contains options to import “from file” or “from default definition”. Importing from the default definition is straightforward and sets the alphabet to exactly what it would be in the default mode. Through the “from file option”, you can import whole meta-definitions or individual alphabets that have been previously created and saved. Even if the imported definition is for a different kind of structure, i.e. a grammar imported into an FSA, the imported symbols will still be added to the alphabets being edited. This same editing interface is used in the new meta-definition creation module, with minor modifications to accommodate a larger set of alphabets. (Figure 8.5)

8.3.4 Meta-Definition Creation Module

One final tool newly added to JFLAP is the Definition Creation Module. (Figure 8.5) Accessible from the main menu, this module allows for the completion of custom meta-definitions which can be exported as .jdef files, a new JFLAP extension used specifically for these types of files. The meta-definition produced, once saved, can have its constituent definitions imported into the construction of any specific definition’s language. For example, when a user is in user-defined Mode and wants to apply the FSA input alphabet from their latest constructed meta-definition, they simply import that .jdef file using the File->“Import from File” command, which will pick out the appropriate input alphabet from the meta-definition and apply the constituent symbols of that alphabet to the user’s currently constructing language.

Moreover, any complete meta-definition can be used to override the default JFLAP backwards compatible character-based symbolic meta-definition. As a result, the definitions contained in the new meta-definition will be the ones imported by default in every new formal definition, thereby mitigating the amount of repetitive alphabet construction/symbol addition that must occur.
Figure 8.5 – Definition Creation Panel – Above is a screenshot of a meta alphabet currently being constructed through the new definition creation module, as highlighted above in the main-menu inset (1). In this case, the interface is being used to construct a new default definition which, after being saved, can be imported into JFLAP as the default meta-definition. This means that the user can design a series of varying formal definitions under a single meta-definition header, and then use those definitions to import symbols or customize the default alphabets. (2) This mode comes with special editing tools that speed up the process of building the alphabets. Copy will store an entire alphabet of the user’s choosing (the selected alphabet) and all of the symbols from that alphabet can then be pasted into a different alphabet. This same interface is provided in the definition creation dialog of user-defined mode while the user is defining a single alphabet, greatly speeding up that process. Finally, the ability to import and export other definitions from a file (or from the default alphabet) can save time for the user as they design their definitions.
9. Interface Modification and Code Housekeeping

Since the aforementioned changes affect most of JFLAP’s modules, an updating of the code base to current Java standards and refactoring of the overall design became a logical extension. These adjusts seek to make the user-facing program more intuitive, and to transform the code base into a more robust, unified, and flexible library.

9.1 Updated Editing Interfaces and Undo/Redo System

Following the implementation of explicit formal definitions, a refactoring of the undo/redo system was executed with the goal of giving every module access to undo and redo, linking these actions to familiar hotkeys, and decreasing the memory limitations previously present in the clone-based undo keeper.

Previous versions of JFLAP applied undoing and redoing via a tool-like interface, requiring the undo/redo tools to be toggled on before the adjustments could actually be applied. This system has been eliminated, and a new more familiar click-to-undo system has been implemented. The undo and redo buttons are no longer considered tools, and can be left clicked for a single application or right-clicked for multiple application.

Additionally, as a byproduct of the improved flexibility in an action-based system, many new features have been added to the basic automaton and grammar editors to streamline editing, provide immediate and useful feedback, and imitate familiar software paradigms thereby decreasing the JFLAP familiarity barrier by calling on expertise accrued through utilization of other editing programs. The new design features a shift from supplying information to the user at “run-time” to returning important feedback immediately at “compile-time” (when needed) of a formally defined object. In other words, feedback on the correctness of an automaton or grammar will mostly happen as each is constructed rather than when that object is used in an algorithm or saved.

9.1.1 Action-based Undo/Redo System

The new undo keeper maintains two FIFO queues of undoable actions. These actions all have an undo and redo method unique to the action rather than simply setting the model to an old or new clone. Each action, upon being performed, is automatically added to the current undo keeper. The undo keeper can then call the undo and redo functions of the undoable action class as needed.
Undoable actions are now linked to every modification in JFLAP, both from editing the structure of the alphabets to adjusting the productions in a grammar to any change made to an automaton.

Undo and Redo as tools have also been eliminated. Although the buttons remain on the toolbar, they no longer require you to click into the pertinent pane to execute the undo or redo. Instead the actions are much more like other software, executing on the click of the button and using as hotkeys the ctrl-z and ctrl-y keys respectively.

9.1.2 Changes in the Automaton Editing Interface

The changes in the automaton editor have been manifold. First, the construction, bending, and dragging of transitions has been adjusted to work appropriately, with transition labels remaining stacked on top of a bending arc and that arc remaining stretched in the same way relative to its starting and ending states. The transition editing table is now in direct contact with the transition and thus automaton it is editing. This allows for immediate error responses as the user’s input string is checked against input alphabets and language rules already present in the formal definition. There is now a hotkey linked to delete (ctrl-D) which deletes all selected states and transitions from the automaton. Transition arrows are now independent of the functions themselves. One can now select a transition by clicking on its label, and that will not highlight the arrow. Or the arrow can be highlighted, signifying that the entire “transition stack” is selected. The right click menu has also seen many useful additions, including right clicking on components and being given the option to edit them or delete them.

9.1.3 Changes in the Grammar Editing Interface

The grammar editor interface has experienced a similar reconstruction and increase in modification options, primarily focused around the deleting, inserting, and reordering of productions. In previous versions of JFLAP, empty transitions remain blank in the production table, duplicate productions are allowed, and productions cannot be reordered. Now the grammar editor will automatically trim empty transitions. Similarly, it is no longer possible to have duplicate productions. It is also possible to delete a selected production and undo/redo actions in the grammar editor, features that were previously found only in the automaton editor.
This example of an English language grammar is just one of the new ways in which the improved capabilities can be realized in more complex grammar structure. This example of English language grammar is just one of the new ways in which the improved capabilities can be realized in more complex grammar structure. This example of an English language grammar is just one of the new ways in which the improved capabilities can be realized in more complex grammar structure. This example of an English language grammar is just one of the new ways in which the improved capabilities can be realized in more complex grammar structure.
9.1.4 Modify Menu

As a byproduct of new menu options and the presence of undo/redo in most modules there has been a new modify menu placed on the upper menu bar for both grammars and all automata. The goal of this modify menu is to consolidate into a single logical menu every modifying action which can be applied without using a new algorithm/proof view. The menu can be split into 3 portions, namely: generic editor actions, module-specific actions, and alphabet affecting actions.

Generic editor actions concern those actions which are consistent across platforms such as undo, redo, and delete selected. Module specific actions change dependent on the module being used. This means that automata will see such new actions as “Sort Transition Labels” or “Rename all states” in conjunction with more familiar, traditional actions such as the graph layout menu. All of these actions apply superficial modifications that can always be undone. Finally, alphabet affecting actions, although generally present across every editor interface, provide a variety of ways in which to cosmetically adjust the formation of symbols in the alphabet and the appearance of those alphabets.

Figure 9.2 – Automata Modify Menu – The new modify menu, present in both automata and grammar, contains various new and old options to modify such aspects of the display as the transition graph, the alphabets, the graph layout, or the productions in a grammar. This menu also contains an alternative way to access the new undo and redo features.

9.2 Improved Error Reporting

A new system of error reporting has been implemented in the hopes of simplifying the challenges of getting detailed feedback out to the user. A new class, BooleanWrapper, provides a simple wrapper to combine a boolean object and string message. Methods can now return both if they failed but also why, providing a mechanism for the exchange of important error dialogue. These wrappers can then be directly transformed in JOptionPane error or in the throwing of exceptions.
Another very important benefit to this system is the ability to generate strings at the location in code where the error event took place, thus enabling more specific messages rather than previous generic, less helpful equivalents. Errors can now not only tell the user that something went wrong, but also where, how, or why it did so. Without telling them the exact issue, these messages promote active incorporation of previous knowledge to reach a rapid solution.

### 9.3 Code Unification - MVC Architecture

In an effort to unify every module under a centralized code base, a pure model view controller design has been applied to JFLAP. Each window corresponds to a controller with an associated model-view pair. Each of these controllers is registered to the Universe, a singleton class which only exists once in each JFLAP program. By having every module linked to a single controller, changes to every model can be added to the undo keeper of that model’s controller. Thus the capability to undo and redo actions can be extended to every module of JFLAP through a common undo keeper class.

The goal of this refactored design effort is to enforce good design on future extensions to JFLAP and make it easy to follow the good design laid out in the API. Additionally, having a unified system with shared components makes future refactoring’s and large scale additions like those described in this project’s new Symbol design much more simple to apply on a global scale.
10. Assessment Survey and Results

In order to judge the efficacy of the changes outlined in this document and executed in JFLAP v8.0, a small scale beta test was performed which included a brief assessment survey. The entire results of this survey, as well as a copy of the survey itself, can be found in the appendix of this document (Appendix 1-3).

10.1 The Survey

The survey was performed on a small scale with a total of 12 students participating, all from the basic FLA course at Duke University. The students were provided with a beta version of the JFLAP v8.0 software, an explicit step-by-step guide on what and how to use the new version, and then left to explore the program with these guidelines and the associated assessment sheet.

The survey was composed of three different exercises which tested various functionalities of the new software, each followed by a short set of assessment questions. The first exercise worked with regular grammars and FSA’s to exhibit the backwards compatibility of JFLAP v8.0 and to highlight the direct translation of state names into variable names and vice-versa when RG’s and FSA’s are interconverted. Also, in having the students build an automaton, the hope was to prompt instances of new, explicit feedback and allow them a glimpse of the improved automaton editing capabilities. The second exercise was an exploration of a simple PDA and the conversion of that PDA to a CFG. The goal here was to show how the formerly temporary variable names (i.e. \((q0q3)\)) that were constructed in this algorithm could now be directly carried over into the grammar, thus preserving some of the meaning of the automaton from which the grammar was constructed. Finally, the third exercise asked students to employ the new user-defined system to construct a complex, literal language they were all familiar with. The example used was a specific small-scale programming language designed for the FLA course in which the survey was given, known as MouseCat.

The assessment of the above exercises contained two different types of questions. The first was a statement or claim regarding the new version of JFLAP followed by a Likert scale bubbling response. One example of such a question and the visual representation is show below:

<table>
<thead>
<tr>
<th>The new version of JFLAP...</th>
<th>Strongly Agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly Disagree</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>... provides a simple and useful interface for presenting formal definitions and symbols in grammars and FA’s</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
<td>o</td>
</tr>
</tbody>
</table>
The other section of the assessment was a series of open-ended written-response questions. This included such queries as “Was the process of constructing a grammar from scratch easy and intuitive?” or, “Compared to your experiences with the previous version, do you feel like the additions in JFLAP v8.0 would help you understand FLA theory better?” For the context of this second question, it is important to note that the participants in this survey had been using the old version of JFLAP for three months. Thus they each have significant experience with and previous knowledge of the program. This likely facilitated their learning of new JFLAP additions and provided a basis for comparing v8.0 to the previous version.

### 10.2 Results

Overall, the results of this survey trended toward a positive reaction. In order to analyze the aggregate responses from the Likert scale questions, each option was assigned a numeric score:

- **Strongly Agree** = +2
- **Agree** = +1
- **Neutral** = 0
- **Disagree** = -1
- **Strongly Disagree** = -2

Based on this scale, the maximum positive response any one statement could receive is for every participant to strongly agree, therefore yielding an aggregate score of $2 \times 12 = 24$ point score. Of the 18 Likert-based assessment questions, 10 of the 18 yielded an average score greater than 1, meaning that each participant on the exam tended to agree or strongly agree that JFLAP v8.0 had accomplished the state task. Not a single participant strongly disagreed with any of the statements, and no more than 2 students disagreed with any one statement.

The highest average assessment came with the construction and editing of the complex grammar, the 3rd survey exercise described in the previous section. Eleven of the twelve students, 91% of the participants, agreed or strongly agreed that JFLAP v8.0 allows the user to represent a literal language like the complex literal grammar more clearly than the previous version. This yielded an aggregate score of 17. Similarly, ten of the twelve students agreed that the new interface makes it easy and intuitive to construct custom alphabets with complex, multi-character symbols, summing to an aggregate score of 16. (See question 3 results, Appendix 3)

The other area of success, based on this survey, was in the translation of a DFA to a RG, where the new multi-character symbol alphabet allows for the preservation of state names as variables in the regular grammar.
Again, eleven of twelve students agreed or strongly agreed that this system makes it easier to understand how an FA relates to a regular grammar by directly translating state names into variables. (See question 1 results, Appendix 3)

The lowest or least consistent responses had to do with the feedback provided by the new system and the comparison of the new JFLAP environment to the old one. Only five participants agreed that JFLAP v8.0 “gives constructive feedback while building the alphabets, explaining clearly what went wrong and how to fix it.” Similarly, of the ten students who responded to the claim, six agreed, two disagreed, and two were neutral on whether the new version of JFLAP provides an environment identical to previous versions of JFLAP simply with more information available. The latter result could be a product of using such a literal word as “identical,” which might have misled students into a direct, visual comparison between the two version of JFLAP rather than an appraisal of functionality.

Of the written responses, few were of publishable value. Among the positive statements received were such comments as “[JFLAP v8.0 has a] generally easier interface...smarter JFLAP interactions, easier to see mistakes and quickly fix” regarding the additions they liked the most, and “I feel like it does since it gives us more flexibility to work with, analyze, and personalize the modes,” regarding whether or not the new version of JFLAP will improve FLA theory education.

**10.3 Conclusion**

Generally, the feedback was good and provides valuable, albeit inconclusive evidence that the new extensions to JFLAP, the implementation of an explicit formal definition and complex symbols, enhance the flexibility of JFLAP and improves the capabilities of the software toward becoming a more comprehensive FLA suite. However, as the low positive results suggest, some of the goals were not achieve. These responses, in addition to some of the written requests for features, corroborated the idea that the interface could be restructured and provide more clear access to some of its components. Additionally, it seems as though the ideal of providing more detailed and productive feedback fell short of initial goals. These observations will provide helpful guidelines for final editing of the JFLAP v8.0 extension before it is tested on a larger scale.
11. Future Projects/Continued Work

11.1 Continued Work on Additions and Refactoring

Although a significant amount of the work done toward implementing the new explicit formal definition system, MVC design, and undo/redo functionality has been completed, there is still much left to do. First and foremost, the algorithms and proofs present in JFLAP v7.0 must be updated to reflect the new definition of a symbol as a user-defined object from its old definition as a single character. This will involve some major refactoring, and will require detailed exploration of each algorithm to ensure its functionality is preserved in the new system.

Additionally, as mentioned earlier, the ultimate goal is to extend the symbol system and undo/redo system into L-Systems and regular expressions to make these modules less subject to user error and more capable of thoughtful examples and exploration. The difficulty in this lies in the fact that there has never existed a standard for a “default” symbol alphabet and thus much of the work would need to be explored in depth before implementation in order to prevent misguiding of users and loopholes in design considerations.

11.2 Template Meta-definitions

Leveraging the import functionality of JFLAP and the meta-definition creation system, it would be nice to design a collection of canonical meta-definitions in which the capability of the new version of JFLAP is really explored. For example, creating expression-based, English-base, or Java-based meta-definition template would enable educators to simply import these as their default definition and that use the definitions to create complex machines and grammars without needing to implement the alphabets themselves. This would also be a good way to show users how the system is intended to be used, and to what extent it can really magnify the potential of JFLAP as a learning tool.

11.3 Testing and Assessment

It will be important in the immediate future to do more significant, semi-quantitative analysis of the effectiveness of these additions. This testing would be in an effort to help further modify and improve the changes made to JFLAP as well as to confirm that those changes were beneficial to learning and expedite exploration of FLA theory concepts. This will likely begin with additional small scale testing in a classroom setting at Duke University followed by a beta-version release to the international JFLAP community.
11.4 Improved Documentation and Online Help Resources

Documentation is fundamental to help both users working with the program and coders working on the program. The former is one of the most important factors in helping to shrink the initial learning barrier when beginning work on the platform. The latter is important to preserve the currently implemented design and help code contributors to use the already constructed infrastructure in future JFLAP extensions.

11.4.1 Help System

An update of the intra-JFLAP help system could potentially make information even more accessible, and the software more usable. What would be ideal is a linking up of the wiki information into the GUI via HTML rendering Swing panes. These panes could then serve as a basis for a native tutorial and help system that can give immediate feedback and assistance to the user, without requiring access to the internet wiki. Other programs have a similar built-in tip or help system that assists in the exploration of the software and expedites familiarity.

11.4.2 A Guide for Future Coders

In order to encourage future students working on JFLAP to use the API constructed for JFLAP v8.0, it would be of great benefit to create a detailed compendium of the infrastructure of the code, a guide to its new design. This would explain how to use the different components of the new system from an example-based perspective rather than simple Javadocs. In this way, future contributors will be far more likely to use components created by former JFLAP coders because they will have a central repository to explain to them just how the code is supposed to work.
References


Genkins 41
Appendix
1. Suppose you have a language over the alphabet $\Sigma = \{a, b\}$, where the number of $a$’s and the number of $b$’s are both either even or odd.

   a. Write 3 strings in the language and 3 strings not in the language

   b. Construct a DFA for this language
      “Main Menu $\rightarrow$ Finite Automaton”
      In your construction, test some of the following features:
      - New undo/redo buttons (hotkeys ctrl-Z, ctrl-Y)
      - Delete all selected objects (select with mouse and type ctrl-D)
      - “Modify” $\rightarrow$ “Sort Transitions”

   c. Rename each state to something other than “qN”.
      Recommended naming: $q0 = S$, $q1 = A$, $q2 = B$ ...
      To rename a state:
      - Right click on the state
      - Select “Set state name.”
      - Input the new name a click “Ok” – NOTE: 2 states may not have the same name.

   d. Trim the alphabet.
      Menu: “Modify” $\rightarrow$ “Trim alphabets”

   e. Run the strings from part (a) on the DFA
      Menu: “Input” $\rightarrow$ “Simulate Input” or “Input $\rightarrow$ “Fast Run…”

   f. Perform a DFA to RG conversion
      Menu: “Convert” $\rightarrow$ “Convert FSA to RG”
      - Click on transition labels to process a transition into a production or just click the “Show all” button to finish the conversion
      - Click the “Export” button to open the completed grammar in a new window

   g. Sort the productions in your new grammar.
      “Modify $\rightarrow$ Sort Productions”
      This will sort the productions lexicographically with the start productions first.

   h. Convert the RG back to a DFA.
      “Convert $\rightarrow$ Right linear Grammar to FA”

COMPLETE ASSESSMENT QUESTION 1 NOW
Assessment questions can be found on page 5 of this survey.
2. A language for regular expressions – PDA’s and CFG’s

   a. Open file “2_pda.jff”

   b. Write 5 strings that are accepted by this PDA

   c. Now convert the PDA to a CFG.
      “Convert” → “PDA to CFG conversion”
      • Click transitions until all have been added to the grammar pane on the right.
      • Note the contents of the variable alphabet. (lots of symbols!)

   d. When done, click “Export.”
      • When prompted to “Trim to grammar” click yes and note how the grammar changes – both the
        alphabets and production set should have been reduced in size

   e. Observation: Take a moment to study the resulting grammar and relate it to the PDA from which it was
      derived. No written response needed, just think about these questions.
      i. How do the variables in the grammar relate to the stack in the PDA?
      ii. If you did not have the PDA to look at, could you quickly recreate it from the grammar because
          of the specifically formatted variables?

COMPLETE ASSESSMENT QUESTION 2 NOW
Assessment questions can be found on page 5 of this survey.
3. Creating a complex, literal grammar

In this exercise you will be using the new capabilities of JFLAP to construct the MouseCat grammar as shown below.

MouseCat grammar $G = (V, T, S, P)$

$P:$

(1) \( <\text{Program}> \rightarrow \text{size} \ \text{int} \ \text{int} \ \text{begin} <\text{List}> \ \text{halt} \)
(2) \( <\text{List}> \rightarrow <\text{Statement}> ; \)
(3) \( <\text{List}> \rightarrow <\text{List}> <\text{Statement}> ; \)
(4) \( <\text{Statement}> \rightarrow \text{cat} \ \text{var} \ \text{int} <\text{Direction}> \)
(5) \( <\text{Statement}> \rightarrow \text{mouse} \ \text{var} \ \text{int} <\text{Direction}> \)
(6) \( <\text{Statement}> \rightarrow \text{hole} \ \text{int} \)
(7) \( <\text{Statement}> \rightarrow \text{move} \ \text{var} \)
(8) \( <\text{Statement}> \rightarrow \text{move} \ \text{var} \ \text{int} \)
(9) \( <\text{Statement}> \rightarrow \text{clockwise} \ \text{var} \)
(10) \( <\text{Statement}> \rightarrow \text{repeat} \ \text{int} <\text{List}> \ \text{end} \)
(11) \( <\text{Direction}> \rightarrow \text{north} \)
(12) \( <\text{Direction}> \rightarrow \text{south} \)
(13) \( <\text{Direction}> \rightarrow \text{east} \)
(14) \( <\text{Direction}> \rightarrow \text{west} \)

$V = \{<\text{Program}>, <\text{List}>, <\text{Statement}>, <\text{Direction}>\}$
$T = \{\text{size, begin, halt, ;, cat, var, mouse, hole, move, repeat, end, north, south, east, west, clockwise, int}\}$
$S = <\text{Program}>$

a. Go to the main menu.

b. Open the “Preferences” pop-up menu.

c. Check the option that says “User defines symbols.” This will put JFLAP into user-defined mode.

d. Begin constructing a new grammar.

e. Select “< >” as your grouping pair. Click “Ok”

Like the representation of the MouseCat grammar above, this will force your variables to have these two characters flanking them, therefore making the productions more readable.

f. Construct the variable and terminal alphabets as described above.

- You can add symbols either by right-clicking the alphabet and selecting the “Add” option from the menu or using the text input bar in the bottom of the dialogue.
  - The latter option will add the symbols to your currently selected alphabet
  - To select an alphabet, simply click on it.(it will turn blue)

- In both cases you can add multiple symbols at one time by separating them with a space.

- If you make a mistake, right click the incorrect symbol and select the “Modify” or “Remove” option to modify or remove that symbol respectively.

- Finally, to set a start variable, right click on the symbol in your variable alphabet which you wish to make the start variable and click the “Set State Variable” option.

g. Once the alphabets are complete, click “Ok”. The grammar window should now appear with your user-defined alphabets at the bottom.
h. Construct all of the productions that define the MouseCat grammar, as shown above.

To create productions easily:
- Begin editing (double-click into) the part of the production (LHS/RHS) that you wish to modify/create
- Click on symbols in your alphabet to add them to the entry you are currently editing
- Press enter once the production is complete.

COMPLETE ASSESSMENT QUESTIONS 3 & 4 NOW
Assessment questions can be found on page 5 of this survey.
A2. Survey Assessment

Fill in the information below about your work with JFLAP v8.0. If you cannot answer or feel that the statement does not apply to the exercise, select “N/A.”

1. Finite state accepters and regular grammars

<table>
<thead>
<tr>
<th>The new version of JFLAP…</th>
<th>Strongly Agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly Disagree</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>… makes it easier to construct a DFA.</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… makes clear the formal definition of a DFA and how the graphic representation of the structure relates to its definition</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… provides a simple and useful interface for presenting formal definitions and symbols in grammars and FA’s</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… makes it easier to understand how an FA relates to a regular grammar by directly translating state names into variables</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… improved the flexibility of the “Simulate Input” window, making this process more easy to explore and analyze</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

2. Pushdown automata and context-free grammars

<table>
<thead>
<tr>
<th>The new version of JFLAP…</th>
<th>Strongly Agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly Disagree</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>… makes the formal definition of a PDA very clear.</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… gives insight into how a PDA is converted to a CFG through showing the relationship between states, symbols, and variables</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… allows states and variables to have a more obvious purpose in defining a language and making language more clear</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

3. MouseCat grammar and literal languages

<table>
<thead>
<tr>
<th>The new version of JFLAP…</th>
<th>Strongly Agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly Disagree</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>… makes it easy and intuitive to construct custom alphabets with complex, multi-character symbols</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… gives constructive feedback while building the alphabets, explaining clearly what went wrong and how to fix it</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… allows the user to represent a literal language like the MouseCat grammar more clearly than the previous version</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… creates a bridge between the concepts of FLA theory and its actual applications through exploration of literal languages</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… is flexible and makes it easy to undo/redo edits to a grammar</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
<tr>
<td>… provides useful feedback in constructing alphabets and symbols</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
<td>O</td>
</tr>
</tbody>
</table>

a. Was the process of constructing a grammar from scratch easy and intuitive?

b. What features did you like?

c. What features did you not like or think should change?
4. **General Questions** – Please answer these in as much detail as possible!

<table>
<thead>
<tr>
<th>The new version of JFLAP…</th>
<th>Strongly Agree</th>
<th>Agree</th>
<th>Neutral</th>
<th>Disagree</th>
<th>Strongly Disagree</th>
<th>N/A</th>
</tr>
</thead>
<tbody>
<tr>
<td>… provides an environment identical to previous versions of JFLAP simply with more information available</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>… allows for the creation of more interesting languages</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>… provides a simple and useful interface for presenting formal definitions and symbols</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>… generally improves the user experience, making the material more engaging and relatable</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
<tr>
<td>… encourages more user interaction with each grammar or automaton through the formal definition interface</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
<td>○</td>
</tr>
</tbody>
</table>

1. Compared to your experiences with the previous version, do you feel like the additions in JFLAP v8.0 would help you understand FLA theory better?

2. Would you prefer to work with symbolic languages (as in exercises 1 & 2) or literal languages (as in exercise 3) while learning about and exploring FLA theory?

3. Of what you saw in JFLAP v8.0, what addition(s) do you like the most? Explain.

4. What do you like the least? Explain.

5. Provide any other feedback – Features you would like to see added to JFLAP, helpfulness of the interactions with the formal definition, the value error/warning messages, etc.

Thank you for helping provide feedback for the new version of JFLAP!

If you have more comments or would like to be involved with the testing process as it continues, feel free to email Professor Rodger (rodger@cs.duke.edu) so that she can put you in touch with a developer.

Genkins 48
# A3. Survey Results

## Table 1: Individual Participant Responses

The table below shows the individual participants responses to the survey and assessments of JFLAP v8.0. As discussed in section 10 of the paper, each category on the Likert scale was given an associated value, and those values are what were input into this table (see 10.2 for more details). If the cell is blacked out, that means that the participant either did not fill in the bubble or selected the N/A option.

<table>
<thead>
<tr>
<th>... makes it easier to construct a DFA.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
<th>10</th>
<th>11</th>
<th>12</th>
</tr>
</thead>
<tbody>
<tr>
<td>... makes clear the formal definition of a DFA and how the graphic representation of the structure relates to its definition</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>... provides a simple and useful interface for presenting formal definitions and symbols in grammars and FA’s</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... makes it easier to understand how an FA relates to a regular grammar by directly translating state names into variables</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... improved the flexibility of the “Simulate Input” window, making this process more easy to explore and analyze</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>... makes the formal definition of a PDA very clear.</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... gives insight into how a PDA is converted to a CFG through showing the relationship between states, symbols, and variables</td>
<td>2</td>
<td>-1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... allows states and variables to have a more obvious purpose in defining a language and making language more clear</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... makes it easy and intuitive to construct custom alphabets with complex, multi-character symbols</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>... gives constructive feedback while building the alphabets, explaining clear what went wrong and how to fix it</td>
<td>1</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... allows the user to represent a literal language like the MouseCat grammar more clearly than the previous version</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>... creates a bridge between the concepts of FLA theory and its actual applications through exploration of literal languages</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>-1</td>
</tr>
<tr>
<td>... is flexible and makes it easy to undo/redo edits to a grammar</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>... provides useful feedback in constructing alphabets and symbols</td>
<td>2</td>
<td>0</td>
<td>-1</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>... provides an environment identical to previous versions of JFLAP simply with more information available</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>-1</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... allows for the creation of more interesting languages</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... provides a simple and useful interface for presenting formal definitions and symbols</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>... generally improves the user experience, making the material more engaging and relatable</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>... encourages more user interaction with each grammar or automaton through the formal definition interface</td>
<td>2</td>
<td>1</td>
<td>-1</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>
### Table 2: Tabulations

The table below shows the aggregate sum, mean score, and individual score counts for each assessment statement. If a cell is highlighted in yellow (and associated sum is bold), then that claim was nearly universally agreed upon – aggregate sum ≥ 15. Green highlighted averages indicate all averages over 1, i.e. every situation in which the average student agreed or strongly agreed with the statement. Note that the strongly disagreed column was removed as that was not a single instance of a strong-disagreeing selection.

<table>
<thead>
<tr>
<th>_statement</th>
<th>SUM</th>
<th>MEAN</th>
<th>STRONGLY AGREE</th>
<th>AGREE</th>
<th>NEUTRAL</th>
<th>DISAGREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>... makes it easier to construct a DFA.</td>
<td>10</td>
<td>0.91</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>... makes clear the formal definition of a DFA and how the graphic representation of the structure relates to its definition</td>
<td>10</td>
<td>0.91</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>... provides a simple and useful interface for presenting formal definitions and symbols in grammars and FA’s</td>
<td>13</td>
<td>1.18</td>
<td>3</td>
<td>7</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... makes it easier to understand how an FA relates to a regular grammar by directly translating state names into variables</td>
<td>15</td>
<td>1.36</td>
<td>5</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... improved the flexibility of the “Simulate Input” window, making this process more easy to explore and analyze</td>
<td>12</td>
<td>1.09</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>... makes the formal definition of a PDA very clear.</td>
<td>10</td>
<td>0.83</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>... gives insight into how a PDA is converted to a CFG through showing the relationship between states, symbols, and variables</td>
<td>8</td>
<td>0.67</td>
<td>3</td>
<td>4</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>... allows states and variables to have a more obvious purpose in defining a language and making language more clear</td>
<td>11</td>
<td>0.92</td>
<td>3</td>
<td>6</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>... makes it easy and intuitive to construct custom alphabets with complex, multi-character symbols</td>
<td>16</td>
<td>1.33</td>
<td>6</td>
<td>4</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>... gives constructive feedback while building the alphabets, explaining clear what went wrong and how to fix it</td>
<td>6</td>
<td>0.50</td>
<td>1</td>
<td>5</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>... allows the user to represent a literal language like the MouseCat grammar more clearly than the previous version</td>
<td>17</td>
<td>1.42</td>
<td>6</td>
<td>5</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>... creates a bridge between the concepts of FLA theory and its actual applications through exploration of literal languages</td>
<td>13</td>
<td>1.08</td>
<td>3</td>
<td>8</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>... is flexible and makes it easy to undo/redo edits to a grammar</td>
<td>13</td>
<td>1.08</td>
<td>5</td>
<td>3</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>... provides useful feedback in constructing alphabets and symbols</td>
<td>9</td>
<td>0.75</td>
<td>4</td>
<td>2</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>... provides an environment identical to previous versions of JFLAP simply with more information available</td>
<td>5</td>
<td>0.50</td>
<td>1</td>
<td>5</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>... allows for the creation of more interesting languages</td>
<td>10</td>
<td>1.00</td>
<td>5</td>
<td>1</td>
<td>3</td>
<td>1</td>
</tr>
</tbody>
</table>