CRASH COURSE: JFLAP

by

Paul J. Kaiser

First Paperback Edition 2021
ISBN: XXXXXXXXXX

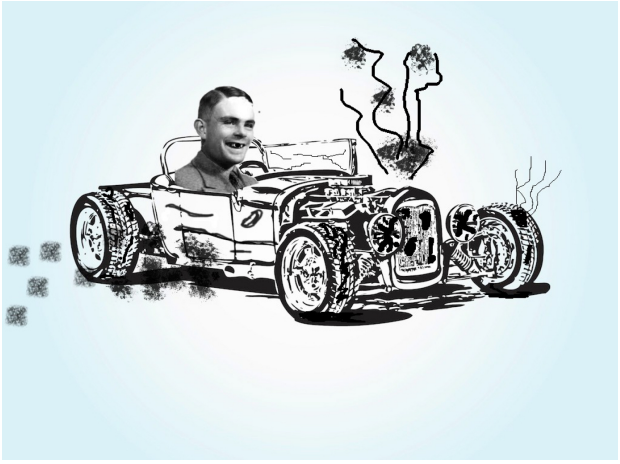
Printed in the United States of America

Please visit my personal website at:
http://www.cs.lewisu.edu/ kaiserpa/

Contents

Chapter 1

Introduction



1.1 Turing Machines

Definition 1.1.1. A **Turing Machine** M is a 7-tuple (Q, Σ, Γ, δ, q_0, Q_{accept}, Q_{reject}) where Q, Σ, and Γ are all finite sets and

1. Q is a set of **states**
2. Σ is the **input alphabet** (non-blank symbols)
   a blank symbol will be represented with a □op
3. Γ is the **tape alphabet**
   $\square \in \Gamma, 0 \in \Gamma, 1 \in \Gamma$, and $\Sigma \subseteq \Gamma$men
4. δ : Q x Γ → Q x Γ x {L,R,S}, a **transition function**
   i.e., δ (q,s) = (q',s',m) where
   q' = next state
   s' = write symbol (possibly a □)
   m = left / right / stationary
5. q_0 is a unique **start state**
6. $Q_{accept} \subseteq$ Q, a set of **accept states** (may be empty $\emptyset$)
7. $Q_{reject} \subseteq$ Q, a set of **reject states** (may be empty $\emptyset$)

Turing Machines are the most powerful of all the finite state machines ... and ... they were the first to be defined. So this brief tutorial begins at the deep end of the pool! We will shortly begin with the most basic example of a Turing Machine – a deterministic single-tape Turing Machine.

1.2 JFLAP

For our study of finite state machines we will use JFLAP software. It may be downloaded from the following URL:

www.jflap.org

As its website states: "JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing Machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar."

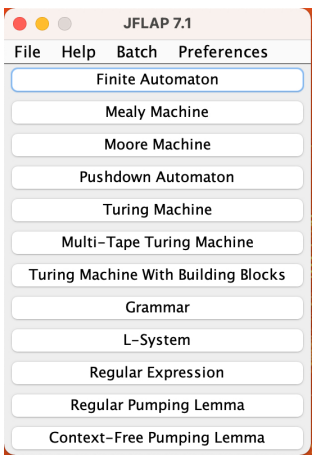
A good tutorial may be found on-line at: ***www.jflap.org/tutorial***

A good textbook may be found at: ***amazon.com***

> JFLAP: An Interactive Formal Languages
> and Automata Package
> by Susan H. Rodger and Thomas W. Finley
> Jones & Bartlet Publishers, Sudbury, MA
> ISBN 076378344

The following command will bring on screen the basic JFLAP menu:

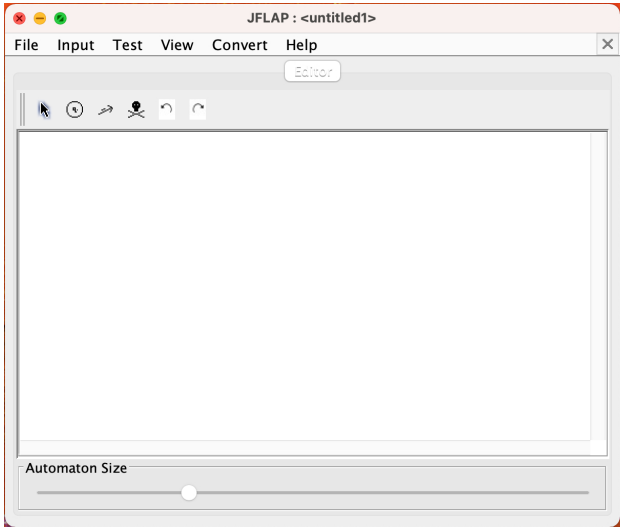
$ java -jar JFLAP7.1.jap



During this crash course in JFLAP, we sill concentrate primarily on the three entries:

- Finite Automaton
- Pushdown Automaton
- Turing Machine

If you click on the "Turing Machine" entry, you move into the **editor mode** window:



The **editor mode** window is very traditional in its organization – a list of pop-up sub-menus across the top and a menu bar of icons beneath that.

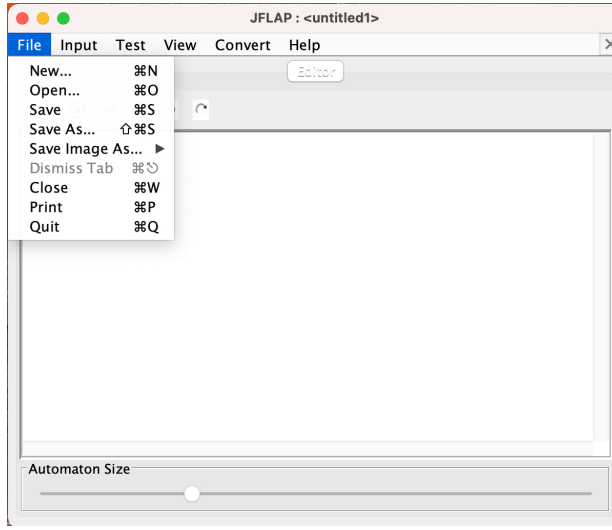
The **list of pop-up sub-menus** includes:

- File
- Input
- Test
- View
- Convert
- Help

The **menu bar icons** are (in order from left to right):

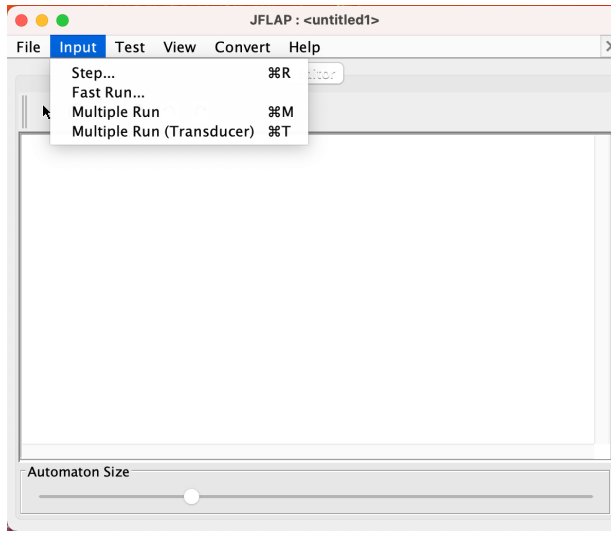
- attribute editor – initial state, final state, ...
- state creator – create individual states $q_0, q_1, \ldots$
- transition creator – defines transition function elements
  - current symbol
  - print symbol
  - move: L, R, S
- deleter – erases components of the Turing Machine
- undoer – undo the previous step
- redoer – redo the previous step

The "File" sub-menu is quite typical of "File" sub-menus in most applications!



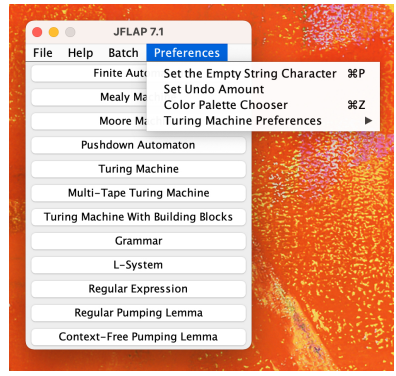
The "Input" sub-menu is used to operate the Turing Machine that is defined in **edit** mode:

- Step allows you to see the Turing Machine in action both with and without data on the input tape
- Fast Run is useful for studying Turing Machines which generate no output other than ACCEPT / REJECT
- Multiple Run allows you to see the end result of running the Turing Machine on multiple items of data on the input tape



1.3 Customizing JFLAP

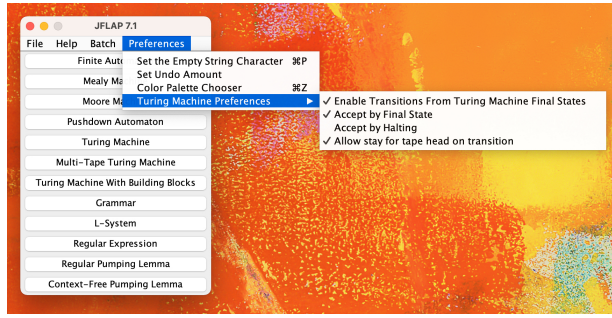
The "Preferences" tab on the top of the initial JFLAP screen allows the user to customize the appearance and the features available within the Turing Machines we create.



"Set the EmptyString Character" gives you two choices: the symbol λ or the more traditional ϵ.

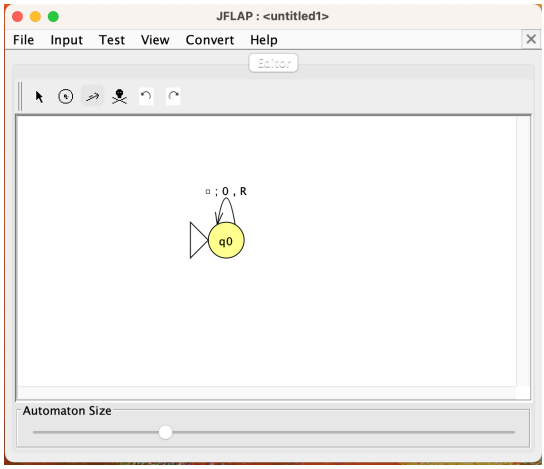
"Turing Machine Preferences" provides four options:

- allow transitions out of final states (recommend YES)
- ACCEPT by final state (recommend YES)
- ACCEPT by halting (recommend NO)
- allow STAY for tape head movement (recommend YES)

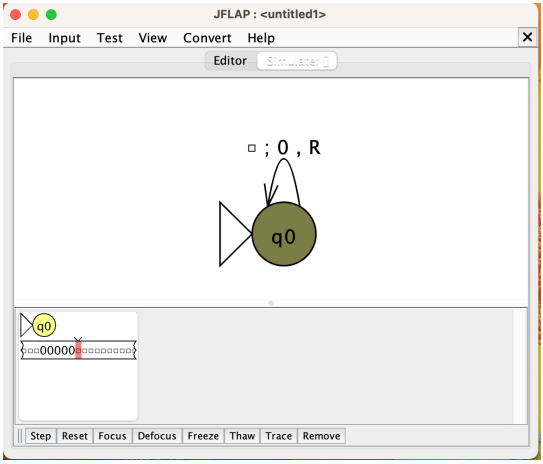


1.4 Computable Numbers

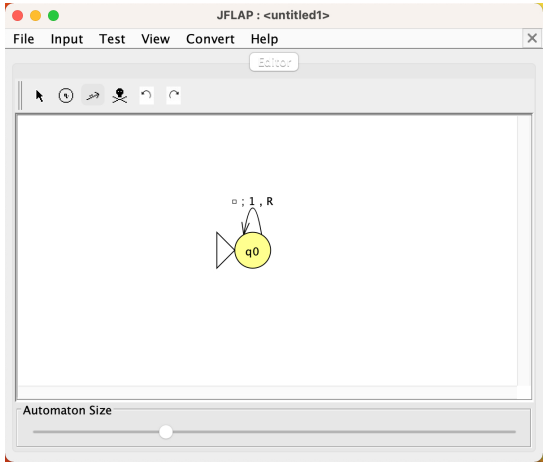
x = 0 = 0.000000 ...



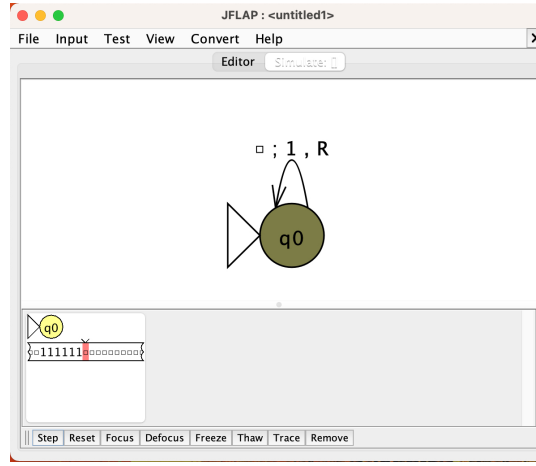
The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.



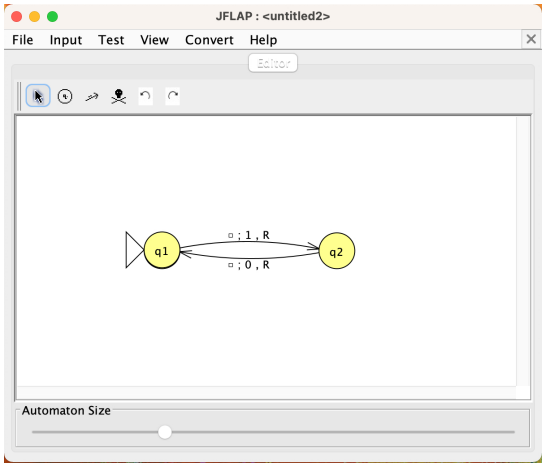
$\mathbf{x = 1 = 0.111111\ldots}$



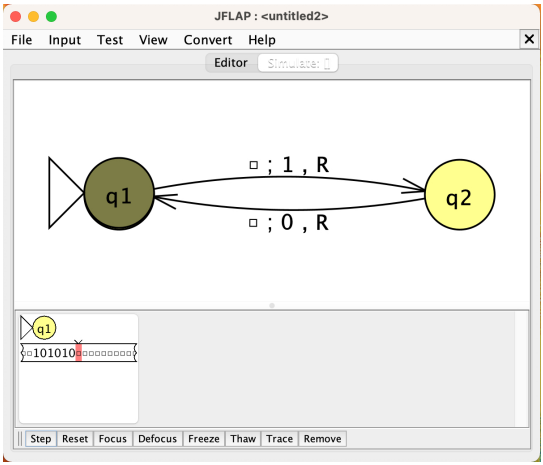
The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.



x = 2/3 = 0.101010 ...

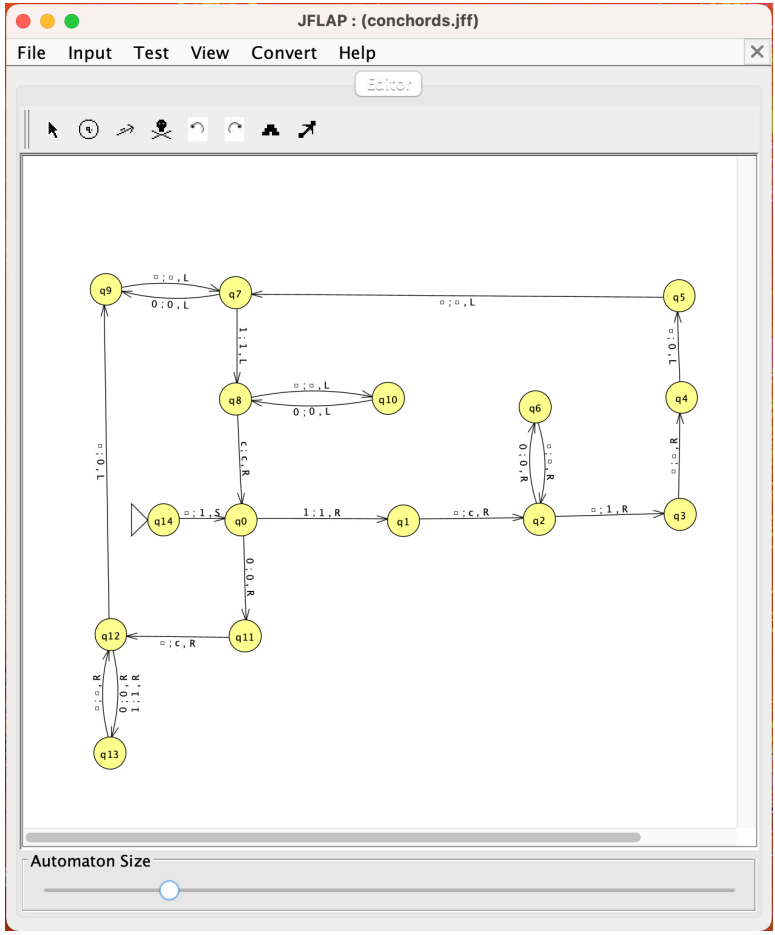


The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.

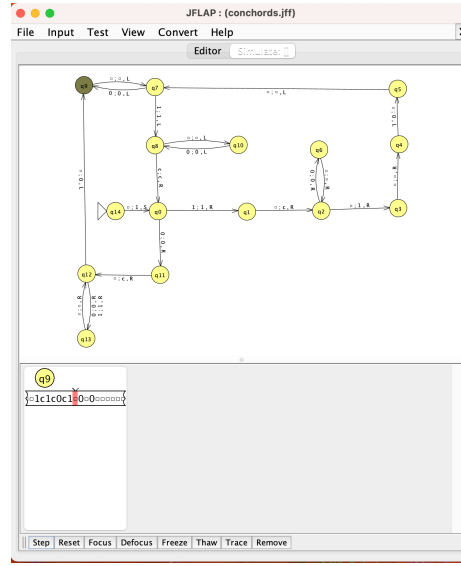


The following is a truly ***irrational number*** – a binary expansion which does not terminate and does not repeat. The number of zeros found between consecutive ones increases by one each pattern!

x = 0.11010010001000010000010000001000000001 ...



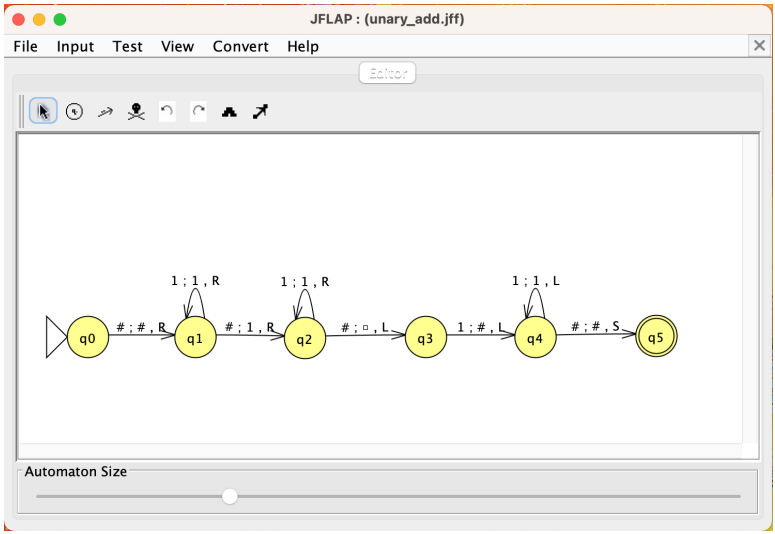
The image below right illustrates using the "Input" popu-up menu to "step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.


JFLAP : (conchords.jff)
File Input Test View Convert Help
Editor
q9
Step Reset Focus Defocus Freeze Thaw Trace Remove

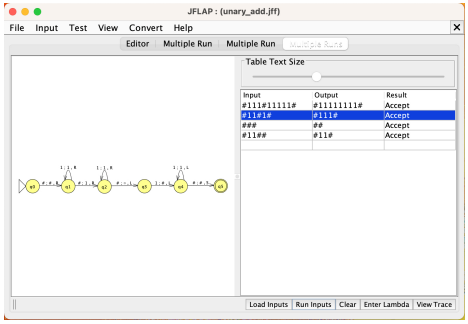
1.5 Turing Machine as CALCULATOR

This section illustrates the ability of a Turing Machine to perform basic calculations, such as addition, subtraction, multiplication, and division. We demonstrate two implementations: the first for unary addition, the second for binary addition.

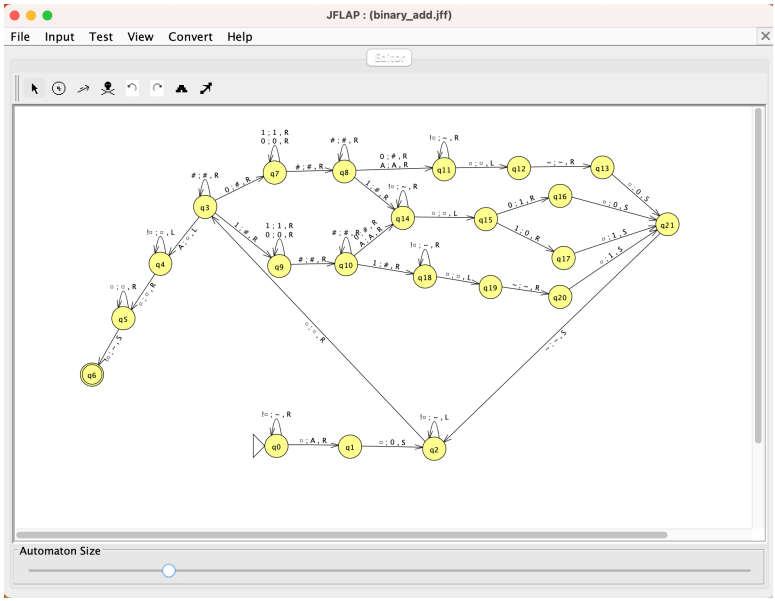
unary addition



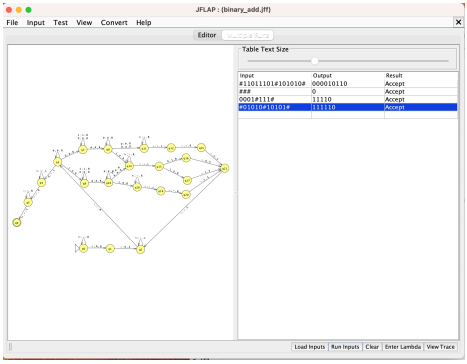
The image below right illustrates using the "Input" popu-up menu to "Multiple Run (Transducer)" not "Step" process several different input strings at one time. The "Transducer" option generates the output strings.



binary addition



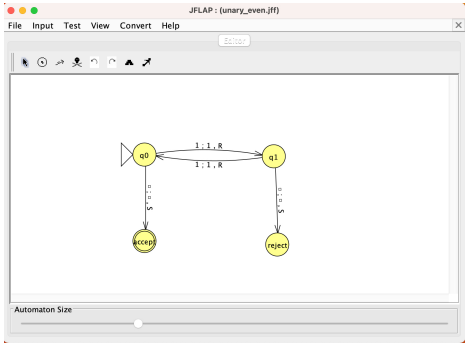
The image below right illustrates using the "Input" popu-up menu to "Multiple Run (Transducer)" not "Step" process several different input strings at one time. The "Transducer" option generates the output strings.



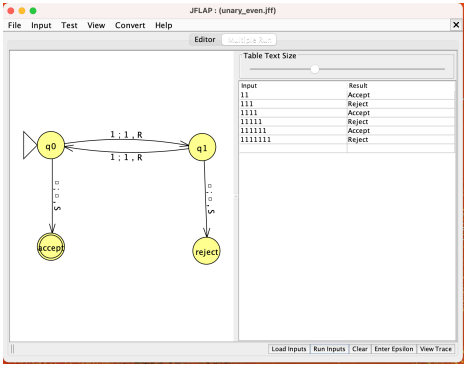
1.6 Turing Machine as ACCEPTOR

This section illustrates the ability of a Turing Machine to serve as a recognizer or acceptor for questions that ultimately result in a YES/NO or ACCEPT/REJECT decision. We demonstrate two implementations: the first for recognizing an even number in unary representation, the second for recognizing an even number in binary representation.

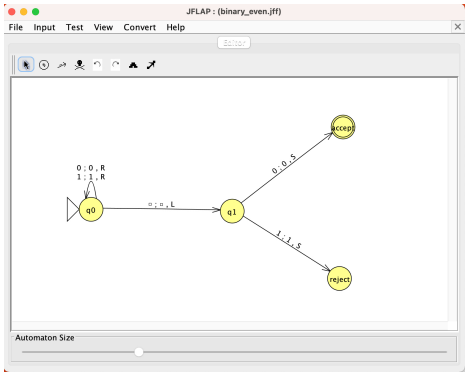
unary even



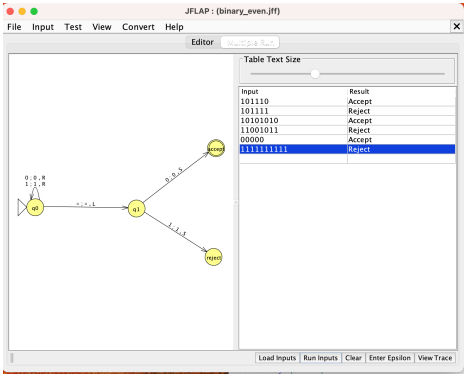
The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.



binary even

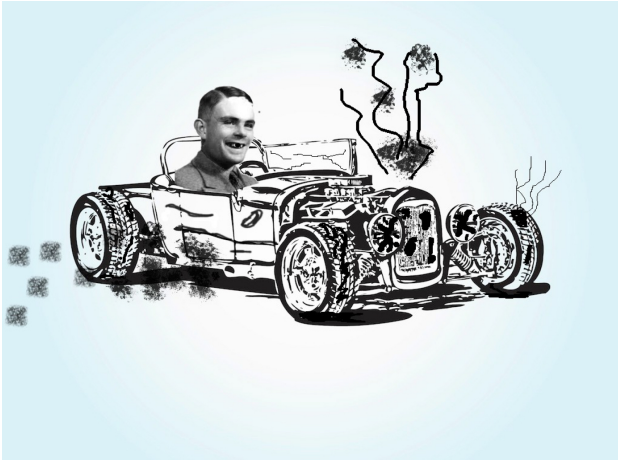


The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.



Chapter 2

Introduction



2.1 Turing Machines

Definition 2.1.1. A **Turing Machine** M is a 7-tuple (Q, Σ, Γ, δ, q_0, Q_{accept}, Q_{reject}) where Q, Σ, and Γ are all finite sets and

1. Q is a set of **states**
2. Σ is the **input alphabet** (non-blank symbols)
   a blank symbol will be represented with a □op
3. Γ is the **tape alphabet**
   $\square \in \Gamma, 0 \in \Gamma, 1 \in \Gamma$, and $\Sigma \subseteq \Gamma$men
4. δ : Q x Γ → Q x Γ x {L,R,S}, a **transition function**
   i.e., δ (q,s) = (q',s',m) where
   q' = next state
   s' = write symbol (possibly a □)
   m = left / right / stationary
5. q_0 is a unique **start state**
6. $Q_{accept} \subseteq Q$, a set of **accept states** (may be empty ∅)
7. $Q_{reject} \subseteq Q$, a set of **reject states** (may be empty ∅)

Turing Machines are the most powerful of all the finite state machines ... and ... they were the first to be defined. So this brief tutorial begins at the deep end of the pool! We will shortly begin with the most basic example of a Turing Machine – a deterministic single-tape Turing Machine.

2.2 JFLAP

For our study of finite state machines we will use JFLAP software. It may be downloaded from the following URL:

www.jflap.org

As its website states: "JFLAP is software for experimenting with formal languages topics including nondeterministic finite automata, nondeterministic pushdown automata, multi-tape Turing Machines, several types of grammars, parsing, and L-systems. In addition to constructing and testing examples for these, JFLAP allows one to experiment with construction proofs from one form to another, such as converting an NFA to a DFA to a minimal state DFA to a regular expression or regular grammar."

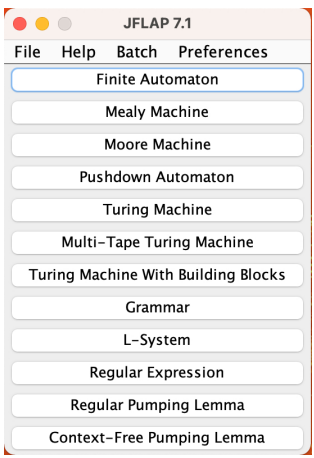
A good tutorial may be found on-line at: ***www.jflap.org/tutorial***

A good textbook may be found at: ***amazon.com***

> JFLAP: An Interactive Formal Languages
> and Automata Package
> by Susan H. Rodger and Thomas W. Finley
> Jones & Bartlet Publishers, Sudbury, MA
> ISBN 076378344

The following command will bring on screen the basic JFLAP menu:

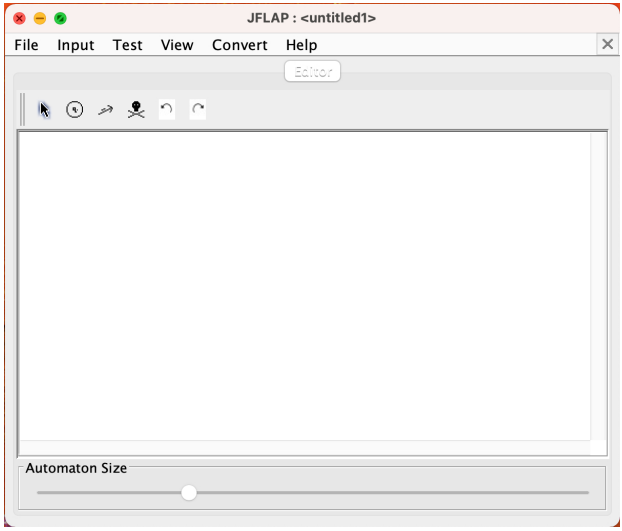
$ java -jar JFLAP7.1.jap



During this crash course in JFLAP, we sill concentrate primarily on the three entries:

- Finite Automaton
- Pushdown Automaton
- Turing Machine

If you click on the "Turing Machine" entry, you move into the **editor mode** window:



The **editor mode** window is very traditional in its organization – a list of pop-up sub-menus across the top and a menu bar of icons beneath that.

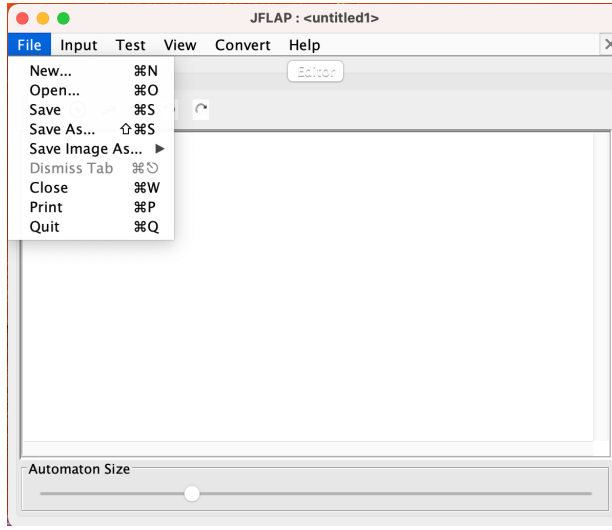
The **list of pop-up sub-menus** includes:

- File
- Input
- Test
- View
- Convert
- Help

The **menu bar icons** are (in order from left to right):

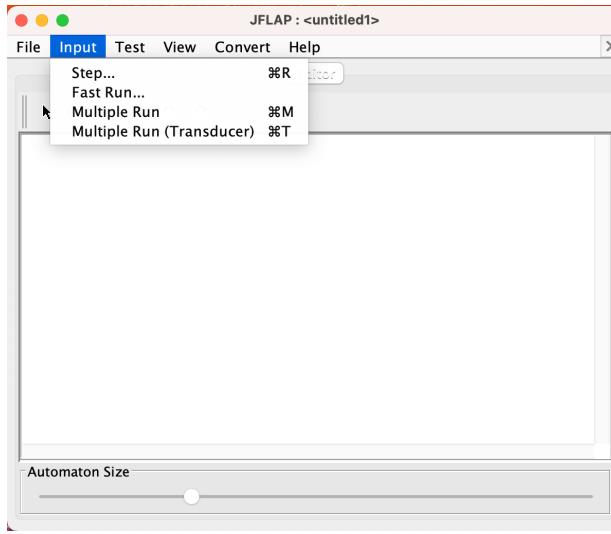
- attribute editor – initial state, final state, ...
- state creator – create individual states $q_0, q_1, \ldots$
- transition creator – defines transition function elements
  - current symbol
  - print symbol
  - move: L, R, S
- deleter – erases components of the Turing Machine
- undoer – undo the previous step
- redoer – redo the previous step

The "File" sub-menu is quite typical of "File" sub-menus in most applications!



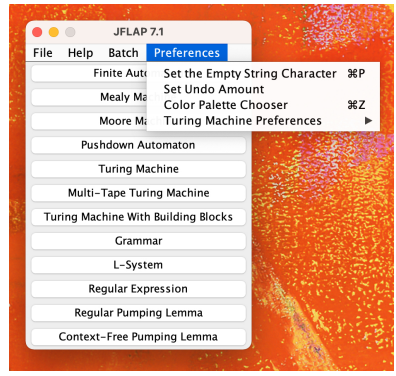
The "Input" sub-menu is used to operate the Turing Machine that is defined in **edit** mode:

- Step allows you to see the Turing Machine in action both with and without data on the input tape
- Fast Run is useful for studying Turing Machines which generate no output other than ACCEPT / REJECT
- Multiple Run allows you to see the end result of running the Turing Machine on multiple items of data on the input tape



2.3 Customizing JFLAP

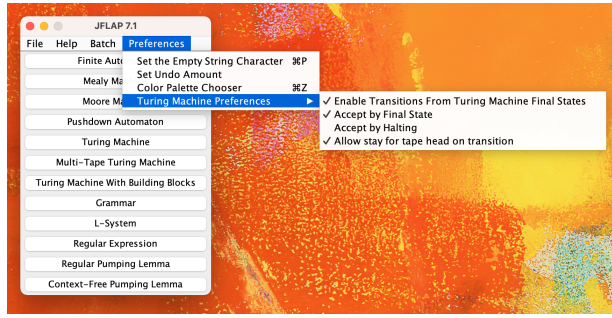
The "Preferences" tab on the top of the initial JFLAP screen allows the user to customize the appearance and the features available within the Turing Machines we create.



"Set the EmptyString Character" gives you two choices: the symbol λ or the more traditional ϵ.

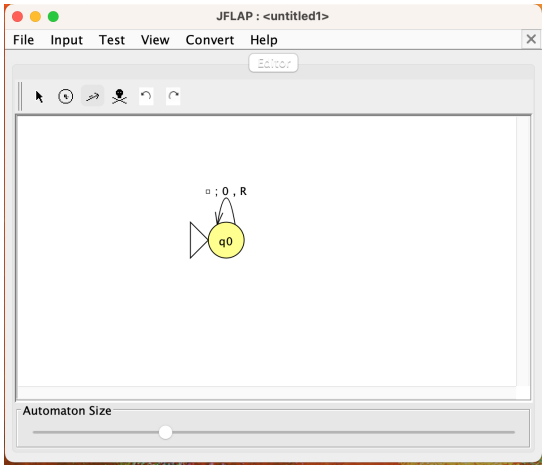
"Turing Machine Preferences" provides four options:

- allow transitions out of final states (recommend YES)
- ACCEPT by final state (recommend YES)
- ACCEPT by halting (recommend NO)
- allow STAY for tape head movement (recommend YES)

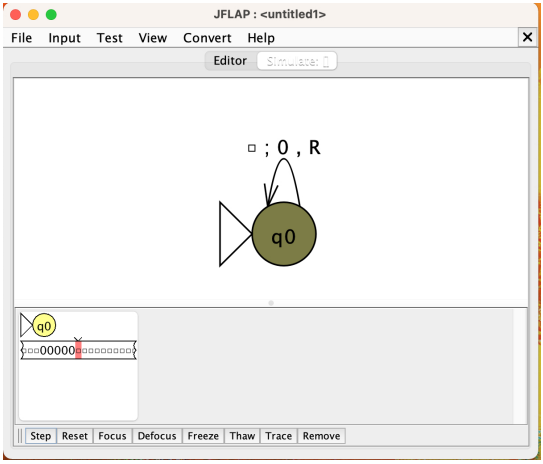


2.4 Computable Numbers

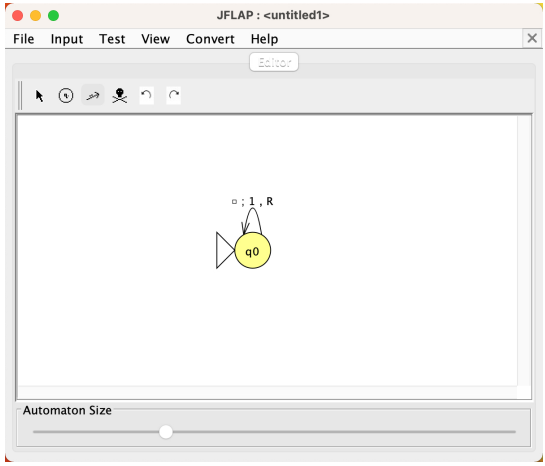
x = 0 = 0.000000 ...



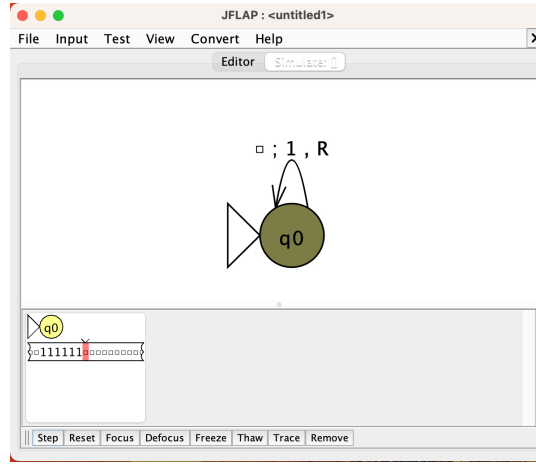
The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.



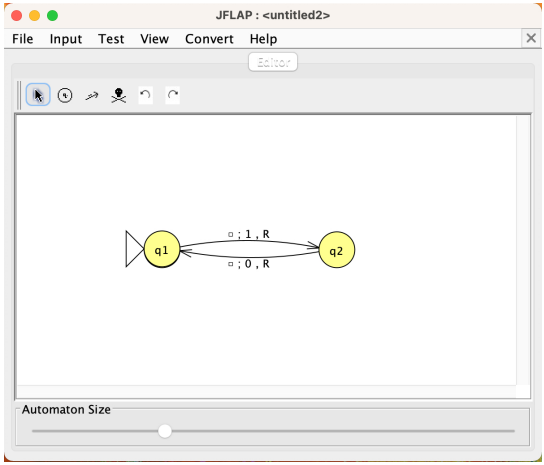
$\mathbf{x = 1 = 0.111111\ldots}$



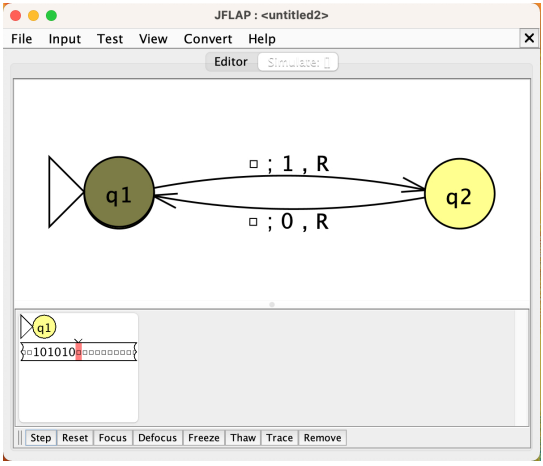
The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.



x = 2/3 = 0.101010 ...

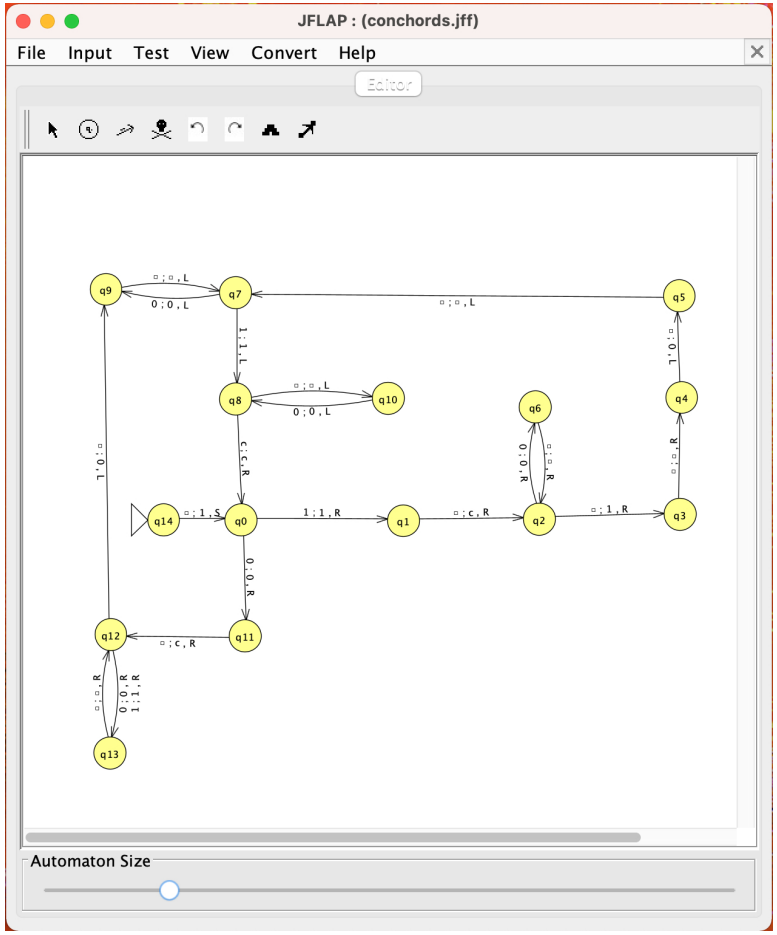


The image below right illustrates using the "Input" popu-up menu to "Step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.

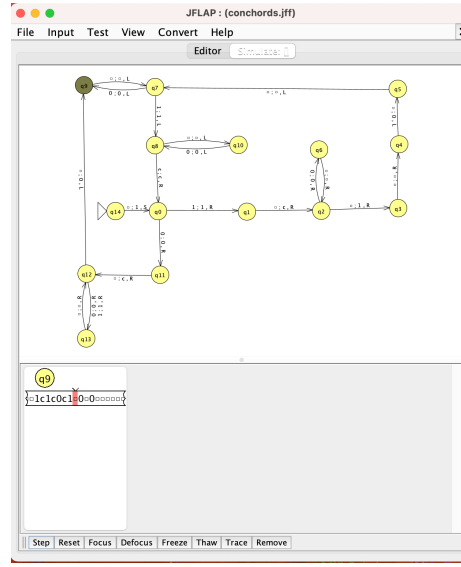


The following is a truly ***irrational number*** – a binary expansion which does not terminate and does not repeat. The number of zeros found between consecutive ones increases by one each pattern!

x = 0.11010010001000010000010000010000001 ...



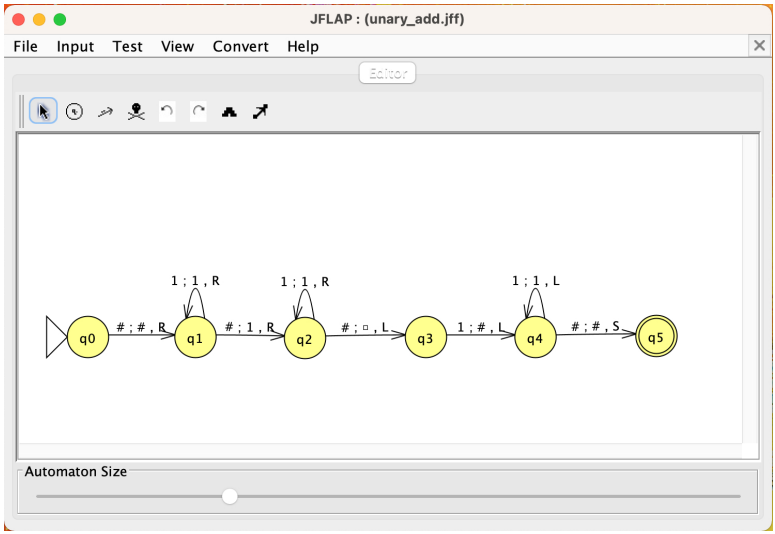
The image below right illustrates using the "Input" popu-up menu to "step" through the Turing Machine generating the computable number. Input to the Turing Machine is the empty string.


JFLAP : (conchords.jff)
File Input Test View Convert Help
Editor
q9
Step Reset Focus Defocus Freeze Thaw Trace Remove

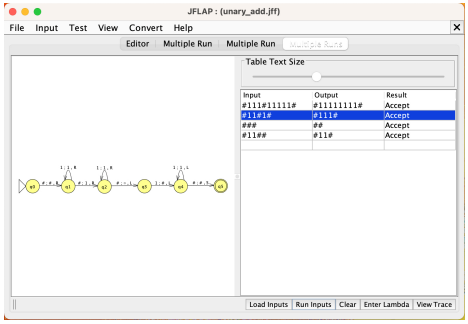
2.5 Turing Machine as CALCULATOR

This section illustrates the ability of a Turing Machine to perform basic calculations, such as addition, subtraction, multiplication, and division. We demonstrate two implementations: the first for unary addition, the second for binary addition.

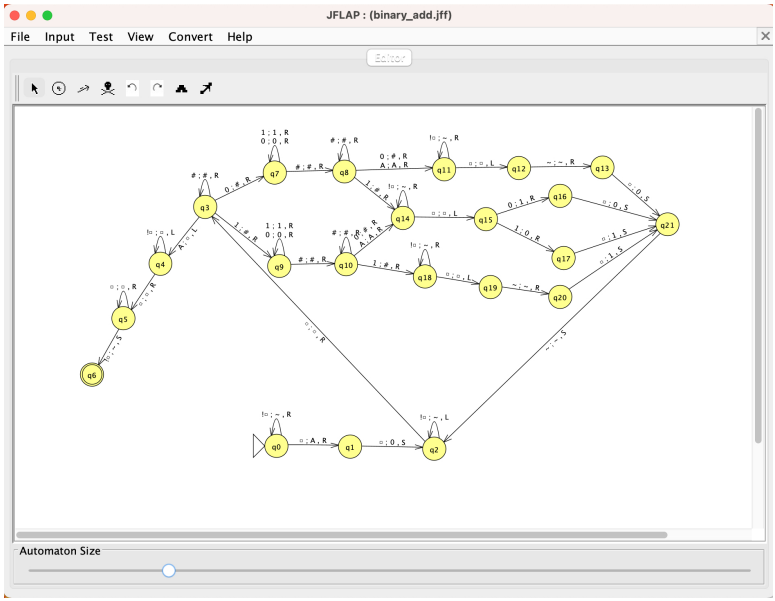
unary addition



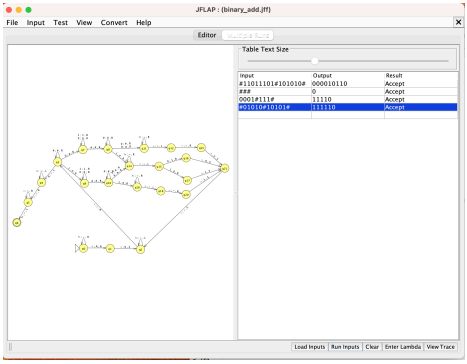
The image below right illustrates using the "Input" popu-up menu to "Multiple Run (Transducer)" not "Step" process several different input strings at one time. The "Transducer" option generates the output strings.



binary addition



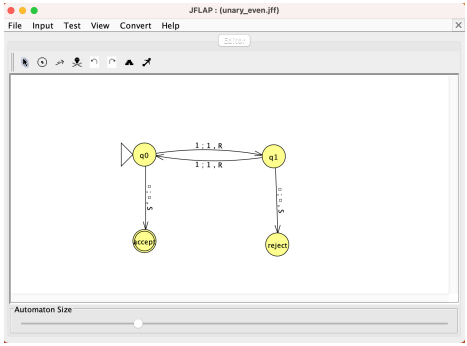
The image below right illustrates using the "Input" popu-up menu to "Multiple Run (Transducer)" not "Step" process several different input strings at one time. The "Transducer" option generates the output strings.



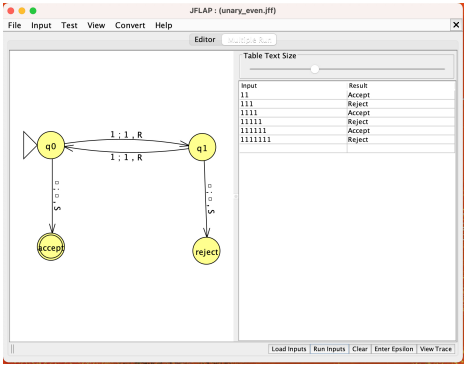
2.6 Turing Machine as ACCEPTOR

This section illustrates the ability of a Turing Machine to serve as a recognizer or acceptor for questions that ultimately result in a YES/NO or ACCEPT/REJECT decision. We demonstrate two implementations: the first for recognizing an even number in unary representation, the second for recognizing an even number in binary representation.

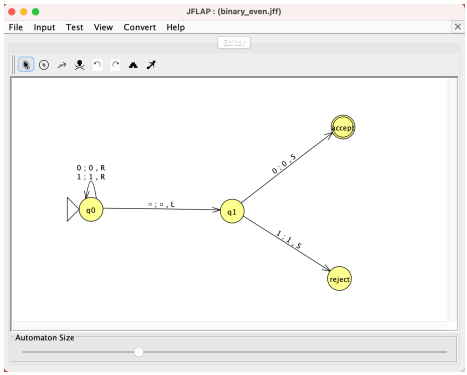
unary even



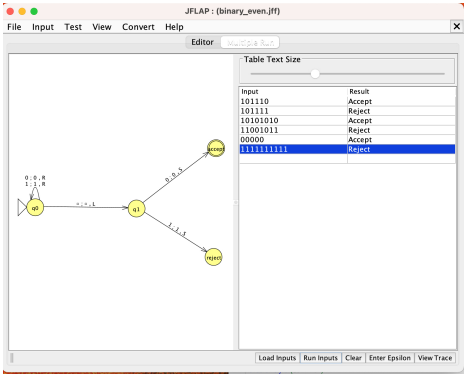
The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.



binary even

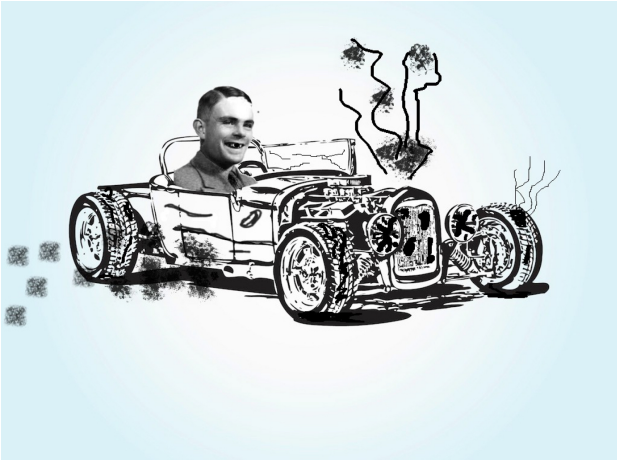


The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.



Chapter 3

Finite Automata



3.1 Definitions

Turing Machines are the foundation for all finite state machines. The rules defining allowable transitions between states determine the specific category of the machine: finite automata, push down automata, or general Turing Machine.

The first (and simplest) variation on a Turing Machine is a **finite automaton**. A finite automaton is a Turing Machine with:

- its tape is used for input only (reading the tape sequentially in a left-to-right order)
- no output is generated
- all transitions are defined by the current state and the current input symbol
- all undefined transitions cause the machine to stop otherwise machine stops when no more input is available
- input strings are either ACCEPTED or REJECTED

Definition 3.1.1. A **finite automaton** M is a 5-tuple (Q, Σ, δ, q_0, F) where Q and Σ are both finite sets and

1. Q is a set of **states**
2. Σ is a set of **symbols** (alphabet)
3. δ is a **transition function**
   δ (q , s) = q_{next}
4. q_0 is a unique **start state**, $q_0 \in Q$
5. $F \subseteq Q$ is a set of **accept states**

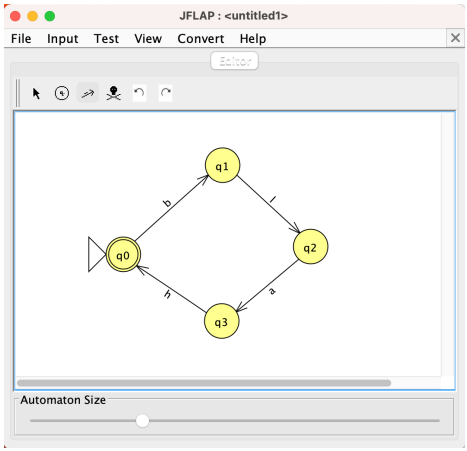
Definition 3.1.2. The language **recognized** by a finite automaton M is the collection

$$L(M) = \{ \text{ words } \omega \mid M \text{ accepts } \omega \}$$

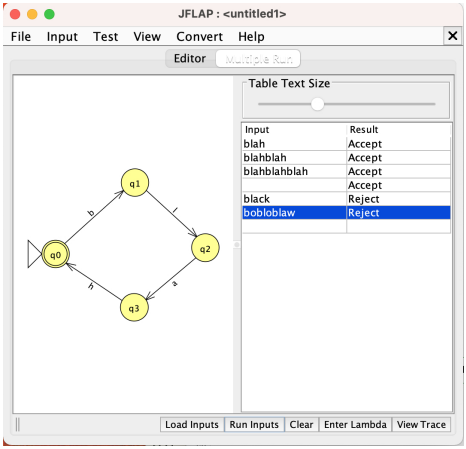
3.2 Examples

Our illustrative example of a finite automata will be the following language L built up from repeated use of the word **blah**.

L = { ϵ , blah , blahblah , blahblahblah , ... }



The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.



Comment. The previous example is of a **deterministic** finite automaton. Observe that every character identifies a unique transition (or STOPS) within the finite automaton. There is no possibility of choice or finding alternate sequences of states.

It is at this point that we bring **nondeterminism** into the discussion. Nondeterminism allows the transitions to includ the possibility of choice or alternate transitions into the process.

Examples

δ (q , a) = q' two different "next" states
δ (q , a) = q" possible from single input character

δ (a , ϵ) = q' move to "next"state
without considering the input character

All the definitions appearing two pages earlier may be modified to incorporate nondeterminism. Hence there are two flavors of finite automata:

- DFA deterministic finite automata
- NFA nondeterministic finite automata

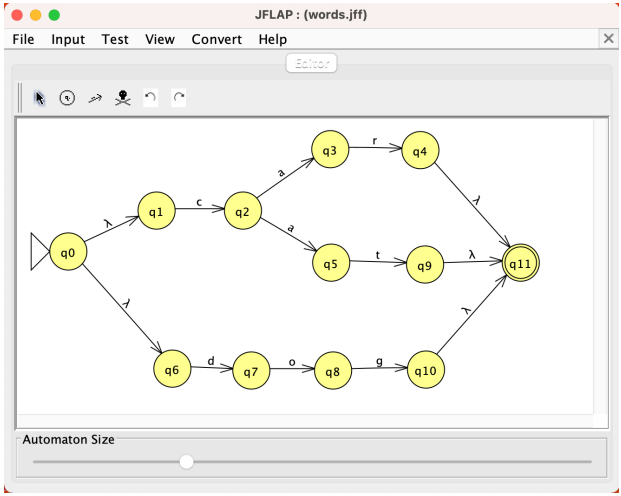
Our second illustrative example of a finite automata will be the following language L built up from three simple words **car**, **cat**, and **dog**.

L = { car , cat , dog }

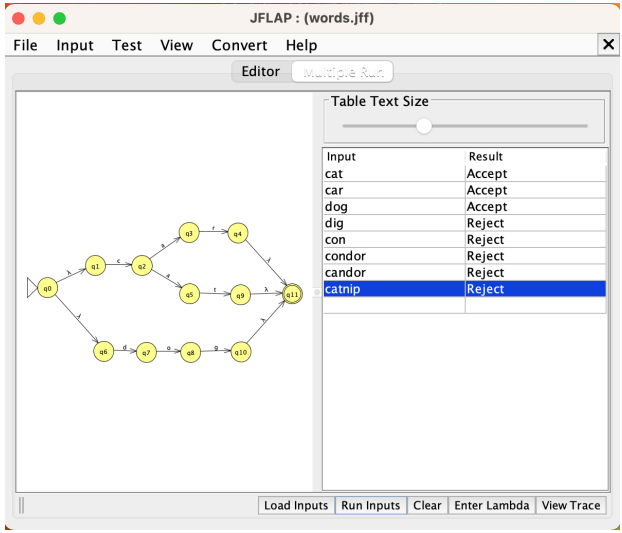
Our first examples only needed a deterministic finite automaton because we were not really confronted with any choices – we simply had to recognize when the **blah**s stopped.

This second example confronts us with nondeterminism – is the first letter a 'c' or a 'd'? can an ϵ-transition allow me to easily connect possible sequences of states?

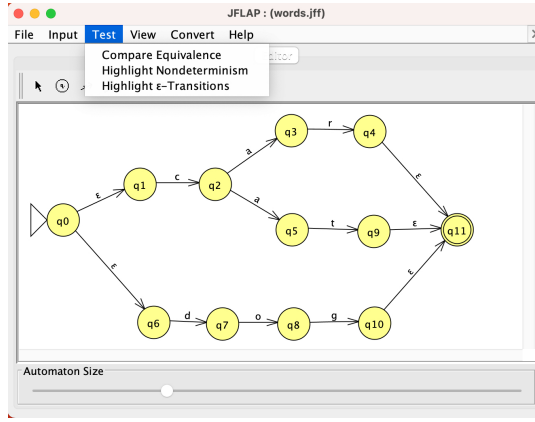
L = { car , cat , dog }

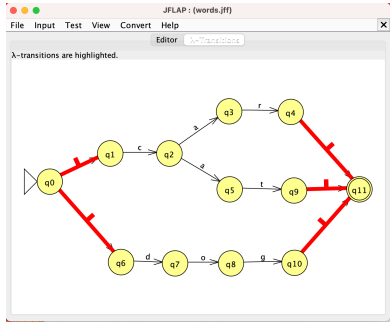


The image below right illustrates using the "Input" popu-up menu to "Multiple Run" not "Step" process several different input strings at one time.

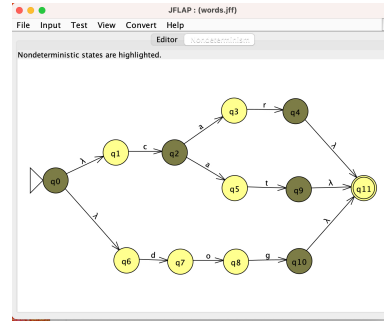


Using JFLAP we can easily determine whether any nondeterminism may be found in a given finite automaton using the "Test" sub-menu as illustrated below





The image above left illustrates using the "Test" popu-up menu to "Highlight ϵ-transitions". The image below right illustrates using the "Test" pop-up menu to "Highlight Nondeterminism".

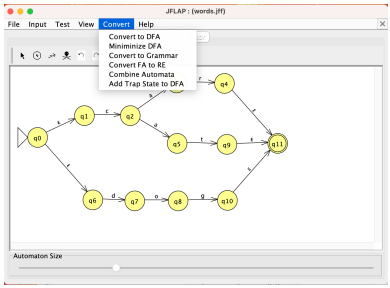


The addition of the concept of nondeterminism to our discussion of finite automata would appear to significantly increase the capability for such machines. However, in the case of finite automata there is no such increase!

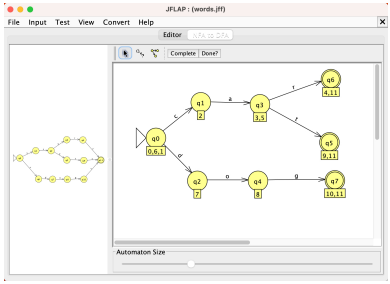
Theorem 3.2.1. *If M is a deterministic finite automaton, then there exists a deterministic finite automaton M' such that L(M) = L(M').*

Since a deterministic finite automaton M is also a nondeterministic finite automaton (with options limited to a single choice!), the capability of both types are identical.

JFAP has a very useful tool for converting a nondeterministic finite automaton to a deterministic finite automaton.



The "Convert" po-up menu provides the option "Convert to DFA". By clicking on this option a new window appears with two buttons: "Complete?" and "Done?" Click on "Complete?" and something like the image below will appear. Unfortunately, the new image might look like a tangled ball of yarn! The image below had to be rearranged using the attribute editor.



3.3 Regular Languages

Definition 3.3.1. A **regular language** is a language L which is recognized by a finite automaton M, i.e.,

$$L = L\ (M)$$

Chapter 4

Regular Expressions

4.1 Definitions

Definition 4.1.1. A **regular expression** R describing the words in a language L is defined inductively:

$\emptyset$ is a regular expression	the empty language
ϵ is a regular expression	the empty string
$s \in \Sigma$ is a regular expression	any single character
if R_1 and R_2 are regular expressions	then so is $R_1 \cup R_2$
if R_1 and R_2 are regular expressions	then so is $R_1 \circ R_2$
if R is a regular expression	then so is R^*

Notation

The notation for regular expressions may take several forms

$(a \cup b \cup c) \circ b^* \circ d \circ e$	traditional
$(a + b + c) \circ b^* \circ d \circ e$	arithmetic
$\{ a , b , c \}\ b^*\ d\ e$	implied concatenation

Remember:

$\emptyset$ = empty language $\Rightarrow$ no words
ϵ = empty string $\Rightarrow$ no characters, but is a word

so

$a \cup \emptyset = a$
$a \circ \emptyset = \emptyset$
$\emptyset^* = \epsilon$
$a \cup \epsilon = \{ a , \epsilon \}$
$a \circ \epsilon = a$
$\epsilon^* = \epsilon$

Definition 4.1.2. If R is a regular expression, then the language **generated** by R is

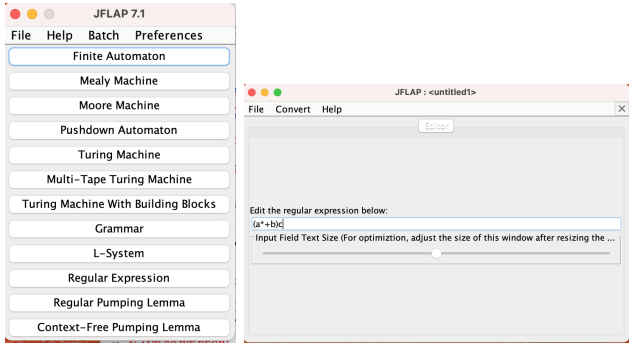
$L = L(R) = \{$ words $\omega \mid \omega$ matches the pattern defined by R $\}$

4.2 Examples

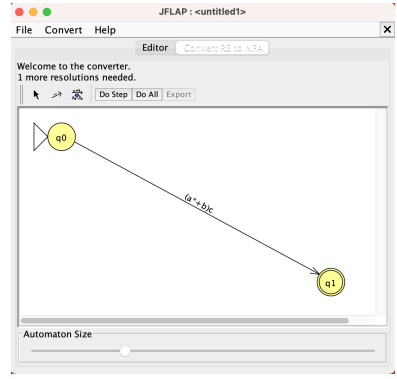
Regular Expressions and Finite Automata are equivalent in expressive power for regular languages. As would be expected, JFLAP has builtin capability to convert between the two representations.

Regular Expression to FA

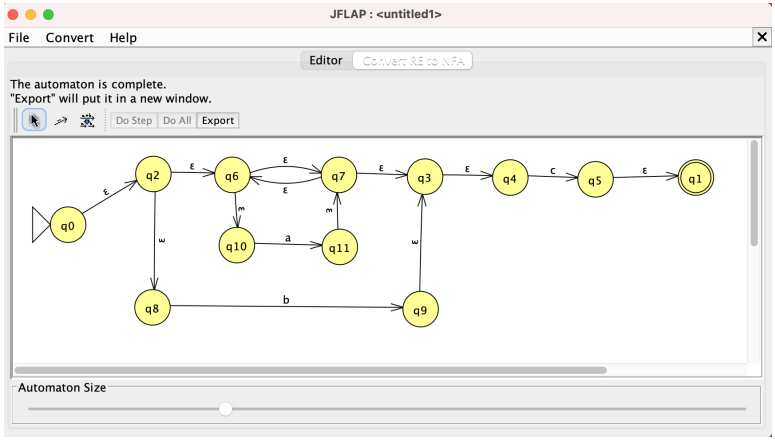
From the JFLAP main menu, select **Regular Expression**. You will then be prompted to enter a regular expression.



The "Convert" pop-up menu has only one option and yields the following very simple representation.

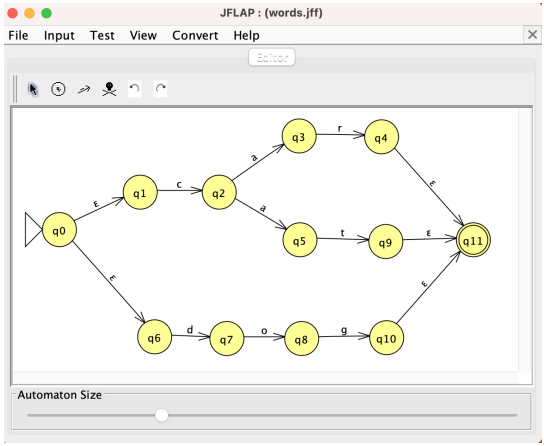


Selecting "Do All" will convert the regular expression into an equivalent finite automaton – although it may once again need to be untangled and rearranged into a more palatable representation.

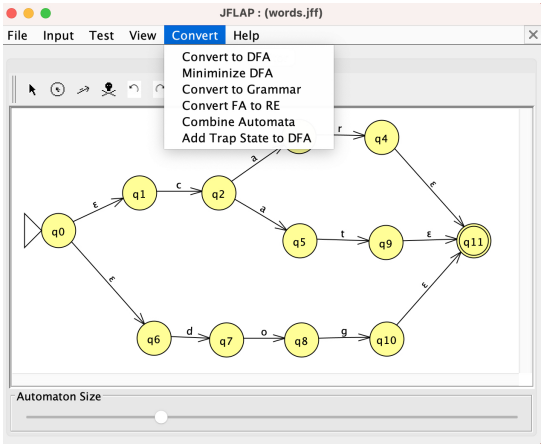


FA to Regular Expression

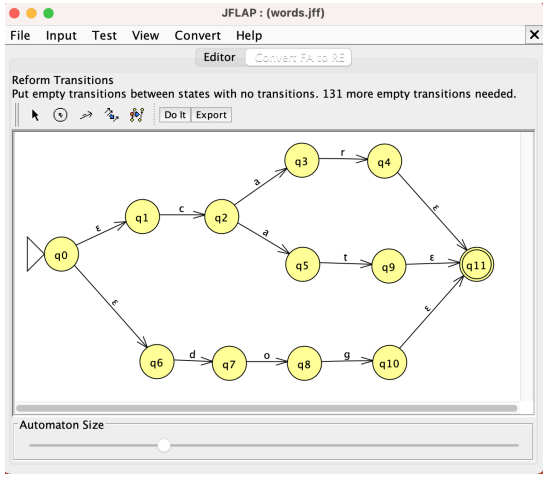
First, define the finite automaton for consideration.



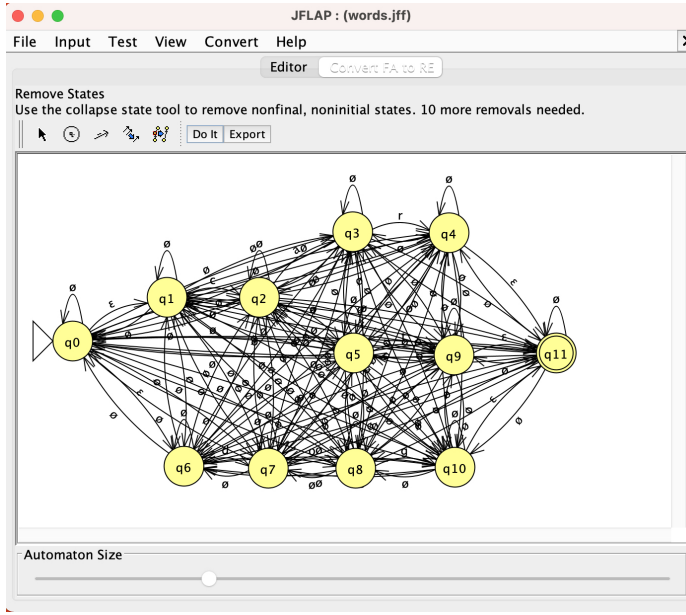
The "Convert" pop-up menu contains the option "Convert FA to RE".



The next screen will be a carbon copy of the previous screen! However, two options appear above the finite automaton: "Do It" and "Export".

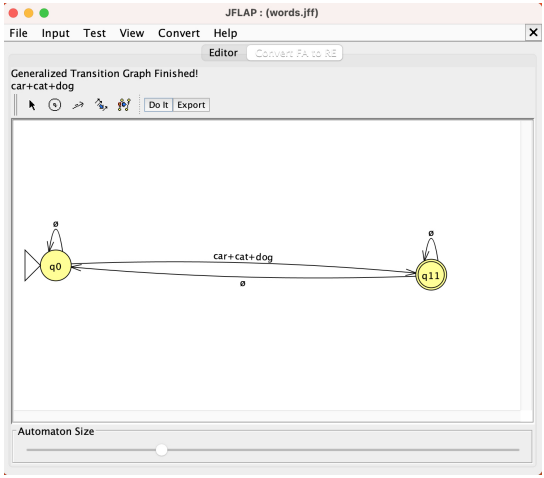


Select the "Do It" option and you will generate one gigantic collection of transitions between every pair of nodes

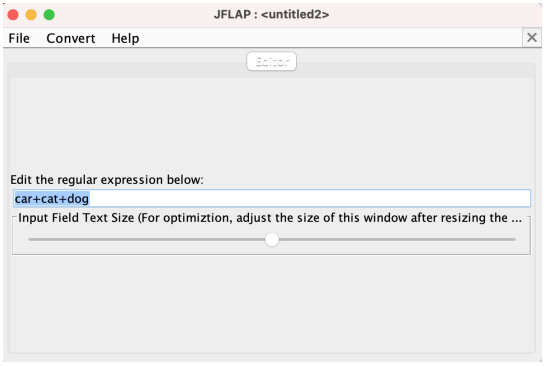


Select the "Do It" option again and you will reduce the giant col-

lection of transitions significantly



Lastly, select the "Export" option and the display screen will present the equivalent regular expression.



Chapter 5

Push Down Automata

5.1 Definitions

The second (and more complicated) variation on a Turing Machine is a **push down automaton**. A push down automaton is a Turing Machine with:

- its tape is used for input only (reading the tape sequentially in a left-to-right order)
- no output is generated
  but temporary storage is provided by a tape operating as a stack
- all transitions are defined by the current state, the current input symbol, and the symbol at the top of the stack
- an undefined transition causes the machine to stop
- stack underflow causes the machine to stop
  otherwise machine stops when no more input is available
- input strings are either ACCEPTED or REJECTED

Definition 5.1.1. A **push down automaton** M is a 6-tuple (V, Σ, Γ, δ, q_0, F) where V, Σ, and Γ are finite sets and

1. V is a set of **vertices** or **states**
2. Σ is a set of **input symbols** or **alphabet**
3. Γ is a set of **stack symbols**
   $\Sigma \subseteq \Gamma$
   $s \in \Sigma$ is called a **terminal symbol**
   $s \in \Gamma \sim \Sigma$ is called a **nonterminal symbol**
4. $\delta : Q \times \Sigma_\epsilon \times \Gamma_\epsilon \to \wp (Q \times \Gamma_\epsilon)$
5. $q_0 \in V$ is the **start state**
6. $F \subseteq V$ is a set of **accept states**

Comment. The above definition allows for choice in the definition of the transition function δ. Hence, the definition is for a *nondeterministic* push down automaton.

Definition 5.1.2. The language **recognized** by a finite push down automaton M is the collection

$$\text{L (M)} = \{ \text{ words } \omega \mid \text{M accepts } \omega \}.$$

The transition function δ may be represented in a variety of ways.

δ (q , s , t) = (q_{next} , t'),

where s is input symbol,
t is *popped* from the stack,
t' is *pushed* onto the stack

For state diagrams found in many textbooks, an arrow ($\rightarrow$) is used to represent the pop / push combination:

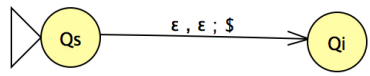
s , t $\rightarrow$ t'

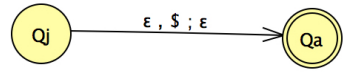
For state diagrams created in JFLAP, the arrow is typically replaced with a semicolon (;):

s , t ; t'

Any of the three symbols s, t, or t' may be replaced with ϵ, i.e., no input symbol considered, no pop off the stack, or no push onto the stack.

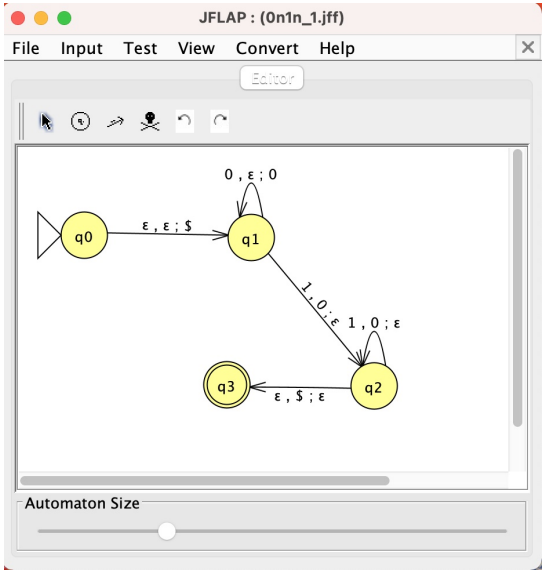
Furthermore, stacks typically have a boolean function empty associated with them. Our examples incorporate the use a sentinel symbol ($) as part of the definition of the push down automaton instead. Note that JFLAP uses the symbol **Z** as the sentinel; it is automatically pushed onto the stack. Unfortunately, we need to be aware of both sentinel techniques as some of JFLAP's algorithms require the specific use of **Z**.

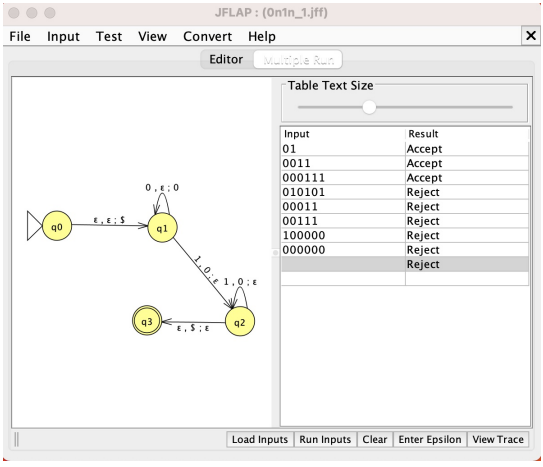




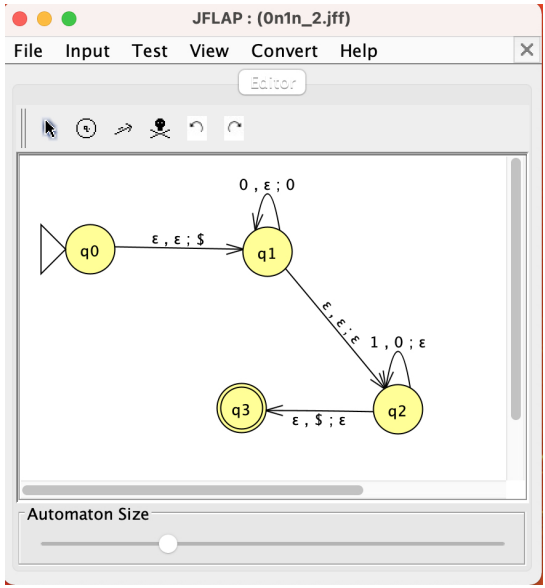
5.2 Examples

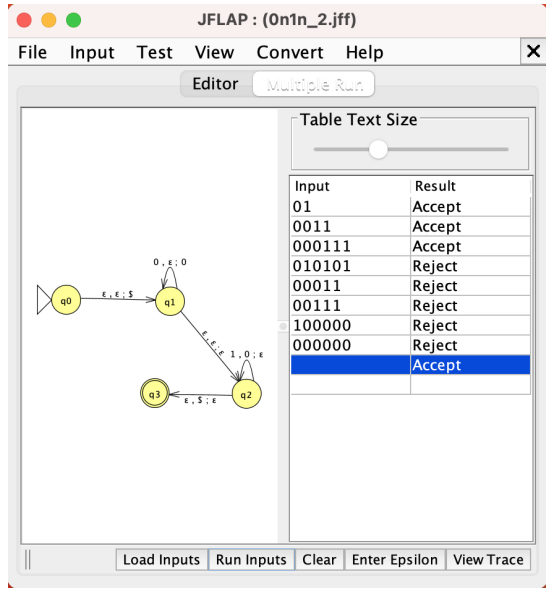
$\mathbf{L = \{\ 0^n 1^n \mid n > 0\ \}}$





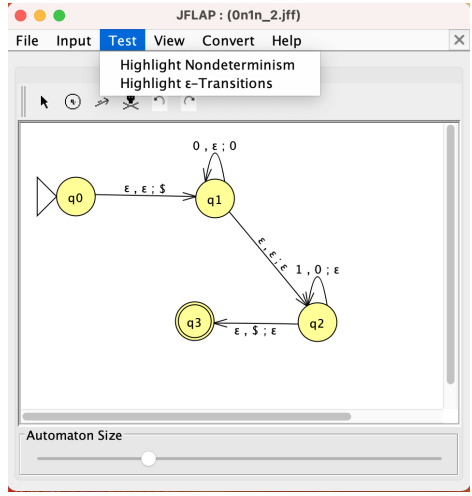
$$\mathbf{L = \{\ 0^n 1^n \mid n \geq 0\ \}}$$





Notice the subtle difference between the two previous examples. The first example is **deterministic** and the second example is not. The fist example does not accept the **empty string** ϵ and the second example does.

As we saw with finite automata in JFLAP, we can test for ϵ-transitions and nondeterminism with push down automata. The "Test" pop-up menu behaves in exactly the same manner as with finite automata.

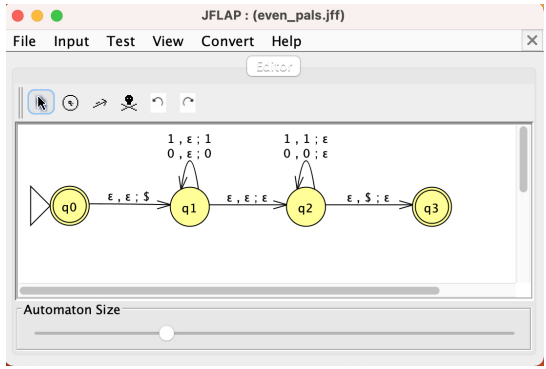


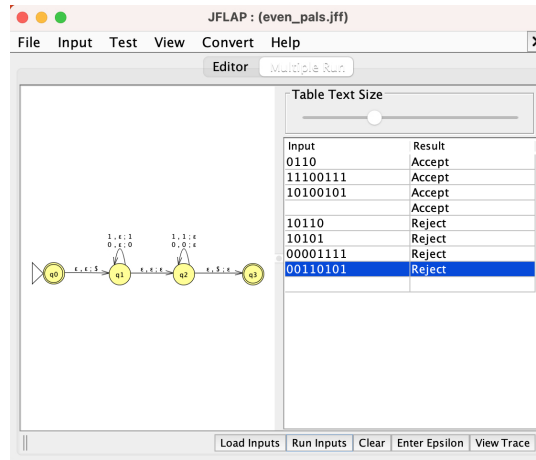
Since finite automata are finite state machines without a stack, any regular language is automatically recognizable by a push down automaton – you simply do not use the stack for any temporary storage! However, push down automata have the ability to recognize / accept a much larger collection of languages.

The next two examples are languages which are not regular but are recognized be the pictured push down automata.

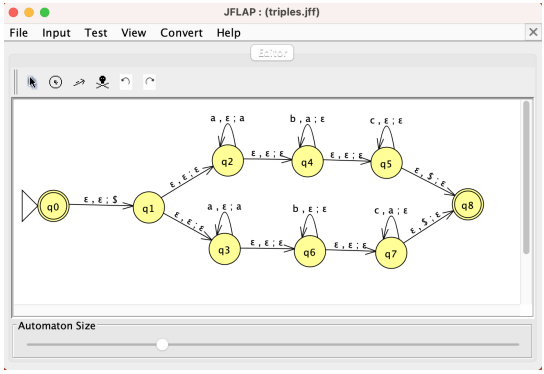
Also, it is important to note here that, unlike finite automata, nondeterminism is not possible to duplicate with a deterministic equivalent using a push down automaton.

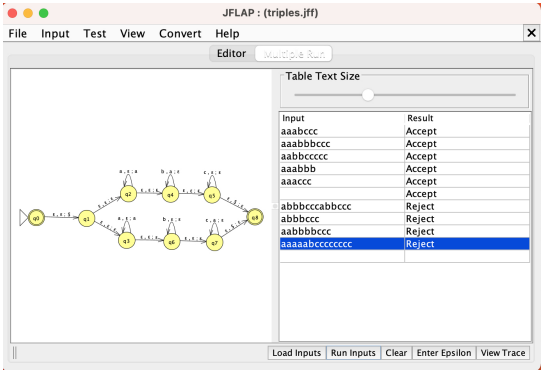
$$\mathbf{L} = \{\ \omega\ \omega^R \mid \omega\ \epsilon\ \{0,1\}^*\ \}$$





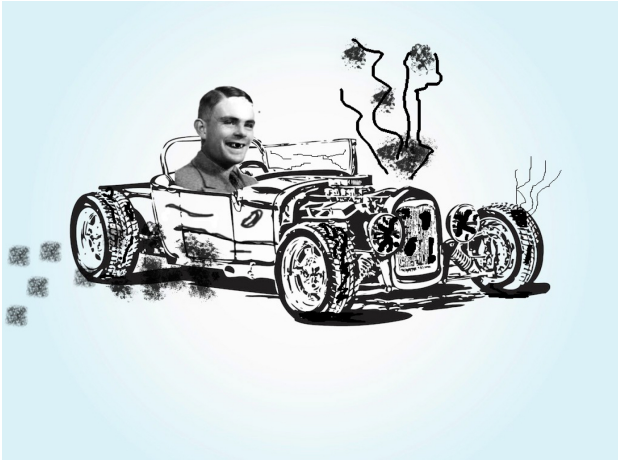
L = { $a^i b^j c^k$ | i,j,k ≥ 0 and ((i = j) or (i = k)) }





Chapter 6

Grammars



6.1 Definitions

Definition 6.1.1. A **grammar** G is a four-tuple (Σ , V , S , P) where:

1. Σ is a finite nonempty set of **terminals** or **alphabet**
2. V is a finite nonempty set of **nonterminals** or **variables**
3. $S \in V$ is a distinguished nonterminal called the **start symbol**
4. P is a set of **production rules** of the form: $a \rightarrow b$
   $a \in (\Sigma \cup V)^* \, V(\Sigma \cup V)^*$
   $b \in (\Sigma \cup V)^*$

Chomsky Hierarchy

Type 0 unrestricted grammar

production rules of the form: $a \rightarrow b$ as defined above
$a \in (\Sigma \cup V)^* \, V \, (\Sigma \cup V)^*$
$b \in (\Sigma \cup V)^*$

Type 1 context sensitive grammar

production rules of the form: $a \rightarrow b$
$| a | \leq | b |$
$S \rightarrow \epsilon$ allowed only if $S \notin$ right hand side of P
for any production rule in P

Type 2 context free grammar

production rules of the form: $a \rightarrow b$
$| a | = 1$
left hand side of P is single nonterminal
for any production rule in P

Type 3 regular (linear) grammar

production rules of the form: $a \rightarrow b$
$| a | = 1$
b is one of four types: ϵ, c, B, or c B

Relationship Hierarchy and Finite State Machines

grammar	automaton	language
regular	FA	regular
context free	NPDA	context free
context sensitive	LBA	context sensitive
unrestricted	TM decider	recursive
	TM recognizer	recursively enumerable

Terminology

Definition 6.1.2. A **linear bounded automaton** M is a Turing Machine where the length of the output may not exceed the size of the input.

Definition 6.1.3. Turing Machine recognizer is a generic Turing Machine which accepts or recognizes input strings ω.

Note: Because Turing Machines are more complicated than either finite automata and push down automata, it is possible that a Turing Machine will fall prey to infinite loops of useless calculation. Hence, a generic Turing Machine may not complete its analysis on some input strings ω.

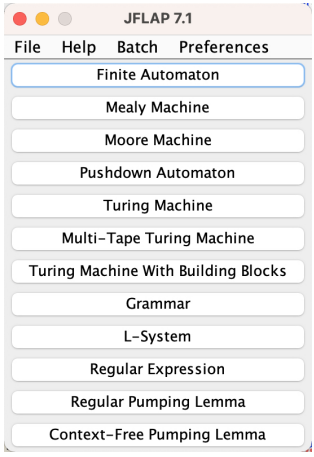
Definition 6.1.4. A **Turing Machine decider** is a Turing Machine which will stop on all input strings ω with either an accept or reject decision.

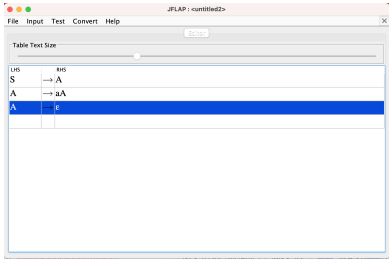
6.2 Examples

$\mathbf{L = \{a^n \mid n \geq 0\}}$

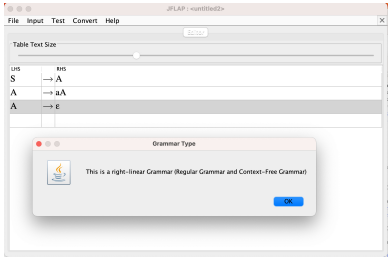
$\mathbf{S \rightarrow A}$
$\mathbf{A \rightarrow aA \mid \mid \epsilon}$

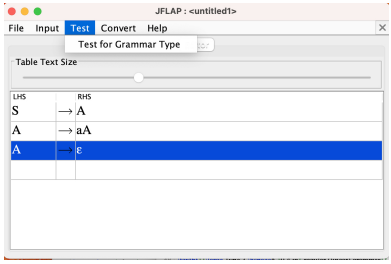
From the JFLAP main menu, select **Grammar**. You will then be prompted to enter a collection of grammar rules (productions), one per line.



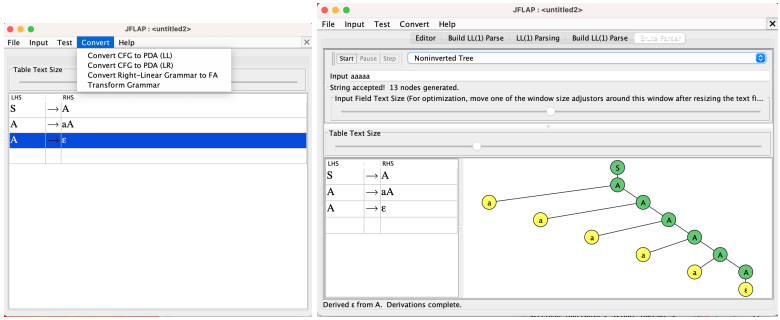


After entering the desired grammar rules, you can check the type of grammar within the Chomsky Hierarchy.

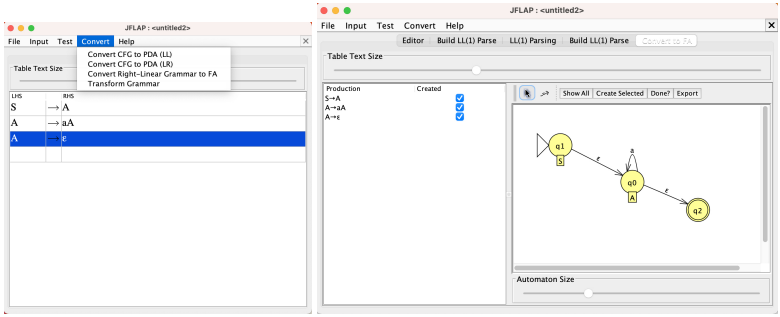




To build a derivation tree, click on **Input** and from the pop-up menu select **Brute Force Parse**.



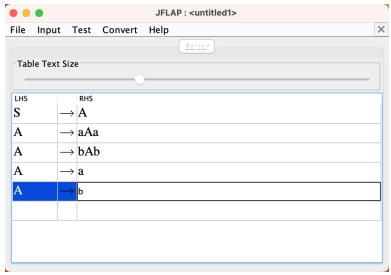
Lastly, JFLAP allows the user to convert from the grammar production rules to the corresponding finite state machine which recognizes the language. Click on **Convert** and from the pop-up menu select the appropriate type of machine (either PDA for context free grammar or FA for regular grammar).

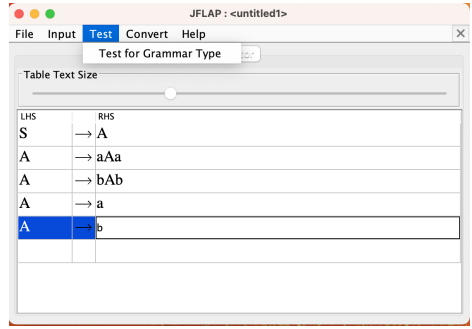


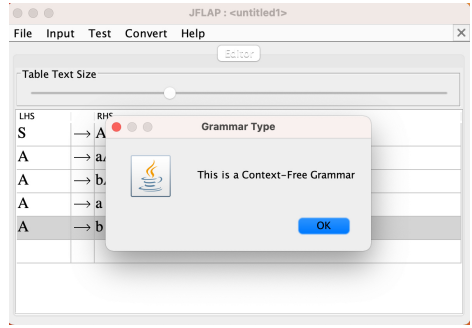
$\mathbf{L = \{\ \omega \in \{a,b\}^* \mid \omega \text{ is palindrome}\ \}}$

$\mathbf{S \rightarrow A}$
$\mathbf{A \rightarrow aAa \mid bAb \mid a \mid b \mid \epsilon}$

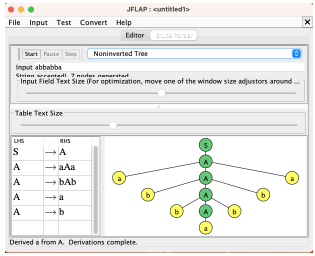
This example is a context free grammar, as demonstrated in the first few graphics below:



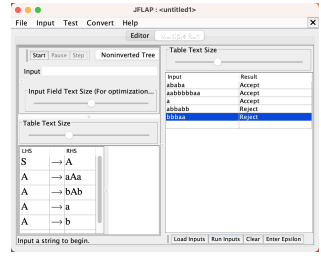




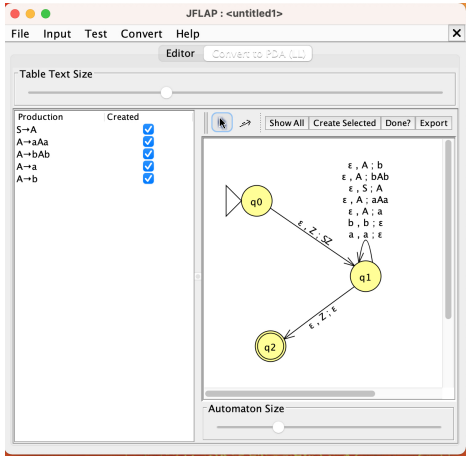
To build a single derivation tree, click on **Input** and from the pop-up menu select **Brute Force Parse**.



To test multiple input samples, click on **Input** and from the pop-up menu select **Multiple Brute Force Parse**.



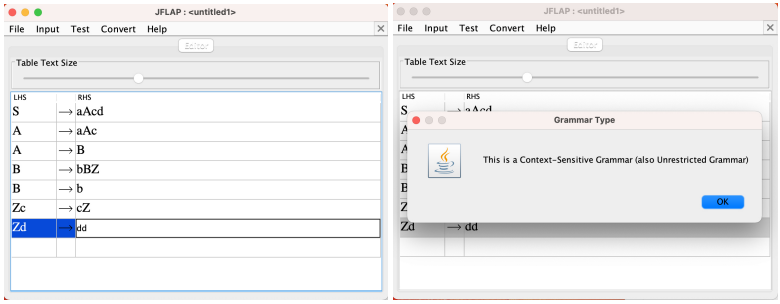
Lastly, context free grammars are equivalent to nondeterministic push down automata. JFLAP incorporates the conversion algorithm in its many features. Sinply click on **Convert** and from the pop-up menu select **Convert CFG to PDA (LL)**.

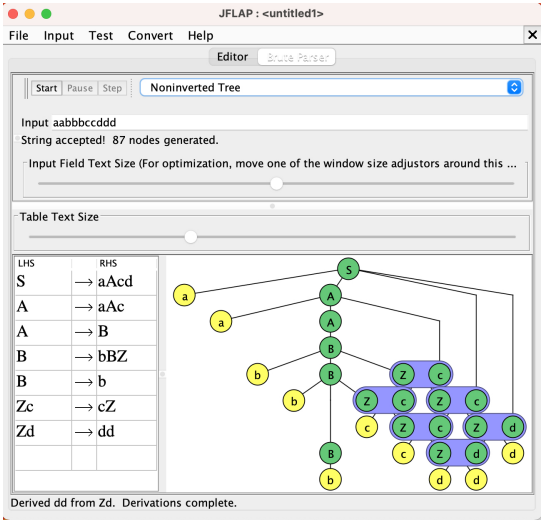


$\mathbf{L = \{a^i\ b^j\ c^m\ d^n \mid i{=}m \text{ and } j{=}n\ \}}$

$\mathbf{S \rightarrow aAcd}$
$\mathbf{A \rightarrow aAc \mid B}$
$\mathbf{B \rightarrow bBZ \mid b}$
$\mathbf{Zc \rightarrow cZ}$
$\mathbf{Zd \rightarrow dd}$

This is an example of a context sensitive grammar.

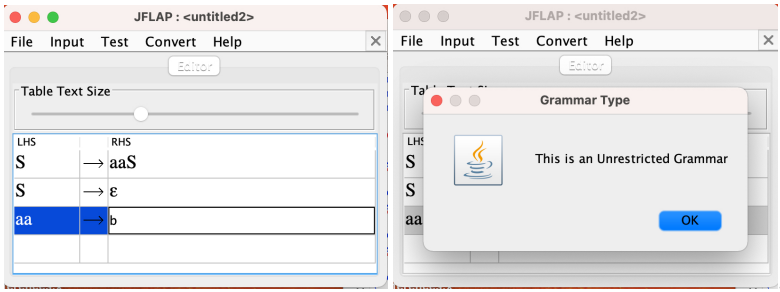


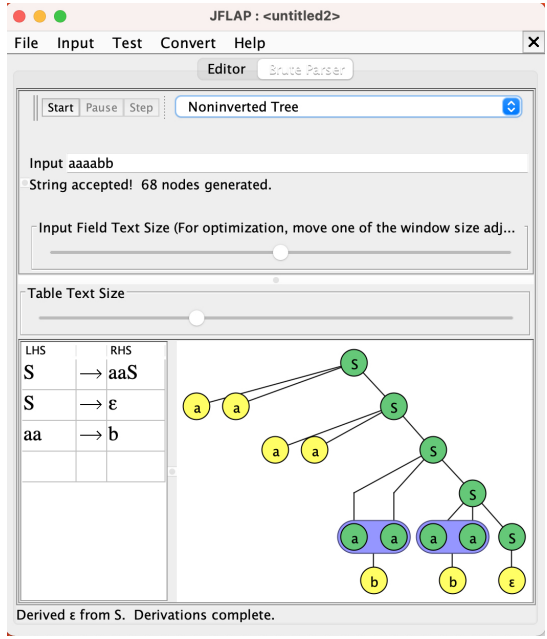


$\mathbf{L = \{a^m\ b^n \mid m \text{ is even and } n \geq 0\}}$

$\mathbf{S \rightarrow aaS \mid \epsilon}$
$\mathbf{aa \rightarrow b}$

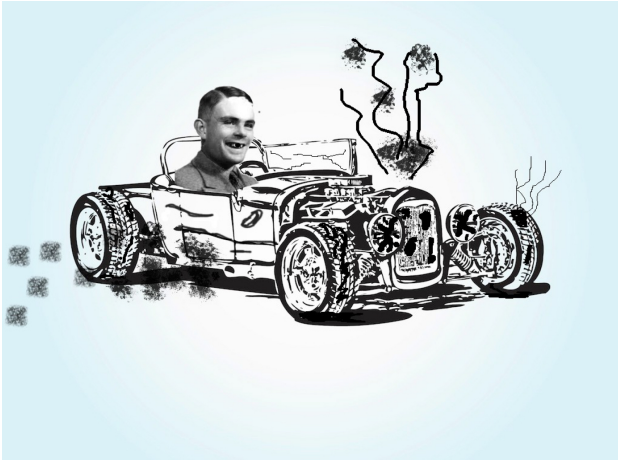
This is an example of an unrestricted grammar.





Chapter 7

Grammars and Machines



7.1 Recap

In the last chapter we summarized the equivalency between the hierarchy of grammars and the hierarchy of finite state machines.

We list these equivalencies here once again.

Relationship Hierarchy and Finite State Machines

grammar	automaton	language
regular	FA	regular
context free	NPDA	context free
context sensitive	LBA	context sensitive
unrestricted	TM decider	recursive
	TM recognizer	recursively enumerable

In the last chapter we also saw that JFLAP had among its implementation features two conversion algorithms:

- regular grammar → finite automaton
- context free grammar → push down automaton

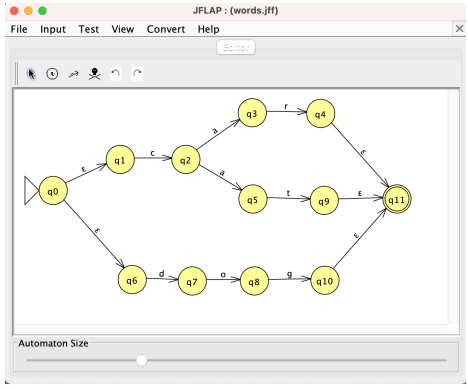
No conversion algorithms were presented for context sensitive grammars or for unrestricted grammars.

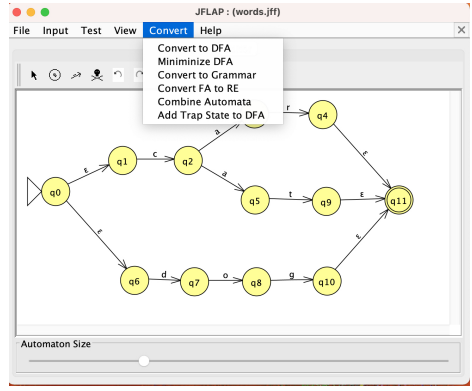
In this chapter, we conclude our crash course on JFLAP by first considering possible conversion algorithms in the opposite direction: FROM finite state machines TO grammars and second considering a very specialize conversion algorithm within context free grammars.

7.2 Conversions: Machines TO Grammars

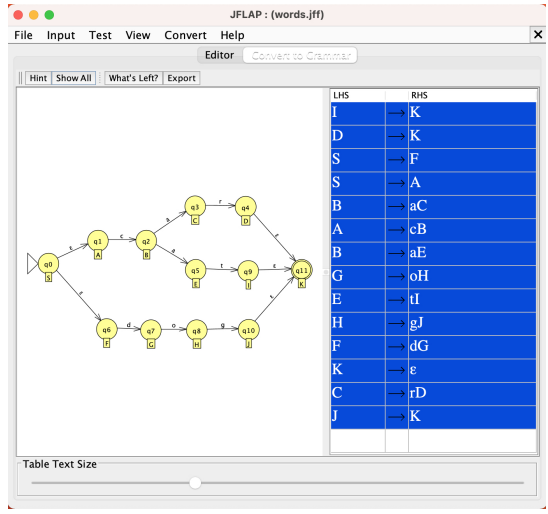
All conversions are FROM a finite state machine (any flavor) TO a grammar. Click on **Convert** and select the **Convert to Grammar** option from the pop-up menu.

finite automaton to regular grammar





The resulting grammar will be displayed as in the following graphic:



push down automaton to context free grammar

The algorithm implemented in JFLAP forces the user to massage the (nondeterministic) push down automaton to satisfy two necessary conditions:

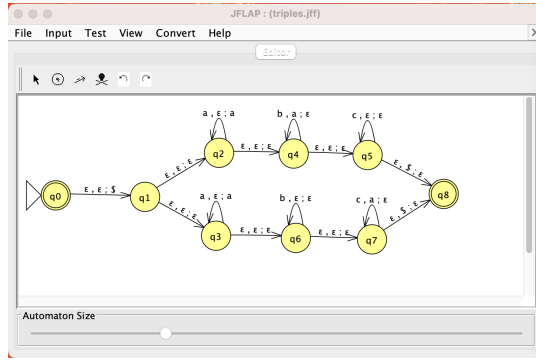
- there must be a unique final state with all transitions entering that state being ϵ**,Z;**ϵ
- every transition must be either a push or a pop of a **single character**

Creating a unique final state is easy with nondeterminism. Simply modify transitions using the **$** sentinel to the JFLAP sentinel **Z**. Although this might seem a bit awkward at first, it is a trivial substitution. You might also choose to move away from the **$** sentinel entirely in the future.

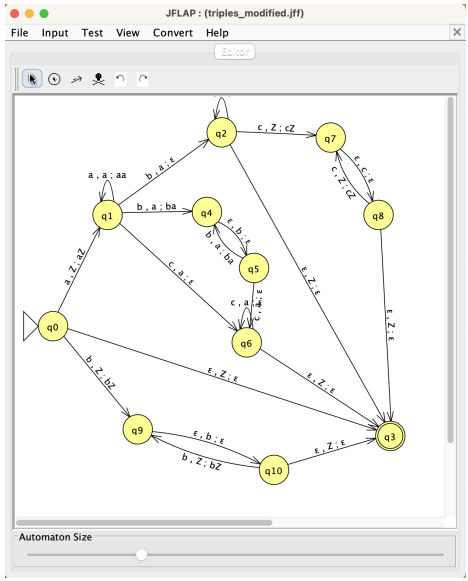
The second requirements is a bit more complicated. Remember that JFLAP automatically removes the top character on the stack (except *epsilon*) by default. To appear to have pushed a single character onto the stack one must push two and immediately pop one!

The following sequence of graphics will illustrate the entire process! We begin with the following language:

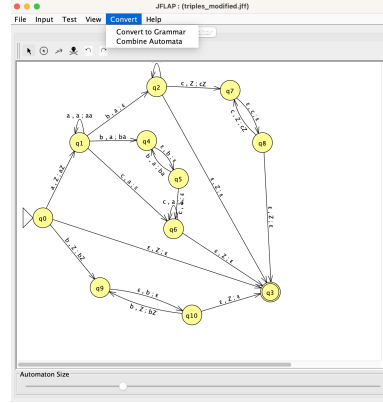
L = { a^i b^j c^k | i=j or i=k }



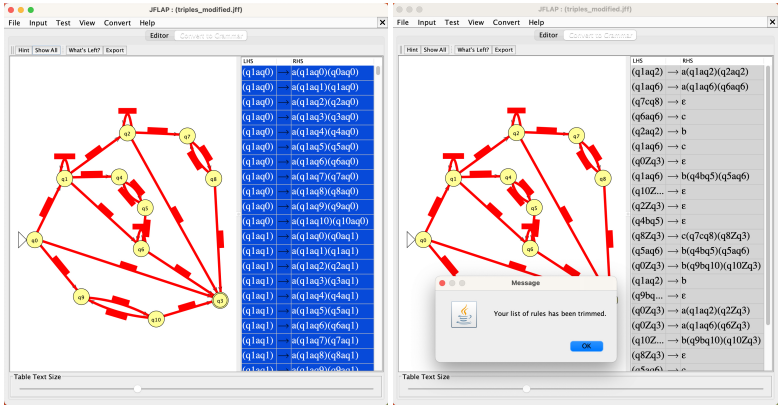
Although this representation would be perfectly fine in most applications, the conversion algorithm we are about to apply has the two requirements described above. The nondeterministic push down automaton does not satisfy to two restrictions and the automaton must be rewritten in an acceptable form, as illustrated below.

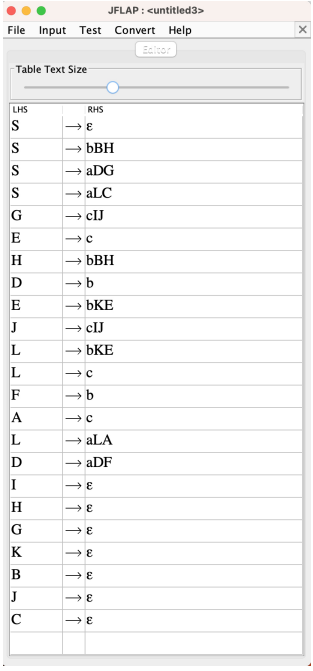


Clicking on the **Convert** option allows us to convert the machine to its equivalent grammar.



The resulting graphic contains two options: **Show All** and **Export**. The first shows a very confusing notation for the desired grammar; the second modifies the representation somewhat and trims unnecessary elements from the grammar.



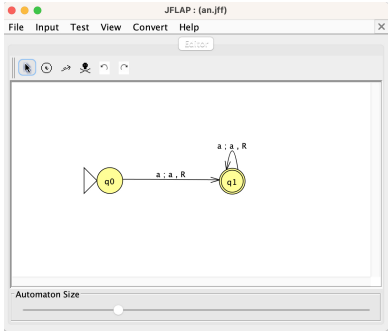


turing machine to unrestricted grammar

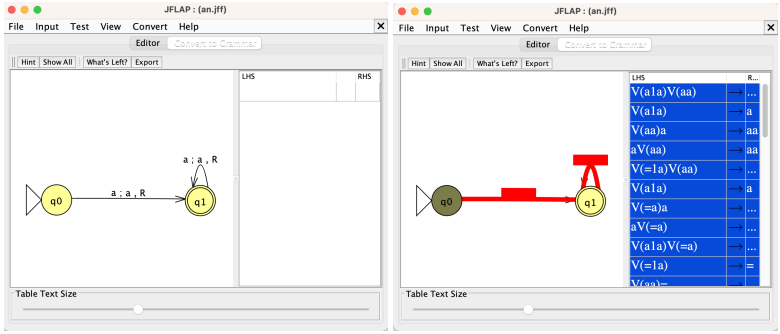
This section will be short and sweet. It is good news immediately followed by bad news!

The good news is that the conversion is three very straightforward steps; the bad news is that the conversion algorithm has exponential growth and works on very small problems.

$\mathbf{L = \{\ a^n \mid n \geq 1\ \}}$

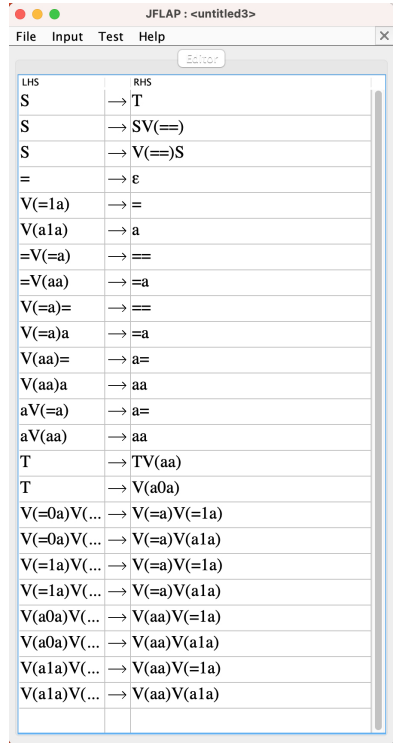


Clicking on the **Convert** option allows us to convert the machine to its equivalent grammar.



The last step in the process is to click on the **Export** option to create a new window containing the resulting grammar.

Note: No attempt is made to simplify the grammar from its shorthand notation. In addition, this last step was (for me) where the conversion process typically failed.



LHS		RHS
S	⟶	T
S	⟶	SV(==)
S	⟶	V(==)S
=	⟶	ε
V(=1a)	⟶	=
V(a1a)	⟶	a
=V(=a)	⟶	==
=V(aa)	⟶	=a
V(=a)=	⟶	==
V(=a)a	⟶	=a
V(aa)=	⟶	a=
V(aa)a	⟶	aa
aV(=a)	⟶	a=
aV(aa)	⟶	aa
T	⟶	TV(aa)
T	⟶	V(a0a)
V(=0a)V(...	⟶	V(=a)V(=1a)
V(=0a)V(...	⟶	V(=a)V(a1a)
V(=1a)V(...	⟶	V(=a)V(=1a)
V(=1a)V(...	⟶	V(=a)V(a1a)
V(a0a)V(...	⟶	V(aa)V(=1a)
V(a0a)V(...	⟶	V(aa)V(a1a)
V(a1a)V(...	⟶	V(aa)V(=1a)
V(a1a)V(...	⟶	V(aa)V(a1a)

7.3 Chomsky Normal Form

The final conversion algorithm available in JFLAP is a conversion from a context free grammar to a very special equivalent form of context free grammar called **Chomsky Normal Form**.

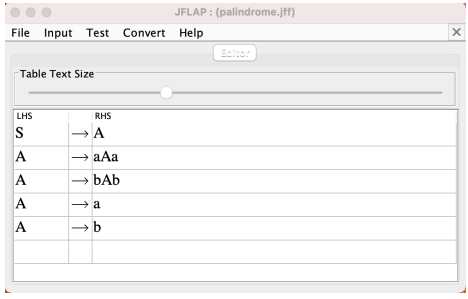
Definition 7.3.1. Chomsky Normal Form is a context free grammar with all its production rules having one of the following three forms:

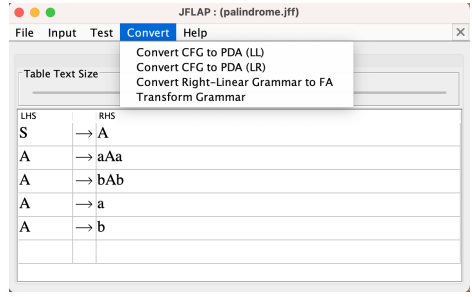
- $A \rightarrow BC$ exactly two non-terminals
- $A \rightarrow a$ a single character
- $S \rightarrow \epsilon$ sole exception to the above two rules

example

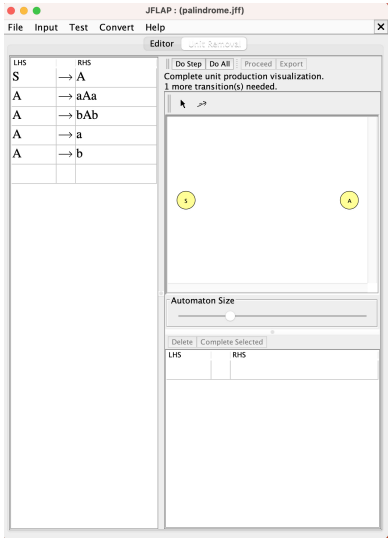
Our illustrative example will be simple palindromes:

$S \rightarrow A$
$A \rightarrow aAa \mid bAb \mid a \mid b$

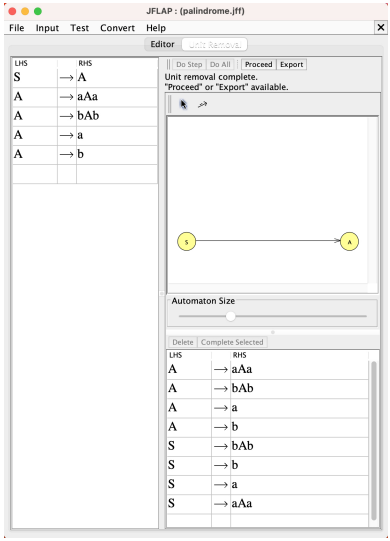




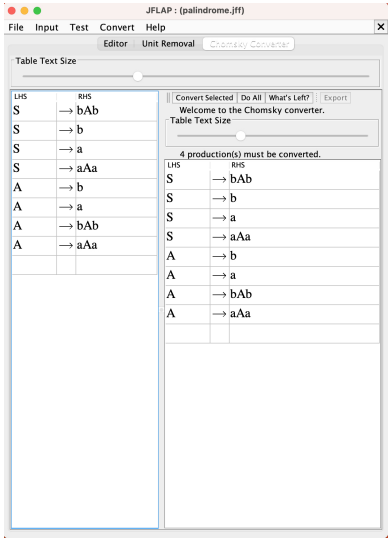
The conversion process begins with a minimalist screen display.



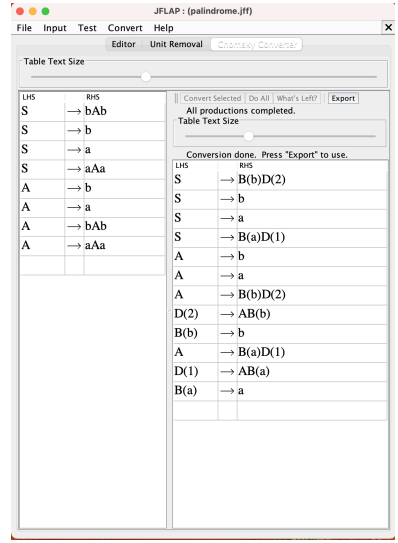
Clicking on the **Do All** option rewrites the original grammar with additional redundant rules.



Clicking on the **Proceed** option rearranges the grammar rules under consideration and discards the graphical image to the top right.



Clicking on the **Do All** option completes the final step in this somewhat lengthy process and generates the desired Chomsky Normal Form context free grammar.



Lastly, clicking on the **Export** option rewrites the grammar rules just generated with new names to highlight the fact that the grammar is indeed in Chomsky Normal Form.