CSci 311, Models of Computation, Chapter 3 Regular Languages and Regular Grammars

H. Conrad Cunningham

29 December 2015

Contents

Acknowledgements: MS student Clay McLeod assisted in preparation of these notes. These lecture notes are for use with Chapter 3 of the textbook: Peter Linz. *Introduction to Formal Languages and Automata*, Fifth Edition, Jones and Bartlett Learning, 2012. The terminology and notation used in these notes are similar to those used in the Linz textbook. This document uses several figures from the Linz textbook.

Advisory: The HTML version of this document requires use of a browser that supports the display of MathML. A good choice as of December 2015 seems to be a recent version of Firefox from Mozilla.

Introduction

Regular languages

- are accepted by dfas and nfas
- but dfas and nfas are **not** concise descriptions

Thus we will examine other notations for representing regular languages.

3.1 Regular Expressions

3.1.1 Syntax

We define the *syntax* (or structure) of regular expressions with an inductive definition.

Linz Definition 3.1 (Regular Expression): Let Σ be a given alphabet. Then:

1. $\emptyset$, λ, and $a \in \Sigma$ are all *regular expressions*. These are called *primitive regular expressions*.
2. If r_1 and r_2 are regular expressions, then $r_1 + r_2$, $r_1 \cdot r_2$, r_1^*, and (r_1) are also regular expressions.
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 in (2).

We use the the regular expression operators as follows:

- $r + s$ represents the *union* of two regular expressions.
- $r \cdot s$ is the *concatenation* of two regular expressions.
- r^* is the *star closure* of a regular expression.
- (r) is the same as regular expression r. It is parenthesized to express the order of operations explicitly.

For example, consider regular expression $(a + (b \cdot c))^*$ over the alphabet $\{a, b, c\}$. Note the use of parentheses.

- a, b, and c are *primitive regular expressions*.
- $(b \cdot c)$ is a *concatenation* of regular expressions a and b.
- $(a + (b \cdot c))$ is *union* of regular expressions a and $(b \cdot c)$.
- $(a + (b \cdot c))^*$ is the *star-closure* of regular expression $(a + (b \cdot c))$.

As with arithmetic expressions, *precedence rules* and conventions can be used to relax the need for parentheses.

- *Star-closure* $(^*)$ has a higher precedence (i.e., priority or binding power) than concatenation $(\cdot)$. That is, $r \cdot s^*$ is equal to $r \cdot (s^*)$, not $(r \cdot s)^*$.
- *Concatenation* $(\cdot)$ higher precedence than union $(+)$. That is, $r \cdot s + t$ is equal to $(r \cdot s) + t$, not $r \cdot (s + t)$. And, transitively, *star-closure* has a higher precedence than concatenation.
- *Concatenation* operator $(\cdot)$ can usually be omitted. That is, rs means $r \cdot s$.

A string $(a + b+)$ is *not* a regular expression. It cannot be generated using the above definition (as augmented by the precedence rules and convention).

3.1.2 Languages Associated with Regular Expressions

But what do we "mean" by a regular expression? That is, what is its *semantics*.

In particular, what languages do regular expressions describe?

Consider the regular expression $(a+(b\cdot c))^*$ from above. As implied by the names for the operators, we intend this regular expression to represent the language $(\{a\} \cup \{bc\})^*$ which is $\{\lambda, a, bc, aa, abc, bca, bcbc, aaa, aabc, bcaa, \ldots\}$.

We again give an inductive definition for the language described by a regular expression. It must consider all the cases given in the definition of regular expression itself.

Linz Definition 3.2: The *language* $L(r)$ denoted by any regular expression r is defined (inductively) by the following rules.

Base cases:

1. $\emptyset$ is a regular expression denoting the empty set.
2. λ is a regular expression denoting $\{\lambda\}$.
3. For every $a \in \Sigma$, a is a regular expression denoting $\{a\}$.

Inductive cases: If r_1 and r_2 are regular expressions, then

4. $L(r_1 + r_2) = L(r_1) \cup L(r_2)$
5. $L(r_1 \cdot r_2) = L(r_1)L(r_2)$
6. $L((r_1)) = L(r_1)$
7. $L(r_1^*) = (L(r_1))^*$

3.1.3 Linz Example 3.2

Show the language $L(a^* \cdot (a + b))$ in set notation.

	$L(a^* \cdot (a + b))$
=	{ Rule 5 }
	$L(a^*)L(a + b)$
=	{ Rule 7 }
	$(L(a))^*L(a + b)$
=	{ Rule 4 }
	$(L(a))^*(L(a) \cup L(b))$
=	{ definition of star-closure of languages }
	$\{\lambda, a, aa, aaa, \ldots\}\{a, b\}$
=	{ definition of concatenation of languages }
	$\{a, aa, aaa, ..., b, ab, aab, aaab, \ldots\}$

3.1.4 Examples of Languages for Regular Expressions

Consider the languages for the following regular expressions.

$L(a^* \cdot b \cdot a^* \cdot b \cdot (a+b)^*) = \{a\}^*\{b\}\{a\}^*\{b\}\{a,b\}^*$
$= \{w : w \in \{a,b\}^*, n_b(w) \geq 2\}$
$L((a+b)^* \cdot b \cdot a^* \cdot b \cdot a^*) = \{a,b\}^*\{b\}\{a\}^*\{b\}\{a\}^*$
$=$ same as above
$L((a+b)^* \cdot b \cdot (a+b)^* \cdot b \cdot (a+b)^*) = \{a,b\}^*\{b\}\{a,b\}^*\{b\}\{a,b\}^*$
$=$ same as above

3.1.5 Linz Example 3.4

Consider the regular expression $r = (aa)^*(bb)^*b$.

- This expression denotes the set of all strings with an even number of a's followed by an odd number of b's.
- In set notation, $L(r) = \{a^{2n}b^{2m+1} : n \geq 0, m \geq 0\}$.

3.1.6 Linz Example 3.5

For $\Sigma = \{0,1\}$, give a regular expression r such that $L(r) = \{w \in \Sigma^* :$ w has at least one pair of consecutive zeros $\}$.

- 00 must appear somewhere in any string.
- Before and after 00 there is an arbitrary string $(0+1)^*$.
- $r = (0+1)^*00(0+1)^*$

3.1.7 Examples of Regular Expressions for Languages

Show regular expressions on the alphabet $\{a,b\}$ for the following languages.

- **exactly one "a"** b^*ab^*
- **at least one "a"** $b^*a(a+b)^*$ – featuring first a
  $(a+b)^*a(a+b)^*$ – featuring middle a
  $(a+b)^*ab^*$ – featuring last a
- **at most one "a"** $b^*ab^* + b^*$
  $b^*(a+\lambda)b^*$
- **all a's immediately followed by a** b $(b^*abb^*)^* + b^*$

3.2 Connection Between Regular Expressions and Regular Languages

3.2.1 Regular Expressions Denote Regular Languages

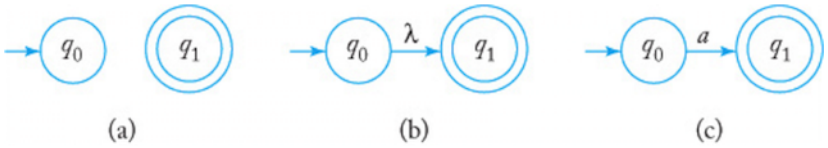
Regular expressions provide a convenient and concise notation for describing regular languages.

Linz Theorem 3.1 (NFAs for Regular Expressions): Let r be a regular expression. Then there exists some nondeterministic finite accepter (nfa) that accepts $L(r)$. Consequently, $L(r)$ is a regular language.

Proof Sketch: Show that any regular expression generated from the inductive definition corresponds to an nfa. Here we proceed informally.

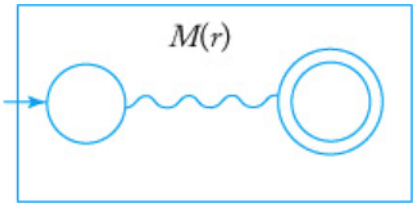
Linz Figure 3.1 diagrammatically demonstrates that there are nfas that correspond to the primitive regular expressions.

(a) nfa accepts $\emptyset$
(b) nfa accepts $\{\lambda\}$
(c) nfa accepts $\{a\}$



Linz Fig. 3.1: Primitive Regular Expressions as NFA

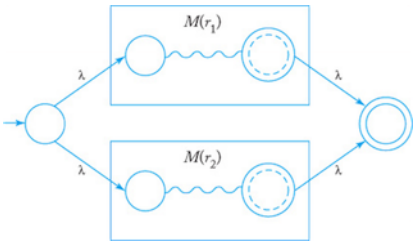
Linz Figure 3.2 shows a general scheme for a nondeterministic finite accepter (nfa) that accepts $L(r)$, with an initial state and one final state.



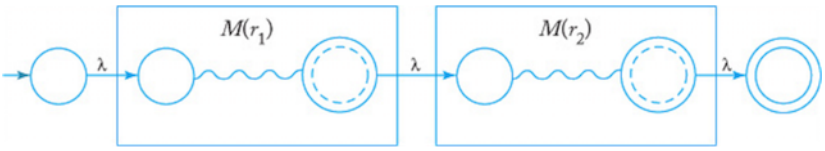
Linz Fig. 3.2: Scheme for NFA Accepting L(r)

Linz Figure 3.3 gives an nfa for $L(r_1 + r_2)$. Note the use of λ-transitions to connect the two machines to the new initial and final states.

Linz Figure 3.4 shows an nfa for $L(r_1 r_2)$. Again note the use of λ-transitions to connect the two machines to the new initial and final states.

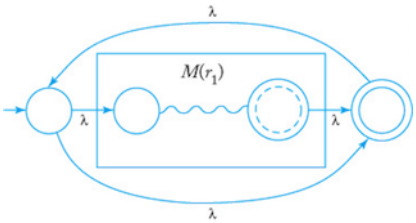


Linz Fig. 3.3: NFA for Union



Linz Fig. 3.4: NFA for Concatenation

Linz Figure 3.5 shows an nfa for $L(r_1^*)$. Note the use of λ-transitions to represent zero-or-more repetitions of the machine and to connect it to the new initial and final states.



Linz Fig. 3.5: NFA for Star-Closure

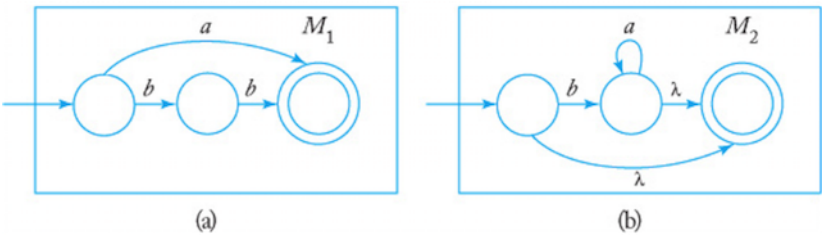
Thus, Linz Figures 3.3 to 3.5 illustrate composing nfas for any regular expression from the nfas for its subexpressions. Of course, the initial and final states of components are replaced by the initial and final states of the composite nfa.

3.2.2 Linz Example 3.7

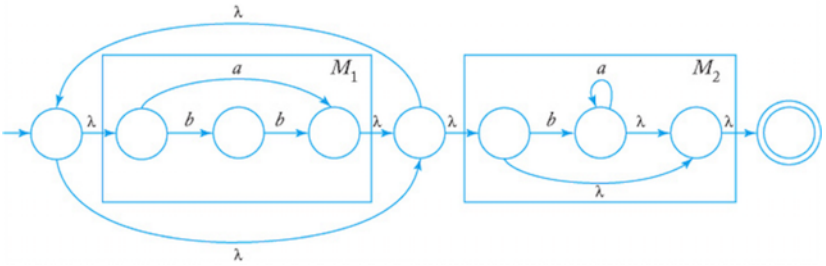
Show an nfa that accepts $r = (a + bb)^*(ba^* + \lambda)$.

Linz Figure 3.6, part (a), shows M_1 that accepts $L(a + bb)$. Part (b) shows M_2 that accepts $L(ba^* + \lambda)$.

Linz Figure 3.7 shows an nfa that accepts $L((a + bb)^*(ba^* + \lambda)$.



Linz Fig. 3.6: Toward a Solution to Ex. 3.6



Linz Fig. 3.7: Solution for Ex. 3.6

3.2.3 Converting Regular Expressions to Finite Automata

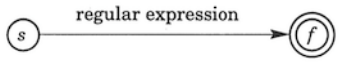
The construction in the proof sketch and example above suggest an algorithm for converting regular expressions to nfas.

This algorithm is adapted from pages 273-4 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

The diagrams in this section are from the Hein book, which uses a slightly different notation than the Linz book. In particular, these diagrams use capital letters for the expressions.

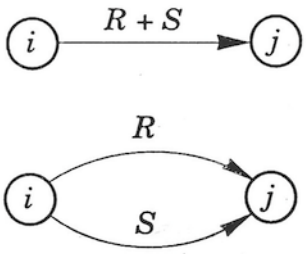
Algorithm to convert a regular expression to an nfa

- Start with a "machine" with a single start state, a single final state, and a connecting edge labeled with the regular expression.

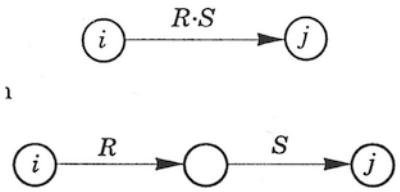


- While there are edges labeled with regular expressions other than elements of the alphabet or λ apply any of the following rules that are applicable:

1. If an edge is labeled with $\emptyset$, then remove the edge.
2. If an edge is labeled with $r + s$, then replace the edge with two edges labeled with r and s connecting the same source and destination states.



3. If an edge is labeled with $r \cdot s$, the replace the edge with an edge labeled r connecting the source to a new intermediate state, followed by an edge labeled s connecting the intermediate state to the destination.

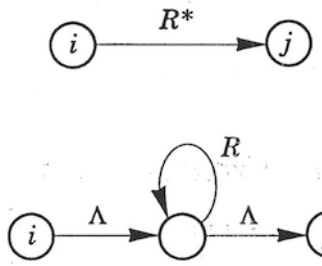


4. If an edge is labeled with r^*, then replace the edge with a new intermediate state with a self-loop labeled r with edges labeled λ connecting the source to the intermediate state and the intermediate state to the destination.

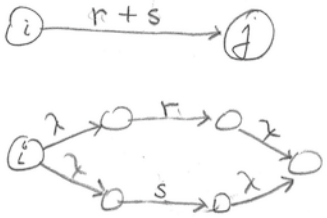
End of Algorithm

Rule 2 in the above algorithm can result in an unbounded number of edges originating at the same state. This makes the algorithm difficult to implement. To remedy this situation, replace Rule 2 as follows.

2. If an edge is labeled with $r + s$, then replace the edge with subgraphs for each of r and s. The subgraph for r consists of with a new source state connected to a new destination state with an edge labeled r. Add



edges labeled λ to connect the original source state to the new source state and the original destination state to the new destination state. Proceed similarly for s.

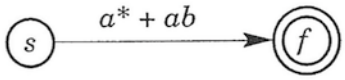


3.2.4 Example Conversion of Regular Expression to NFA

This example is from page 275 of the Hein textbook cited above.

Construct an nfa for $a^* + a \cdot b$.

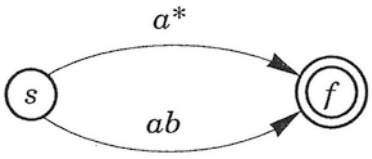
Start with a the two-state initial diagram.

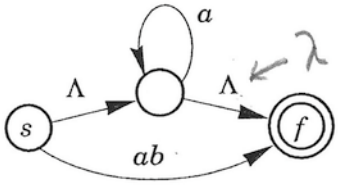


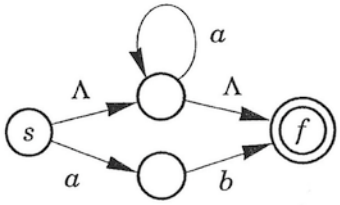
Next, apply Rule 2 to $a^* + a \cdot b$.

Next, apply Rule 4 to a^*.

Finally, apply Rule 3 to $a \cdot b$.


a*
s
f
ab


a
Λ
Λ
s
f
ab


a
Λ
Λ
s
f
a
b

3.2.5 Converting Finite Automata to Regular Expressions

The construction in the proof sketch and example above suggest an algorithm for converting finita automata to regular expressions.

This algorithm is adapted from page 276 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

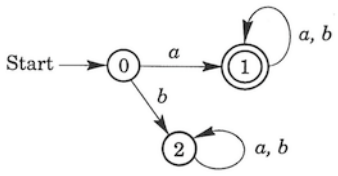
Algorithm to convert a finite automaton to a regular expression

Begin with a dfba or an nfa.

1. Create a new start state s and connect this to the original start state with an edge labeled λ.
2. Create a new final state f and connect the original final states to this state by edges labeled λ.
3. For each pair of states i and j that has more than one edge connecting them, replace all the edges with the regular expression formed using union $(+)$ to combine the labels on the previous edges.
4. Construct a sequence of new machines by eliminating one state at a time until the only states remaining are s and f. To eliminate some state k, construct a new machine as follows.
   - Let $\text{old}(i, j)$ represent the label on the edge (i, j) on the current (i.e., old) machine.
   - If there is no edge (i, j), then set $\text{old}(i, j) = \emptyset$.
   - For every pair of edges (i, k) and (k, j), where $i \neq k$ and $j \neq k$, calculate a new edge label $\text{new}(i, j)$ as follows:
     $\text{new}(i, j) = \text{old}(i, j) + \text{old}(i, k)\ \text{old}(k, k)^*\ \text{old}(k, j)$
   - For all other edges (i, j), where $i \neq k$ and $j \neq k$, set:
     $\text{new}(i, j) = \text{old}(i, j)$.
   - The states of the new machine are the states of the old machine with state k eliminated. The edges of the new machine are the (i, j) where the $\text{new}(i, j)$ has been calculated.

After eliminating all states except s and f, the regular expression is the label on the one edge remaining.

End of Algorithm

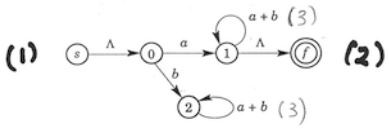


3.2.6 Example Conversion of Finite Automata to Regular Expressions

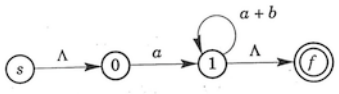
This example is from pages 277-8 of the Hein textbook cited above.

Consider the following dfa.

After applying Rule 1 (new start state), Rule 2 (new final state), and Rule 3 (create union), we get the following machine.



We can eliminate state 2 readily because it is trap state. That is, there is no path through 2 between edges adjacent to 2, so $\text{new}(i, j) = \text{old}(i, j)$ for any states $i \neq 2$ and $j \neq 2$. The resulting machine is as follows.

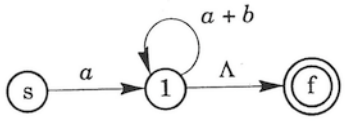


To eliminate state 0, we construct a new edge that is labeled as follows:

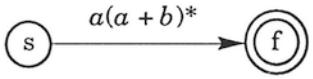
- $\textbf{new}(s, 1) = \text{old}(s, 1) + \text{old}(s, 0)\ \text{old}(0, 0)^*\ \text{old}(0, 1)$
  $= \emptyset + \lambda\emptyset^* a$
  $= a$

Thus, we can eliminate state 0 and its edges and add a new edge $(s, 1)$ labeled a.

We can eliminate state 1 by adding a new edge (s, f) labeled as follows



- **new**(s, f) = old(s, f) + old$(s, 1)$ old$(1, 1)^*$ old$(1, f)$
  $= \emptyset + a(a + b)^*\lambda$
  $= a(a + b)^*$

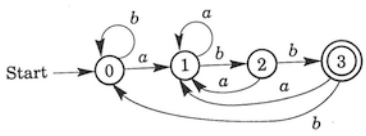


Thus the regular expression is $a(a + b)^*$.

3.2.7 Another Example Conversion of Finite Automa to Regular Expressions

This example is from pages 277-8 of the Hein textbook cited above.

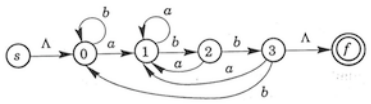
Consider the following dfa. Verify that it corresponds to the regular expression $(a + b)^*abb$.

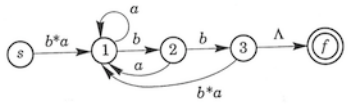


Applying Rules 1 and 2 (adding new start and final states), we get the following machine.

To eliminate state 0, we add the following new edges.

- new$(s, 1) = \emptyset + \lambda b^* a = b^* a$

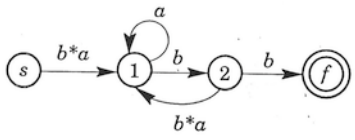




- new$(3, 1) = a + bb^*a = (\lambda + bb^*)a = b^*a$

We can eliminate either state 2 or state 3 next. Let's choose 3. Thus we create the following new edges.

- new$(2, f) = \emptyset + b\emptyset^*\lambda = b$
- new$(2, 1) = a + b\emptyset^*b^*a = a + bb^*a = (\lambda + bb^*)a = b^*a$

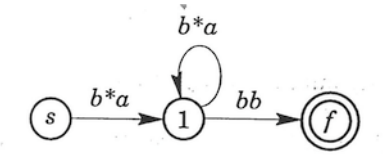


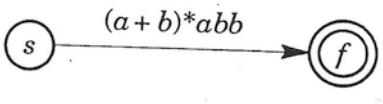
Now we eliminate state 2 and thus create the new edges.

- new$(1, f) = \emptyset + b\emptyset^*b = bb$
- new$(1, 1) = a + b\emptyset^*b^*a = (\lambda + bb^*)a = b^*a$

Finally, we remove state 1 by creating a new edge.

- **new**$(s, f) = \emptyset + b^*a(b^*a)^*bb$
  $= b^*(b^*a)^*abb$
  $= (a + b)^*abb$





3.2.8 Regular Expressions for Describing Simple Patterns

Pascal integer constants

Regular expression sdd^* where

- s : sign from $\{+, -, \lambda\}$
- d : digit from $\{0, 1, ..., 9\}$

Pattern matching

- Unix ed $/aba^*c/$ (different syntax)
- Find pattern in text

Program for Pattern Matching

We can convert a regular expression to an equivalent nfa, the nfa to a dfa, and the dfa to a transition table. We can use the transition table to drive a program for pattern matching.

For a more effiicent program, we can apply the *state reduction algorithm* to the dfa before converting to a transition table. Linz section 2.4, which we did not cover this semester, discusses this algorithm.

3.3 Regular Grammars

We have studied two ways of describing regular languages–finite automata (i.e. dfas, nfas) and regular expressions. Here we examine a third way–*regular grammars.*

Linz Definition 3.3 (Right-Linear Grammar): A grammar $G = (V, T, S, P)$ is said to be *right-linear* if all productions are of one of the forms

$$A \to xB$$
$$A \to x$$

where $A, B \in V$ and $x \in T^*$.

Similarly, a grammar is said to be *left-linear* if all productions are of the form $A \to Bx$ or $A \to x$.

A *regular grammar* is one that is either right-linear or left-linear.

- one variable on right at most
- consistently rightmost (or leftmost)

3.3.1 Linz Example 3.13

The grammar $G_1 = (\{S\}, \{a, b\}, S, P_1)$, with P_1 given as

- $S \to abS \mid a$

is right-linear.

The grammar $G_2 = (\{S, S_1, S_2\}, \{a, b\}, S, P_2)$, with productions

- $S \to S_1ab$
- $S_1 \to S_1ab \mid S_2$
- $S_2 \to a$

is left linear. Both G_1 and G_2 are regular grammars.

$L(G_1) = L((ab)^*a)$

$L(G_2) = L(aab(ab)^*)$

3.3.2 Linz Example 3.14

The grammar $G = (\{S, A, B\}, \{a, b\}, S, P)$ with productions

- $S \to A$
- $A \to aB \mid \lambda$
- $B \to Ab$

is *not regular*.

Although every production is either in right-linear or left-linear form, the grammar itself is neither right-linear nor left-linear, and therefore is not regular. The grammar is an example of a linear grammar.

Definition (Linear Grammar): A *linear grammar* is a grammar in which at most one variable can appear on the right side of any production.

3.3.3 Right-Linear Grammars Generate Regular Languages

Linz Theorem 3.3 (Regular Languages for Right-Linear Grammars): Let $G = (V, T, S, P)$ be a right-linear grammar. Then $L(G)$ is a *regular language*.

Strategy: Because a regular language is any language accepted by a dfa or nfa, we seek to construct an nfa that simulates the derivations of the right-linear grammar.

The algorithm below incorporates this idea. It is based on the algorithm given on page 314 of the book: James L. Hein, *Theory of Computation: An Introduction*, Jones and Bartlett, 1996.

Algorithm to convert a regular grammar to an nfa

Start with a right-linear grammar and construct an equivalent nfa. We label the nfa's states primarily with variables from the grammar and label edges with terminals in the grammar or λ.

1. If necessary, transform the grammar so that all productions have the form $A \to x$ or $A \to xB$, where x is either a terminal in the grammar or λ.
2. Label the start state of the nfa with the start symbol of the grammar.
3. For each production $I \to aJ$, add a state transition (edge) from a state I to a state J with the edge labeled with the symbol a.
4. For each production $I \to J$, add a state transition (edge) from a state I to a state J with the edge labeled with λ.
5. If there exist productions of the form $I \to a$, then add a single new state symbol F. For each production of the form $I \to a$, add a state transition from I to F labeled with symbol a.
6. The final states of the nfa are F plus all I such there is a production of the form $I \to \lambda$.

End of algorithm

3.3.4 Example: Converting Regular Grammar to NFA

Construct an nfa for the following regular grammar G:

- $S \to aS \mid bI$
- $I \to a \mid aI$

The grammar is in the correct form, so step 1 of the grammar is not applicable. The following sequence of diagrams shows the use of steps 2, 3 (three times), 5, and 6 of the algorithm. Step 4 is not applicable to this grammar.

Note that $L(G) = L(a^*ba^*a)$.

3.3.5 Linz Example 3.5

This is similar to the example in the Linz textbook, but we apply the algorithm as stated above.

Construct an nfa for the regular grammar G:

- $V_0 \to aV_1$
- $V_1 \to abV_0 \mid b$

First, let's transform the grammar according to step 1 of the regular grammar to nfa algorithm above.

- $V_0 \to aV_1$
- $V_1 \to aV_2 \mid b$
- $V_2 \to bV_0$

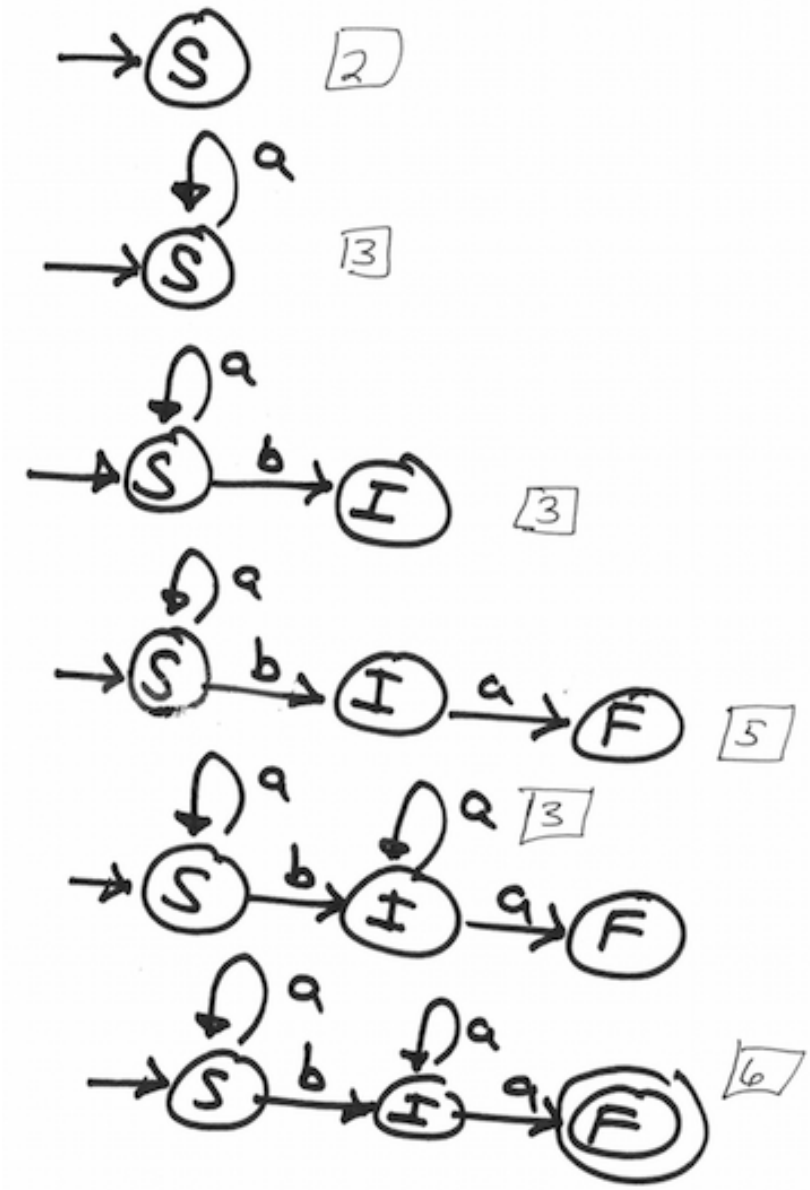
Applying steps 2, 3 (three times), 5, and 6 of algorithm as show below, we construct the following nfa. Step 4 was not applicable in this problem.

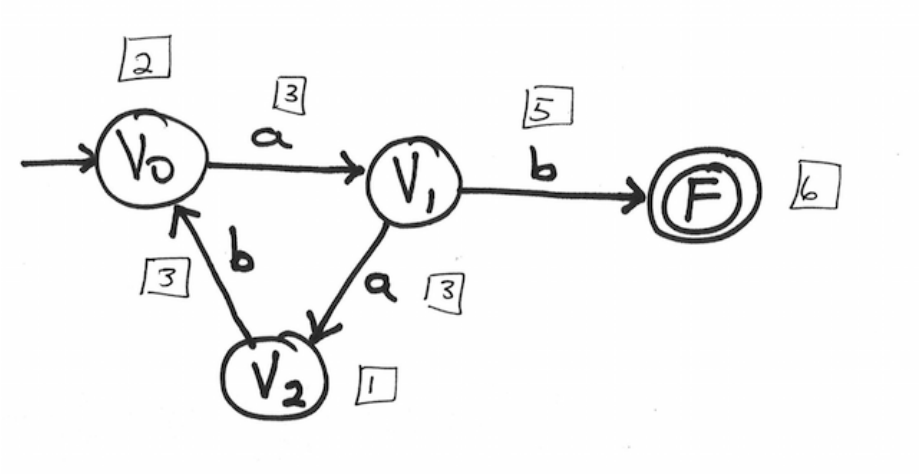
Note that $L(G) = L((aab)^*ab)$.

3.3.6 Right-Linear Grammars for Regular Languages

Linz Theorem 3.4 (Right-Linear Grammars for Regular Languages): If L is a regular language on the alphabet Σ, then there exists a right-linear grammar $G = (V, \Sigma, S, P)$ such that $L = L(G)$.

Strategy: Reverse the construction of an nfa from a regular grammar given above.


S
2
a
S
3
a
S
b
I
3
a
S
b
I
a
F
5
a
a
3
S
b
I
a
F
a
a
S
b
I
a
F
6



The algorithm below incorporates this idea. It is based on the algorithm given on page 312 of the Hein textbook cited above.

Algorithm to convert an nfa to a regular grammar

Start with an nfa and construct a regular grammar.

1. Relabel the states of the nfa with capital letters.
2. Make the start state label the start symbol for the grammar.
3. For each transition (edge) from a state I to a state J labeled with an alphabetic symbol a, add a production $I \to aJ$ to the grammar.
4. For each transition (edge) from a state I to a state J labeled with λ, add a production $I \to J$ to the grammar.
5. For each final state labeled K, add a production $K \to \lambda$ to the grammar.

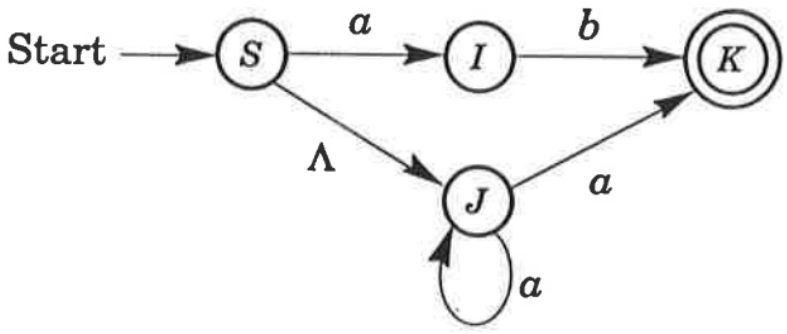
End of algorithm

3.3.7 Example: Converting NFA to Regular Grammar

Consider the following nfa (adapted from the Hein textbook page 313). (The Hein book uses Λ instead of λ to label silent moves and empty strings.)

We apply the steps of the algorithm as follows.

1. The nfa states are already labeled as specified.



2. Choose S as start symbol for grammar.
3. Add the following productions:
   - $S \to aI$
   - $I \to bK$
   - $J \to aJ$
   - $J \to aK$
4. Add the following production:
   - $S \to J$
5. Add the following production:
   - $K \to \lambda$

So, combining the above productions, we get the final grammar:

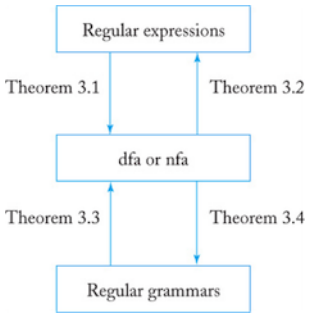
- $S \to aI \mid J$
- $I \to bK$
- $J \to aJ \mid aK$
- $K \to \lambda$

3.3.8 Equivalence Between Regular Languages and Regular Grammars

Linz Theorem 3.5 (Equivalence of Regular Languages and Left-Linear Grammars): A language L is *regular* if and only if there exists a left-linear grammar G such that $L = L(G)$.

Linz Theorem 3.6(Equivalence of Regular Languages and Right-Linear Grammars): A language L is regular if and only if there exists a regular grammar G such that $L = L(G)$.

The four theorems from this section enable us to convert back and forth among finite automata and regular languages as shown in Linz Figure 3.19. Remember that Linz Theorem 2.2 enabled us to translate from nfa to dfa.



Linz Fig. 3.19: Equivalence of Regular Languages and Regular Grammars