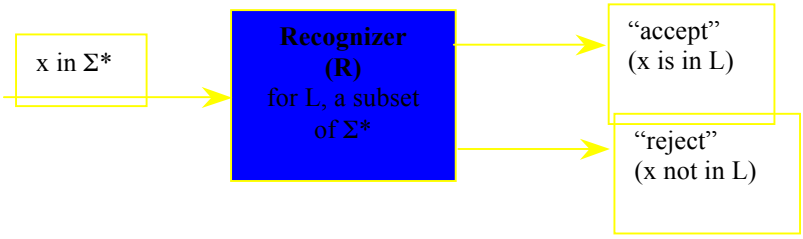
Finite State Automata

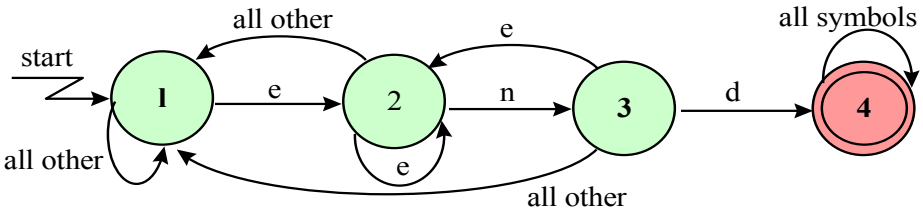
Our next series of definitions and results characterize the family of *Regular Languages*. Specifically, this family is exactly the set of languages that can be recognized or accepted by *Deterministic Finite Automata* (DFAs) and *Non-deterministic Finite Automata* (NFAs). DFAs and NFAs are examples of a more general class of formal language specifications called *recognizers*. A model of a typical Recognizer is illustrated in the figure below. A Recognizer takes as its only input some string x over its input alphabet, Σ. If $x \in L(R)$, the language recognized by R, then R must eventually halt and output *accept*. If $x \notin L(R)$, then R will have one of two behaviors, (a) it will halt and output *reject*, or (b) it will never halt.

For example, message decoding devices (e.g. communication protocols) and programming language compilers are examples of recognizers. In the former case, messages are symbol strings over some natural language alphabet, while inputs to compilers are strings (programs) over the alphabet of some programming language.



Example 9. To illustrate the recognizer concept, consider an algorithm for searching a text file for an occurrence of the string "end". The algorithm will "accept" the text file if it contains at least one occurrence of this word and will "reject" or "fail" otherwise. We model the algorithm with a state transition diagram (STD), where "states" (numbered circles) denote distinct configurations of the algorithm's "memory"(local variables) that define intermediate stages of "success" in making a final determination about the correctness or incorrectness of the input file. Transitions between states occur with the next character read from the input file. In the STD below,

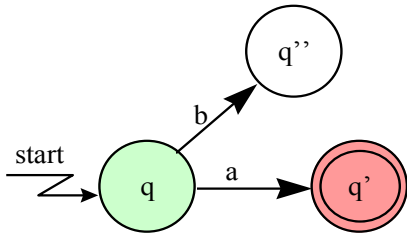
State 1. Defines the initial starting point of computation, and denotes a state of processing that implies no occurrence of the target word has been encountered, and last character read was not the first letter "e" of the target word.
States 2 and 3. Define intermediate states of processing where some proper prefix of the target word has just been encountered.
State 4. Denotes the accept state. That is, this state can only be reached if a complete occurrence of the target word has just been encountered. The algorithm could terminate with success at this point.



Our next definition formalizes the concept of finite state recognizer illustrated in the previous example.

Definition 9. A *Deterministic Finite Automata (DFA)* is a 5-tuple, $M = (Q, \Sigma, \delta, q_0, A)$ where:
Q = a finite non-empty set of states,
Σ = the input alphabet,
$q_0 \in Q$, is the initial state,
$A \subseteq Q$, is the (possibly empty) set of accepting states, and
$\delta : Q \times \Sigma \rightarrow Q$, a the transition function (a total function). If $q' = \delta(q, a)$, then "q" denotes the "current state" of M, "a" denotes the next symbol read from its input, and "q' " denotes the "next state" M enters after reading its input.

DFAs can be expressed in the form of *State Transition Diagrams* (STDs) using the conventions illustrated in the diagram below. Single-circles denote non-accepting states, double-circles accepting states, and directed arcs denote transitions on a given input symbol (arc label).

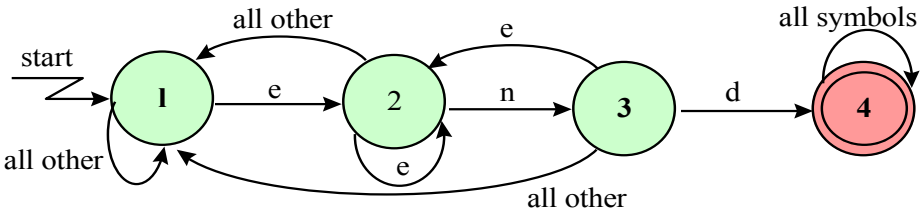


A DFA accepts (rejects) a given input string $x = a_1 a_2 ... a_k$ if, after reading each symbol of x in sequence, starting in its initial state q_0, it ends in an accepting(non-accepting) state. This notion is captured formally in the next definition of the language accepted by a DFA, M.

Definition 10. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA. We extend the domain of the function δ to strings of any length so that we can formally define the behavior of M, beginning from in any given state, q, on any input, x, to be the state M will be in after reading x. Specifically, we define $\delta^{\cdot}_{M}: Q \times \Sigma^* \rightarrow Q$ inductively as follows. The subscript M will be dropped whenever M is understood from context.

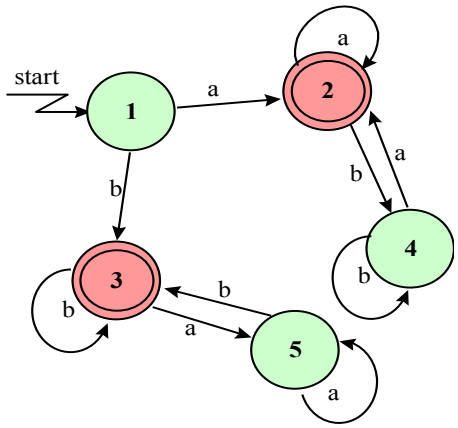
Basis: $\delta^{\cdot}_{M}(q, \lambda) = q$, for all $q \in Q$.
Inductive rule: $\delta^{\cdot}_{M}(q, x \cdot a) = \delta(\delta^{\cdot}_{M}(q, x), a)$, for all $q \in Q$, for all $a \in \Sigma$, and all $x \in \Sigma^*$.

Example 10. Referring once again to the DFA introduced in Example 9. We see that $\delta^{\cdot}_M(1, e) = \delta^{\cdot}_M(1, abee) = \delta^{\cdot}_M(3, enene) = 2$.



Definition 11. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA, then the language accepted or recognized by M, is the set $L(M) = \{ x \in \Sigma^* \mid \delta^{\cdot}_M(q_0, x) \in A \}$.

Example 11. Let $L = \{ x \in \{a,b\}^* \mid x \neq \lambda$ and x begins and ends with the same symbol $\}$



M = ({1,2,3,4,5},{a,b}, δ, 1, {2,3}) where δ is given by the *transition table* below.

δ_M	a	b
1	2	3
*2	2	4
*3	5	3
4	2	4
5	5	3

Exercise 6. Give an inductive definition of L(M), where M is the DFA given in Example 11.

Problem 1. Consider the following definition of *Generalized Inductive Definition for a set S.*
Let $A_1, A_2, \ldots, A_k$, for some $k \geq 1$, be sets defined inductively by:
Basis Rules: For each j, B_j, is a finite set defining the initial membership to A_j. That is, $B_j \subseteq A_j$, for $1 \leq j \leq k$.
Inductive Rules: A finite set of rules of the form*: if P(x) then f(x)* $\subseteq A_j$, where P(x) is any predicate defined for existing members of $A_1, A_2, \ldots, A_k$, and f(x) is a finite set.
Completion Rule: Nothing is a member of $A_1, A_2, \ldots, A_k$ that does not gain membership by finite application of the above rules. Finally, $S = A_j$, for some j, $1 \leq j \leq k$..

Show that if M is any DFA, then a generalized inductive definition of L(M) can be given in terms of the sets S_q, $q \in Q$, where $S_q = \{ x \mid \delta^*_M(q_0,x) = q \}$. Furthermore, show that such a definition can be given where all the inductive rules are of one of two forms: **(a)** if $x \in A_i$ then $xa \in A_j$, for some i,j, and $a \in \Sigma$, or **(b)** if $x \in A_i$ then $x \in A_j$.

Having defined the language accepted by a DFA, we can formally define the family of languages recognized by DFAs.

Definition 12. Let Σ be an alphabet and let $L \subseteq \Sigma^*$. Then L is said to be a *Regular language over* Σ if and only if there exists a DFA, $M = (Q, \Sigma, \delta, q_0, A)$, such that L = L(M).

Our first theorem describes an important sub-family of the Regular languages. The construction technique used to establish this result is as important as the result itself.

Theorem 1. Let F be any finite language over Σ, then F is Regular over Σ.

Proof. Two cases arise. $F = \Phi$ or $F \neq \Phi$. In both cases we must construct a DFA that accepts F. Then it follows directly by Definition 12 that F is Regular.

Case (a) $F = \Phi$. For this case we define $M = (\{q_0\}, \Sigma, \delta, q_0, \Phi)$ and $\delta(q_0,a) = q_0$, for all $a \in \Sigma$. It should be clear that $\delta^*(q_0, x) = q_0$, for all $x \in \Sigma^*$ (this can be proven by induction on |x| using the definition of δ^*). Thus $L(M) = \{x \mid \delta^*(q_0, x) \in A\} = \Phi$, since A is Φ.

Case (b). $F = \{x_1, x_2, \ldots, x_n\}$, for some $n \geq 1$. Let $m = \text{Max} \{ |x| \mid x \in F \}$. Then define $M = (Q, \Sigma, \delta, q_0, A)$ as follows. $Q = \{ q_x \mid x \in \Sigma^* \text{ and } |x| \leq m \} \cup \{\Omega\}$. $q_0 = q_\lambda$.
$A = \{q_x \mid x \in F\}$ and for all $a \in \Sigma$ and $q_x \in Q$, $\delta(q_x, a) = q_{xa}$, provided $|xa| \leq m$, else $\delta(q_x, a) = \Omega$.
Finally, $\delta(\Omega, a) = \Omega$, for all $a \in \Sigma$. By a simple inductive proof one can easily establish that $\delta^*(q_\lambda, x) = q_x$, provided $|x| \leq m$, and $\delta^*(q_\lambda, x) = \Omega$, otherwise. Thus $L(M) = \{x \mid \delta^*(q_0, x) \in A\} = \{x \mid \delta^*(q_0, x) = q_x, \text{ for some } x \in F\} = F$.

Exercise 7. Give a definition for the *minimal-state DFA* that accepts a finite set. How many states will such a machine have?

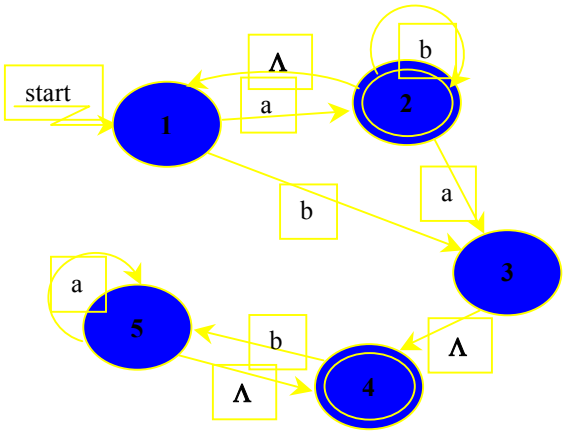
The next definition generalizes the notion of a DFA to one that exhibits unpredictable or "non-deterministic" behavior. That is, a FSA in which transitions from one state to another can occur "spontaneously" without reading any input, and/or in which transitions on the same symbol may leave the FSA in more than one possible next state. Furthermore, we may not always know exactly what state the FSA will be in when it is "turned on," so we will model this NFA as beginning in any one of several possible initial states. Finally, we will say that an NFA accepts its input string, x, if there exists a transition sequence that reads all of x and leaves M in an accepting state. This is formalized in our next sequence of definitions.

Definition 13. A *Non-deterministic Finite Automata (NFA)* is a 5-tuple, $M = (Q, \Sigma, \delta, Q_0, A)$ where:
Q = a finite non-empty set of states,
Σ = the input alphabet,
$Q_0 \subseteq Q$, is a non-empty set of initial states,
$A \subseteq Q$, is the (possibly empty) set of accepting states, and
$\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow Q$, is the transition relation (a partial relation).

Read transitions are defined as transitions where, $\delta(q, a) \subseteq Q$, is a non-empty set, for a $\in \Sigma$. Note that if $\delta(q, a) = \Phi$, then M cannot make any transition in state q that reads an "a" (it may, however, be able to leave its current state by a spontaneous transition described below.)

S*pontaneous transitions* are transitions M can make without reading from its input. Spontaneous transitions are expressed by, $\delta(q, \Lambda) \subseteq Q$ being a non-empty set. Note we never allow $q \in \delta(q, \Lambda)$, for any q in Q. Therefore $\delta(q, \Lambda)$ can only contain states $p \neq q$. By choosing one of these other states, M can continue to operate without reading from its input.

Example 12. An NFA.



The question we consider next is "What language does an NFA accept?" In Example 12, the behavior of M on input abbb can leave M in any one of the following states {2,3,4,5}. Thus, there can be more than one computation of M on the same input, some computations may accept, and some may not. To know definitely whether an NFA accepts its input or not, one must consider all possible sequences of transitions on the same input. M accepts its input if and only

if at least one of these computations reads all of the input and allows the NFA to enter an accepting state. Before formalizing the concepts of the language accepted by an NFA, we formally introduce the notion of "configuration" and "computation" of M.

Definition 14. Let M = (Q, Σ , δ , Q_0 , A) be an NFA. We define a *configuration of M* to be a pair (q, x), where q denotes M's current state and x ∈ Σ* denotes the input remaining to be read. A configuration is said to be an initial configuration if q ∈ Q_0 , and a terminating configuration if x = λ. We define a configuration to be an accepting configuration if it is a terminating configuration for which q is an accepting state. Finally, we define a configuration to by halting, if no next configuration is possible.

The behavior of M is defined in terms of the *move relation, denoted by* $_M\Rightarrow$, defined on configurations as follows:

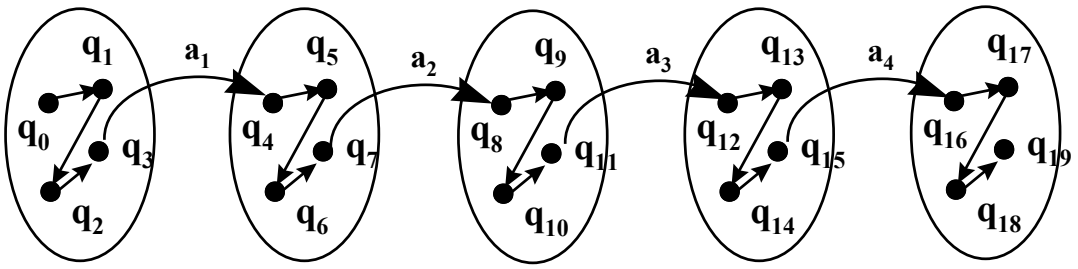
(q, x) $_M\Rightarrow$ (q',x) iff q' ∈ δ(q,**Λ**) and
(q, ax) $_M\Rightarrow$ (q',x) iff q' ∈ δ(q,a).

Observe that $_M\Rightarrow$ is not reflexive. That is, (C, C) ∉$_M\Rightarrow$, for all configurations, C. We will use the notation, $_M\Rightarrow$* to denote the reflexive-transitive closure of $_M\Rightarrow$, and $_M\Rightarrow$+ to denote the transitive closure of $_M\Rightarrow$. The subscript "M" will be dropped when M is clear from context.

A computation of M is a sequence of configurations C_0, C_1 , ... , C_n , n ≥ 0, where C_0 is an initial configuration, C_n is a terminating configuration and C_i $_M\Rightarrow$ C_{i+1}. A sub-computation of M is any sequence of configurations satisfying C_i $_M\Rightarrow$+ C_j.

We can now formally define the language accepted by an NFA.

Definition 15. Let M = (Q, Σ , δ, Q_0 , A) be an NFA. Then the *language accepted by M* is given by, L(M) = { x ∈Σ* | There exists a computation: (s, x) $_M\Rightarrow$* (f, λ), where s ∈ Q_0 and f ∈ A}.



$(q_0, a_1a_2a_3a_4)$ =>+ (q_{19}, λ)

The diagram above illustrates a "typical" NFA accepting computation. Observe that M may perform several **Λ**-transitions before and after a read transition. In this computation, M begins in one of its initial states, q_0, and terminates in state q_{19}. M accepts if any one of the states, q_{16}-q_{19}, is an accept state.

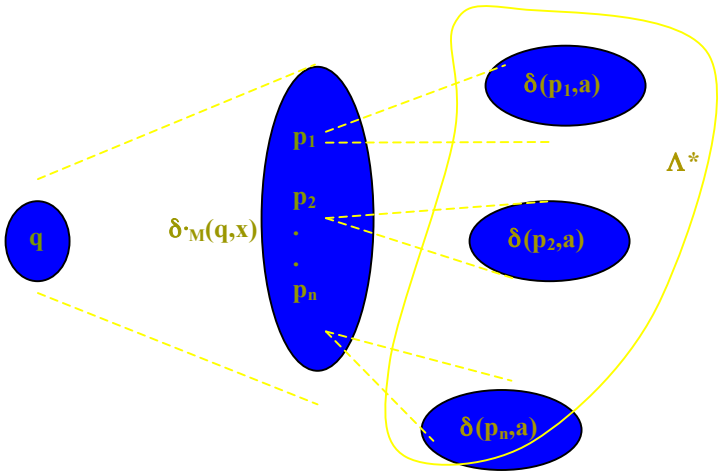
What we now want to show is that NFAs and DFAs accept exactly the same family of languages, that is, the regular languages. To this end, we define a transition function on NFAs that tells us the set of all possible states an NFA can reach by a subcomputation that reads x $\in\Sigma^*$ beginning from the given state. We will use this function to define the behavior of a DFA that accepts the same language as a given NFA. The implication of this result is that NFAs recognize exactly the same family of languages that DFAs recognize.

Definition 16. Let M = (Q, Σ , δ , Q_0 , A) be an NFA. Define

Λ^*: Q → 2^Q , the $\Lambda^*(q) = \{ q' \mid (q, \lambda) \ _M\Rightarrow^* (q',\lambda) \}$.
Λ^*: $2^Q \rightarrow 2^Q$, by $\Lambda^*(S) = \cup_{q\in S} (\Lambda^*(q))$, for every $S \subseteq Q$.
δ^*_M: $Q \times \Sigma^* \rightarrow 2^Q$, is defined inductively as follows
Basis: $\delta^*_M(q, \lambda) = \Lambda^*(q)$, for all $q \in Q$.

Inductive rule: $$\delta^*_M(q, xa) = \Lambda^*\left(\bigcup_{p\in\delta^*_M(q,x)} \delta(p, a) \right)$$

The figure below depicts the meaning of the definition of δ^*_M.



Corollary 16-1: For all $q \in Q$ and $x \in\Sigma^*$, $q' \in \delta^*_M(q,x)$ if and only if $(q,x) \Rightarrow^* (q',\lambda)$.

Proof. By induction on |x|.
Basis: x = λ.
$q' \in \delta^*_M (q, \lambda)$ iff $q'\in \Lambda^*(q)$: by basis of Definition 16 for δ^*_M
$q' \in \Lambda^*(q)$ iff $(q,\lambda) \Rightarrow^* (q',\lambda)$: by Definition 16 for Λ^*.
Thus the corollary holds for |x| = 0.

IH: Assume for $x \in\Sigma^n$, with $n \geq 0$, that $q' \in \delta^*_M(q,x)$ iff $(q,x) \Rightarrow^* (q',\lambda)$.
Consider $q' \in \delta^*_M(q, x\cdot a)$ for some $a \in\Sigma$ and $x \in\Sigma^n$. Then by Definition 16 for δ^*_M ,
$q' \in \delta^*_M(q,x\cdot a)$ iff $q' \in \Lambda^*(S_a)$, where $S_a = \cup_{p\in D} (\delta(p,a))$, and $D = \delta^*_M(q,x)$. But $q' \in \Lambda^*(S_a)$ iff there is $p \in \delta^*_M(q,x)$ and $p' \in \delta(p,a)$ such that $q' \in \Lambda^*(p')$. But this holds iff
[1] $(q,x) \ _M\Rightarrow^* (p,\lambda)$: by IH,

[2] $(p,a) \underset{M}{\Rightarrow} (p', \lambda)$: by definition of $\underset{M}{\Rightarrow}$ and the assumption that $p' \in \delta(p,a)$,
[3] $(p',\lambda) \underset{M}{\Rightarrow}^* (q', \lambda)$: by definition of Λ^*.
[4] But $(q,x) \underset{M}{\Rightarrow}^* (p,\lambda)$ iff $(q,xa) \underset{M}{\Rightarrow}^* (p,a)$: by **an exercise left to the reader.**

Therefore, by definition of $\underset{M}{\Rightarrow}^*$, [4][2] and [3] hold iff $(q,x\cdot a) \underset{M}{\Rightarrow}^* (q', \lambda)$. QED

Corollary 16-2: Let $M = (Q, \Sigma, \delta, Q_0, A)$ be an NFA. Then $x \in L(M)$ iff $\delta^*_M(q, x) \cap A \neq \Phi$, for some $q \in Q_0$.

Proof. If $x \in L(M)$, then for some $q \in Q_0$ and $f \in A$, $(q,x) \Rightarrow^* (f,\lambda)$; this follows from Definition 15. But then by Corollary 16-1, $f \in \delta^*_M(q,x)$. Thus $\delta^*_M(q,x) \cap A \neq \Phi$.

Exercise 8. Complete the proof of Corollary 16-2 by showing that if $\delta^*_M(q,x) \cap A \neq \Phi$, for some $q \in Q_0$, then $x \in L(M)$.

The next lemma establishes a simple property of the Λ^* operator that will be useful in establishing the equivalence between DFAs and NFAs.

Lemma 2. Let $S_i \subseteq Q$, $1 \le i \le n$, be sets of states of some NFA, $M = (Q, \Sigma, \delta, Q_0, A)$. Then

$$\Lambda^*\left(\bigcup_{i=1}^{n} S_i\right) = \bigcup_{i=1}^{n} \Lambda^*(S_i)$$

Proof. Let q' be an element of the left side of this equality. Then by definition of Λ^*, $q' \in \Lambda^*(q)$, for some q in $\cup_{1 \le i \le n} S_i$. But this implies that there is a k, $1 \le k \le n$, such that $q' \in \Lambda^*(q)$, for some q in S_k. But this is a restatement of the definition for $q' \in \Lambda^*(S_k) \subseteq \cup_{1 \le i \le n} \Lambda^*(S_i)$ = the right side. The steps of this argument can be reversed and it follows that the two expressions are equal.

Theorem 2. Every NFA, $M = (Q, \Sigma, \delta, Q_0, A)$, is equivalent to the DFA $M_p = (Q_p, \Sigma, \delta_p, \alpha, A_p)$, called the *powerset machine*, where
$Q_p = 2^Q$ (the power set of Q)
$\alpha = \Lambda^*(Q_0) \in Q_p$
$A_p = \{ S \in Q_p \mid S \cap A \neq \Phi \}$
$\delta_p(S, a) = \cup_{q \in S} (\delta^*_M(q,a))$, for all $S \in Q_p$ and $a \in \Sigma$.

Proof. Let $\delta\cdot_p$ be the extension of δ_p. We shall show by induction on |x| that $f \in \delta\cdot_p(\alpha, x)$ iff there exists $q \in Q_0$ such that $(q, x) \underset{M}{\Rightarrow}^* (f, \lambda)$. From this it follows by definitions 11 and 15 that $x \in L(M_p)$ if and only if $x \in L(M)$.

Basis: $x = \lambda$ ($|x| = 0$). If $f \in \delta\cdot_p(\alpha,\lambda) = \alpha = \Lambda^*(Q_0)$, then there is $q \in Q_0$ for which $f \in \Lambda^*(q)$. This follows from the definition of Λ^*. Thus by definition of $\Lambda^*(q)$, $(q, \lambda) \underset{M}{\Rightarrow}^* (f, \lambda)$.
Conversely, if $(q, \lambda) \underset{M}{\Rightarrow}^* (f, \lambda)$, for some $q \in Q_0$, then by Definition 16, $f \in \Lambda^*(q) \subseteq \Lambda^*(Q_0) = \alpha$

$= \boldsymbol{\delta}^{\cdot}_{\mathbf{p}}(\alpha,\lambda)$.

Induction Case: Assume for all $x \in \Sigma^n$, for some $n \geq 0$, that $f \in \boldsymbol{\delta}^{\cdot}_{\mathbf{p}}(\alpha, x)$ iff there is $q \in Q_0$ such that $(q, x) \underset{M}{\Rightarrow}^* (f, \lambda)$. Consider a string of the form $x \cdot a \in \Sigma^{n+1}$ and suppose $f \in \boldsymbol{\delta}^{\cdot}_{\mathbf{p}}(\alpha, x \cdot a)$. Then by definition of $\boldsymbol{\delta}^{\cdot}_{\mathbf{p}}$ and δ_p (above) we have $f \in \delta_p(S, a) = \cup_{q \in S} (\boldsymbol{\delta}^*_{\mathbf{M}}(q,a))$, where $S = \boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x)$. This holds iff there is $s \in \boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x) = S$ for which $f \in \boldsymbol{\delta}^*_{\mathbf{M}}(s,a)$. Now by our induction hypothesis, $s \in \boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x)$ iff there is $q \in Q_0$ such that $(q, x) \underset{M}{\Rightarrow}^* (s, \lambda)$. By a previous exercise, $(q, x) \underset{M}{\Rightarrow}^* (s, \lambda)$ if and only if $(q, x \cdot a) \underset{M}{\Rightarrow}^* (s, a)$. By Corollary 16-1, $f \in \Delta_M(s,a)$ iff $(s,a) \underset{M}{\Rightarrow}^* (f, \lambda)$. Putting this all together we have $f \in \delta_p(\alpha, x \cdot a)$ iff there is $q \in Q_0$ such that $(q, x \cdot a) \underset{M}{\Rightarrow}^* (s, a) \underset{M}{\Rightarrow}^* (f, \lambda)$. This concludes the proof of our claim that $f \in \boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x)$ iff there is $q \in Q_0$ such that $(q, x) \underset{M}{\Rightarrow}^* (f, \lambda)$.

To complete the proof of theorem we observe that $x \in L(M_p)$ iff $\boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x) \cap A \neq \Phi$. So, let $f \in \boldsymbol{\delta}^*_{\mathbf{p}}(\alpha, x) \cap A$, then by the property we just proved, there is $q \in Q_0$ such that $(q, x) \underset{M}{\Rightarrow}^* (f, \lambda)$. But this holds iff $x \in L(M)$. QED

Before applying Theorem 1 to an example, we need a more convenient expression for computing δ_p. We state and prove this alternative definition with our next lemma.

Lemma 3. For all $S \subseteq Q$, and $a \in \Sigma$,

$$\delta_p(S,a) = \Lambda^*\left(\bigcup_{p \in \Lambda^*(S)} \delta_M(p,a)\right)$$

Proof.

$$\delta_p(S,a) = \bigcup_{q \in S} \delta^*_M(q,a), \text{ by definition of } \delta_p \text{ in Theorem 1.}$$

$$= \bigcup_{q \in S} \Lambda^*\left(\bigcup_{p \in \delta^*_M(q,\lambda)} \delta_M(p,a)\right), \text{ by definition of } \delta^*_M$$

$$= \bigcup_{q \in S} \left(\bigcup_{p \in \Lambda^*(q)} \Lambda^*(\delta_M(p,a)) \right), \text{ by definition of } \Lambda^* \text{ and Lemma 2}$$

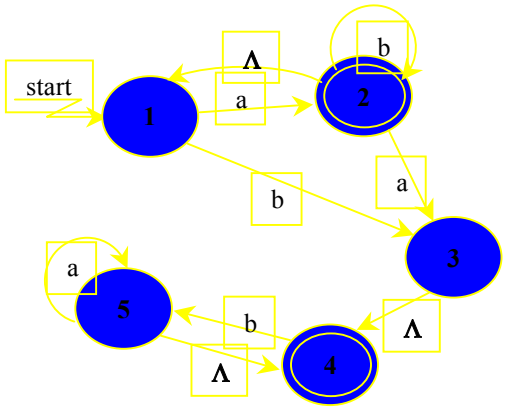
$$= \bigcup_{p \in \Lambda^*(S)} \Lambda^*(\delta_M(p,a)), \text{ by definition of } \Lambda^*(S)$$

$$= \Lambda^*\left(\bigcup_{p \in \Lambda^*(S)} \delta_M(p,a) \right), \text{ by Lemma 2. QED.}$$

Example 13. Let's convert the NFA of Example 12 (diagram is reproduced below) to its equivalent DFA. To make it easier to apply the construction of M_p as described in the above theorem, we first construct the transition table for the NFA, M.

δ_M	a	b	Λ	Λ^*
→1	{2}	{3}	Φ	{1}
*2	{3}	{2}	{1}	{1,2}
3	Φ	Φ	{4}	{3,4}
*4	Φ	{5}	Φ	{4}
5	{5}	Φ	{4}	{4,5}

Transition Table for NFA, M



The initial state α of the DFA M_p is $\Lambda^*(Q_0) = \Lambda^*(\{1\}) = \{1\}$.

δ_p	a	b
→{1}	{1,2}	{3,4}
*{1,2}	{1,2,3,4}	{1,2,3,4}
*{3,4}	Φ	{4,5}
*{4,5}	{4,5}	{4,5}
*{1,2,3,4}	{1,2,3,4}	{1,2,3,4,5}
*{1,2,3,4,5}	{1,2,3,4,5}	{1,2,3,4,5}
Φ	Φ	Φ

Transition Table for DFA, M_p

Observe that only 7 out of a possible $2^5 = 32$ states of M_p are actually *reachable* from its initial state, α.

Exercise 9. Compute δ_p for some of the unreachable states of M_p by filling in the table below. **Hint:** use the result of Lemma 3. Which of these states are accept states of M_p?

δ_p	a	b
{1,5}		
{2,4}		
{1,4}		
{3,4,5}		
{1,2,3}		
{1,3,5}		
{3}		

The Minimal State DFA

We have characterized Regular languages in terms of NFAs (DFAs). In this section we begin to address for formal systems an issue that is always of concern for real programs, namely, "how can we make programs more efficient?" Specifically, we consider the problem of reducing the number of states in a DFA, $M = (Q, \Sigma, \delta, q_0, A)$. In more colloquial terms, "can I design a DFA with fewer states that will do the same thing?" We know intuitively that not all equivalent DFAs have the same number of states. This is demonstrated by trivial examples. Consider M and M' illustrated in the figure below. It is clear that $L(M) = L(M') = \Sigma^*$. In Figure A M has two reachable states, both of which are accept states. Clearly, this machine accepts all strings and is equivalent to the one-state DFA, M'; observe that M' can be obtained from M by simply redirecting the transition, $\delta(1,b)$, from state 2 back to state 1, and eliminating 2 which, after this change, would no longer be reachable. In Figure B, we see that state 2 is unreachable in M. Thus the smaller, more efficient, M', can be constructed by removing state 2.

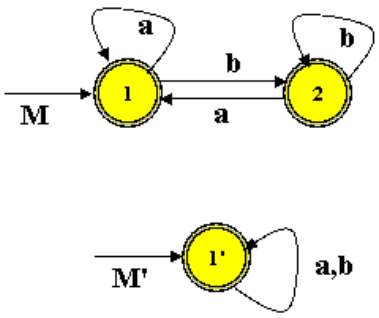


Figure A.

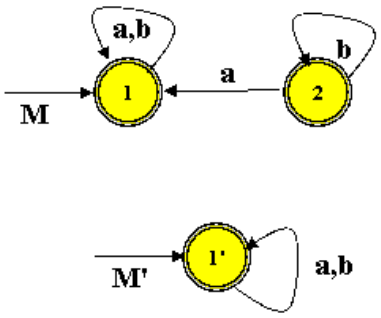


Figure B.

These simple examples introduce two key ideas that will be exploited in the state minimization algorithm we present later: **(a)** the elimination of *unreachable states*, and **(b)** removing redundant or *indistinguishable states*. The first concept is obvious. The second notion is vaguer and needs a formal introduction.

For any DFA, M, and any state $q \in Q$, we can define two sets of strings determined by q. The first set we shall denote as $R_q = \{ x \mid \delta^*(q_0,x) = q \}$. This is just the strings that leave M in state q. The second set determined by q is S_q and is given by $S_q = \{y \mid \delta^*(q, y) \in A \}$. These are the strings that would allow M to accept, once M has reached state q. Sets Rq and Sq are illustrated graphically in Figure C.

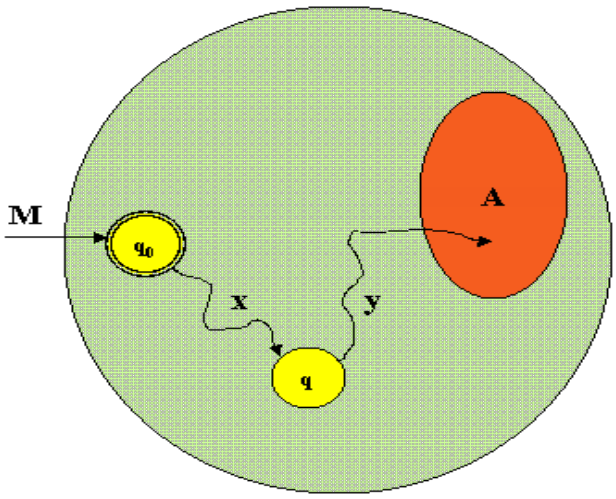


Figure C. A typical x ∈ R_q and a typical y ∈S_q.

Given these sets, the following is true about L(M).

$$L(M) = \bigcup_{q \in Q} R_q S_q$$

Now suppose that $S_q = S_{q'}$ for two distinct state q and q'. To see that this is possible, refer to our machine M in Figure A. Clearly $S_1 = S_2 = \Sigma^*$. However, assuming q and q' are both reachable states of M, then it is never true that $R_q = R_{q'}$, in fact $R_q \cap R_{q'} = \Phi$. For suppose that $x \in R_q \cap R_{q'}$. Because δ^*_M is a function we have $\delta^*_M(q_0,x) = q = q'$, contradicting our assumption that q and q' are distinct. The consequence of the fact that $S_q = S_{q'}$ is that the equation for L(M) can now be

$$L(M) = \left(\bigcup_{p \in Q - \{q,q'\}} R_p S_p \right) \cup \left(R_q \cup R_{q'} \right) S_q$$

written as:

This says that, with respect to what M will eventually accepts, there is no difference in the behavior of M once it reaches q or q'. We say that q and q' are *indistinguishable*. Restating, we can merge q and q' into one state by "rewiring" M in the following way. If $p \neq q'$ is any state in Q and $\delta(p, a) = q'$, then redefine the transition to go to q instead; that is, redefine $\delta(p,a) = q$. Then, remove q' from M and any other states that might become unreachable as a result. If one of q and q' is q_0, then remove the state that is not q_0. After this rewiring, $R_q(new) = R_q(old) \cup R_{q'}(old)$. $S_q(new) = S_q(old)$ and $L(M_{new}) = L(M_{old})$.

Definition BX-1. So, more formally, we say that q and q' are ***indistinguishable states*** if and only if ($S_q = S_{q'}$), where S_q and $S_{q'}$ are defined as specified above.

Now that we know how to reduce the size of a DFA, the next problem we face is how to identify pairs of indistinguishable states. It turns out that ***indistinguishable*** is an equivalence relations on the set of reachable states of a DFA. We develop this fact formally in the following discussion.

Definition BX-2. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA, and let $q_1 , q_2 \in Q$. Then the relation $\equiv$ is defined on Q as follows: $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(q_1, z) \in A \Leftrightarrow \delta^*_M(q_2, z) \in A]$; this is equivalent to saying that q_1 , q_2 are indistinguishable.

Corollary-BX2. $\equiv$ is an equivalence relation on Q. Furthermore, $q_1 \equiv q_2$ if and only if $\forall x \in \Sigma^*$, $\delta^*_M(q_1, x) \equiv \delta^*_M(q_2, x)$.

Proof. It is trivial to show that $\equiv$ is an equivalence relation on Q. This we leave to the reader. We must now show that any pair of states, reachable from q_1 and q_2, must also be indistinguishable.

By Definition BX-2, $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$.
This holds if and only if $\forall x,y \in \Sigma^* [\delta^*_M(q_1,xy) \in A \Leftrightarrow \delta^*_M(q_2,xy) \in A]$ if and only if $\forall x \in \Sigma^*$ $[\forall y \in \Sigma^* [\delta^*_M(\delta^*_M(q_1,x),y) \in A \Leftrightarrow \delta^*_M(\delta^*_M(q_2,x),y) \in A]]$ if and only if
$\forall x \in \Sigma^* [\delta^*_M(q_1,x) \equiv \delta^*_M(q_2,x)]$. QED

The next Theorem BX-1 gives us an iterative algorithm for computing the relation $\equiv$ on the state set of a given DFA, M. It also establishes that the minimal state DFA accepting L(M) can be derived from $\equiv$ by treating equivalence classes $[q]_.$ as states of this machine.

Theorem BX-1. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA where all states in Q are reachable. For all $k \geq 0$ define the relation $\equiv_k$ on Q as follows: $q_1 \equiv_k q_2$ if and only if $\forall z \in \Sigma^{*k} [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$. (We say that q_1 and q_2 are *indistinguishable by strings of length k or less*.) Then

(a) $\equiv_0 = \{Q-A, A\}$ and for each $k \geq 0$, $q_1 \equiv_{k+1} q_2$ if and only $q_1 \equiv_k q_2$ and if for every $a \in \Sigma$, $\delta(q_1,a) \equiv_k \delta(q_2,a)$; and

(b) for some $k \leq n-2$, where n is the number of states in M, $\equiv = \equiv_k$; and finally,

(c) $L(M) = L(M_L)$, where $M_L = (Q', \Sigma, \delta', [q_0]_. , A')$, where
$Q' = \{ [q]_. \mid q \in Q\}$,
$A' = \{ [q]_. \mid q \in A\}$, and
for all $a \in \Sigma$, $\delta'([q]_., a) = [\delta(q,a)]_.$.

Proof(a). Applying the definition for k = 0, we see that $q_1 \equiv_0 q_2$ if and only if $[q_1 = \delta^*_M(q_1,\lambda) \in A \Leftrightarrow q_2 = \delta^*_M(q_2,\lambda) \in A]$. Thus $\equiv_0 = \{Q-A, A\}$. Now consider $q_1 \equiv_{k+1} q_2$.

By definition of $\equiv_{k+1}$, $\forall z \in \Sigma^{*(k+1)} [\delta^*_M(q_1, z) \in A \Leftrightarrow \delta^*_M(q_2, z) \in A]$, then $\forall z \in \Sigma^{*k} [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$. Thus $q_1 \equiv_k q_2$. But because $\Sigma^{*(k+1)} = \Sigma\Sigma^{*k}$, $q_1 \equiv_{k+1} q_2$ holds if and only if $\forall a \in \Sigma [\forall z \in \Sigma^{*k} [\delta^*_M(q_1, az) \in A \Leftrightarrow \delta^*_M(q_2, az) \in A]]$. And this holds if and only if $\forall a \in \Sigma [\forall z \in \Sigma^{*k} [\delta^*_M(\delta(q_1,a),z) \in A \Leftrightarrow \delta^*_M(\delta(q_2,a),z) \in A]$. And finally this holds if and only if $\forall a \in \Sigma$ $[\delta(q_1,a) \equiv_k \delta(q_2,a)]$.

Proof (b). We first show that if $\equiv_k = \equiv_{k+1}$, then $\equiv_k = \equiv$. It should be clear that $q_1 \equiv q_2$ if and only if for all $k \geq 0$, $q_1 \equiv_k q_2$. What we will show is that if $\equiv_k = \equiv_{k+1}$, then $\equiv_k = \equiv_{k+n}$, for all $n \geq 1$. This is easily established by induction on n. The basis (n = 1) is given by assumption. Now suppose $q_1 \equiv_{k+n+1} q_2$. Then by part (a) applied to $\equiv_{k+n+1}$ we have, $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_{k+n} q_2$ and for all $a \in \Sigma$ [$\delta(q_1,a) \equiv_{k+n} \delta(q_2,a)$]. But by our induction hypothesis, $\equiv_k = \equiv_{k+n}$, and we have $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_k q_2$ and for all $a \in \Sigma$ [$\delta(q_1,a) \equiv_k \delta(q_2,a)$]. But this is the same as saying $q_1 \equiv_{k+1} q_2$. And then, since $\equiv_k = \equiv_{k+1}$, it follows that $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_k q_2$. Thus $\equiv_k = \equiv_{k+n+1}$ and the induction is complete.

To finish (b) suppose M has n states and assume both Q-A and A are non-empty. Then by part (a), $\equiv_0 = \{ Q\text{-}A, A\}$. Suppose that $\equiv_0 \neq \equiv_1$, then $\equiv_1$ must have at least one more equivalence class than $\equiv_0$ (at least 3). Furthermore, because $q_1 \equiv_{k+1} q_2$ implies $q_1 \equiv_k q_2$, then at least one class of $\equiv_1$ was formed by splitting one of the classes of $\equiv_0$. Thus if we consider the progression of equivalence relations, $\equiv_0, \equiv_1, \ldots, \equiv_k$, where $|\equiv_0| = 2$ and $|\equiv_{j+1}| \geq |\equiv_j|+1$, then since the $|\equiv_k|$ cannot exceed the number of states of M, it follows that $k \leq n\text{-}2$ and $\equiv_k = \equiv_{k+1}$. Thus for any $j > n\text{-}2$, it must follow that $\equiv_j = \equiv_k = \equiv$.

Proof(c). The first point to observe is that M_L is a well-defined DFA, not NFA. The only possible challenge to this statement might arise from the definition of δ'. One must just realize that this is over states that represent equivalence classes and any representative of an equivalence class can be used to define the transition from that state to the states associated with subsequent equivalence classes.

STATE MINIMIZATION ALGORITHM

Let M = (Q, Σ, δ, q_0, A) be a DFA. The minimal state DFA equivalent to M can be computed in the following way.

Step 1: Eliminate the unreachable states of M to obtain M' = (Q', Σ, δ, q_0, A'). The only difference between M and M' will be that $Q' \subseteq Q$ and $A' \subseteq A$.

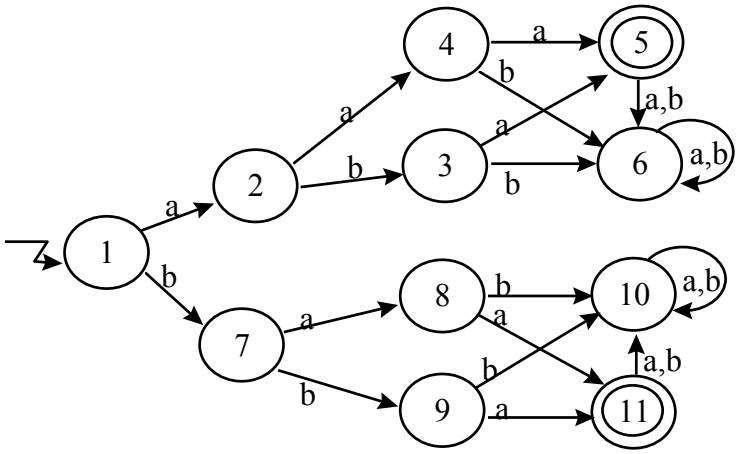
Step 2: If A' = Φ, then L(M) = L(M') = Φ, and both are equivalent to the 1-state DFA, M'' = ({q_0}, Σ, δ'', q_0, Φ), where $\delta''(q_0,a) = q_0$, for all $a \in \Sigma$. On the other hand, if A' = Q', then L(M) = L(M') = Σ*, and both are equivalent to M'' = ({q_0}, Σ, δ'', q_0, {q_0}), where $\delta''(q_0,a) = q_0$, for all $a \in \Sigma$.

Step 3: If A' ≠Φ and Q'-A' ≠Φ, then do the following.

For k = 0, 1, … , compute $\equiv_k$ until $\equiv_k = \equiv_{k+1}$. The minimal state DFA equivalent to M' is the machine, M_L = (Q', Σ, δ', $[q_0]_.$, A') , where Q' = { $[q]_.$ | $q \in Q$}, A' = { $[q]_.$ | $q \in A$}, and for all $a \in \Sigma$, $\delta'([q]_., a) = [\delta(q,a)]_.$. Use $\equiv_k$ as ≡.

Example 30. Constructing the minimal-state DFA for a given regular language L.

Consider the DFA, M, shown below for L(M) = { aaa, aba, baa, bba }.



$\equiv_0$ = { A:[1, 2, 3, 4, 6, 7, 8, 9, 10], B:[5, 11] }

	A									B	
δ	1	2	3	4	6	7	8	9	10	5	11
a	A	A	B	B	A	A	B	B	A	A	A
b	A	A	A	A	A	A	A	A	A	A	A

$\equiv_1$ = { A:[1, 2, 6, 7, 10], C:[3, 4, 8, 9], B:[5, 11] }

	A					C				B	
δ	1	2	6	7	10	3	4	8	9	5	11
a	A	C	A	C	A	B	B	B	B	A	A
b	A	C	A	C	A	A	A	A	A	A	A

$\equiv_2$ = { A:[1, 6, 10], D:[2, 7], C:[3, 4, 8, 9], B:[5, 11] }

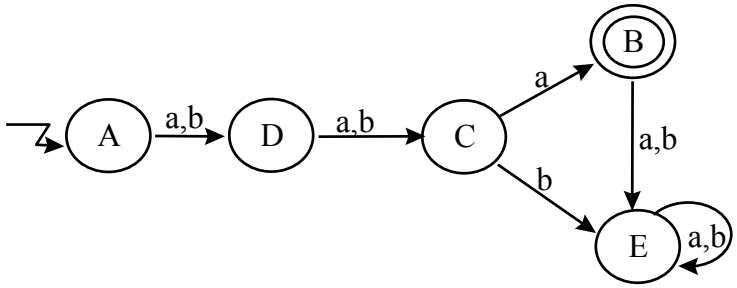
	A			D		C				B	
δ	1	6	10	2	7	3	4	8	9	5	11
a	D	A	A	C	C	B	B	B	B	A	A
b	D	A	A	C	C	A	A	A	A	A	A

$\equiv_3$ = { A:[1], E:[6, 10], D:[2, 7], C:[3, 4, 8, 9], B:[5, 11] }

	A	E		D		C				B	
δ	1	6	10	2	7	3	4	8	9	5	11
a	D	E	E	C	C	B	B	B	B	E	E
b	D	E	E	C	C	E	E	E	E	E	E

The reader can verify that $\equiv_3 = \equiv_4$. Thus $\equiv_3$ defines the states of the minimal DFA accepting L. In fact, the transition table constructed for $\equiv_3$ above is the transition function of the minimal DFA. The accepting state(s) of the minimal DFA are the classes consisting of accepting states of the original DFA, M. In our example, this is class B. The initial state of the minimal DFA is the

class containing the original initial state. In the example, this is A. The state transition diagram for the minimal DFA is shown below.



Hughes preferred approach

Another technique to attack this minimization is to write down a lower diagonal matrix of n-1 columns and n-1 rows, where n is the number of states in the original automaton. The columns are labeled 1..n-1; the rows are labeled 2..n. We refer to an entry or slot in the matrix as (i,j) if it is the ith column, j-th row. Our goal is to place in X in any slot q_i, q_j where the states are distinguishable.

Basis:
for i=1 to n-1 { for j=2 to n {set slot i,j to contain an X if one of q_i, q_j is final and the other is non-final} }

for i=1 to n-1 { for j=2 to n { for each $a \in \Sigma$ write down the state pair $(\delta(q_i,a), \delta(q_j,a))$ in slot i,j except when $\delta(q_i,a) = \delta(q_j,a)$ or $(\delta(q_k,a), \delta(q_m,a)) = (i,j)$ or $(\delta(q_k,a), \delta(q_m,a)) = (j,i)$; if any pair is associated with a slot having an X, reset the i,j slot to also have only an X } }

Induction:
for i=1 to n-1 { for j=2 to n {set slot i,j to contain an X if it contains any pair (k,m) where slot k,m contains an X } }

The above converges when an induction step makes no changes to the matrix. This takes at most n-1 inductive steps (Why?).

A naïve analysis concludes that each of the n-1 iterations requires order n^2 steps. As I've laid it out, that is true and so the algorithm is n^3, but there is a way to implement and analyze this that shows it can be done as an order n^2 algorithm and that is optimal (How and Why?).

Example 30 redone

Basis

2	**(2,4) (7,3)**								
3	**X**	**X**							
4	**X**	**X**							
5	**X**	**X**	**X**	**X**					
6	**(2,6) (7,6)**	**(4.6) (3,6)**	**X**	**X**	**X**				
7	**(2,8) (7,9)**	**(4,8) (3,9)**	**X**	**X**	**X**	**(6,8) (6,9)**			
8	**X**	**X**	**(5,11) (6,10)**	**(5,11) (6,10)**	**X**	**X**	**X**		
9	**X**	**X**	**(5,11) (6,10)**	**(5,11) (6,10)**	**X**	**X**	**X**		
10	**(2,10) (7,10)**	**(4,10) (3,10)**	**X**	**X**	**X**		**(8,10) (9,10)**	**X**	**X**
11	**X**	**X**	**X**	**X**		**X**	**X**	**X**	**X**	**X**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

First inductive step

2	**X**								
3	**X**	**X**							
4	**X**	**X**							
5	**X**	**X**	**X**	**X**					
6	**(2,6) (7,6)**	**X**	**X**	**X**	**X**				
7	**X**	**(4,8) (3,9)**	**X**	**X**	**X**	**X**			
8	**X**	**X**	**(5,11) (6,10)**	**(5,11) (6,10)**	**X**	**X**	**X**		
9	**X**	**X**	**(5,11) (6,10)**	**(5,11) (6,10)**	**X**	**X**	**X**		
10	**(2,10) (7,10)**	**(4,10) (3,10)**	**X**	**X**	**X**		**X**	**X**	**X**
11	**X**	**X**	**X**	**X**		**X**	**X**	**X**	**X**	**X**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

Third inductive step

2	**X**									
3	**X**	**X**								
4	**X**	**X**								
5	**X**	**X**	**X**	**X**						
6	**X**	**X**	**X**	**X**	**X**					
7	**X**	**(4,8)** **(3,9)**	**X**	**X**	**X**	**X**				
8	**X**	**X**	**(5,11)** **(6,10)**	**(5,11)** **(6,10)**	**X**	**X**	**X**			
9	**X**	**X**	**(5,11)** **(6,10)**	**(5,11)** **(6,10)**	**X**	**X**	**X**			
10	**X**	**X**	**X**	**X**	**X**		**X**	**X**	**X**	
11	**X**	**X**	**X**	**X**		**X**	**X**	**X**	**X**	**X**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**

The fourth pass provides no changes, so we converge. The equivalence classes are:
{1}, {2,7}, {3,4,8,9}, {5,11}, (6,10}

Theorem 2 established that NFAs are no more "powerful" than DFAs in terms of the family of languages they can recognize. However, NFAs are more compact and succinct than DFAs because of the potential exponential relationship between the number of states required for a DFA to recognize the same language as an NFA. We will return to this issue later, when we prove that every regular language has a unique minimal state DFA, up to renaming of the states.

Regular Expressions

Definition 20. The set, $\mathbf{E}(\Sigma)$, of *regular expressions over* Σ is a subset of $(\Sigma')^*$ defined inductively below, where $\Sigma' = \Sigma \cup \{\phi, \lambda, (,),+,*,\cdot\}$, called the *meta alphabet* of $\mathbf{E}(\Sigma)$, is assumed to satisfy: $\Sigma \cap \{\phi, \lambda, (,),+,*,\cdot\} = \Phi$.

Basis: $\phi, \lambda \in \mathbf{E}(\Sigma)$ and $\Sigma \subseteq \mathbf{E}(\Sigma)$.
Inductive rule: If $e_1, e_2 \in \mathbf{E}(\Sigma)$, then each of the following strings is in $\mathbf{E}(\Sigma)$:
[1] $(e_1)^*$
[2] $(e_1 \cdot e_2)$
[3] $(e_1 + e_2)$
Completion: Nothing else is in $\mathbf{E}(\Sigma)$ that cannot be obtained by a finite application of the above rules.

Examples. If $\Sigma = \{a,b\}$, then $\Sigma' = \{a,b, \phi, \lambda, (,),+,*,\cdot\}$. The following strings are members of $\mathbf{E}(\Sigma)$. From the basis: ϕ, λ, a, b, are members of $\mathbf{E}(\Sigma)$. From the induction rules: $(\phi)^*$, $(\lambda)^*$, $(a)^*$, $(b)^*$, $(\phi)^*$, $(b+(\phi)^*)$, $((a)^* \cdot b)$, etc., are members of $\mathbf{E}(\Sigma)$.

The following are not members of $\mathbf{E}(\Sigma)$: **(**a**)**, **((***ab**)**

At this point $\mathbf{E}(\Sigma)$ is nothing more than a set of strings, a language over Σ'. In our next definition we assign *meaning* to these strings. That is, for the sake of understanding, think of regular expressions as "programs" written in the language $\mathbf{E}(\Sigma)$. Definition 21 will tell us what we get when these programs are "compiled" and "executed," or "interpreted." The result of executing a regular expression "produces" or "outputs" a *language over* Σ. Definition 21 introduces the "Language of" operator, L[], that maps each $e \in \mathbf{E}(\Sigma)$ to $L[e] \subseteq \Sigma^*$, that is, $L[e]: \mathbf{E}(\Sigma) \rightarrow 2^{\Sigma^*}$, where 2^{Σ^*}, denotes the *power set of* Σ^*, the set of all languages over Σ.

Definition 21. Let Σ be an alphabet and let $\mathbf{E}(\Sigma)$ be the language of regular expressions over Σ. Then each $e \in \mathbf{E}(\Sigma)$ describes(defines or specifies) a language, L[e], over Σ (that is, $L[e] \subseteq \Sigma^*$) defined inductively below.

Basis: $L[\phi] = \Phi$, $L[\lambda] = \{\lambda\}$, and $L[a] = \{a\}$, for each $a \in \Sigma$.
Inductive rules:
[1] if $e = (x)^*$, for some $x \in \mathbf{E}(\Sigma)$, then $L[e] = L[x]^*$
[2] if $e = (x \cdot y)$, for some x and $y \in \mathbf{E}(\Sigma)$, then $L[e] = L[x] \cdot L[y]$, and
[3] if $e = (x+y)$, for some x and $y \in \mathbf{E}(\Sigma)$, then $L[e] = L[x] \cup L[y]$.

Examples.
$L[(\phi)^*] = L[\phi]^* = \Phi^* = \{\lambda\} = L[\lambda]$; by [1] and Basis.

L[((ϕ+((a.b))*) ·ϕ)] = L[(ϕ+((a.b))*)]·L[ϕ] by [2]
= L[(ϕ+((a.b))*)]·Φ = Φ by Basis and Proposition 1(1).

Observe that L[e] = L[f] is possible even though e ≠ f. Also observe that it is important that the inductive rules be applied properly. To this point consider the expression e = (a+(b·a)). If one attempts to apply rule [2] to this expression, then x must equal "a+(b" and y must equal "a)". But since these strings are not valid regular expressions, rule [2] does not apply. The only correct decomposition of e is obtained using rule [3].

Theorem 7. For every e ∈ **E**(Σ), L[e] is a regular language over Σ.

Proof. Define #(e) to be the number of occurrences of the operator symbols {*,+, · } in e. We will prove by induction on #(e) that L[e] is regular.

Basis: #(e) = 0.
Then by Definition 20, e must belong to Σ ∪ {λ, ϕ}. From the Basis of Definition 21 we obtain, L[e] = L[a] = {a}, for a ∈Σ, L[e] = L[ϕ] = Φ, or L[e] = L[λ] = {λ}, respectively. But by Theorem 1, every finite set is Regular. Thus the basis case is true.

Induction Step: *IH*: Assume L[e] is regular if 0 ≤ #(e) ≤ k, for some k. Consider e' where #(e') = k+1. Then e' has at least one occurrence of the operator symbols {*, +, · } and so e' must be of one of the following forms:
(a) (x)*, where #(x) = k
(b) (x·y), where #(x) ≤ k and #(y) ≤ k
(c) (x+y), where #(x) ≤ k and #(y) ≤ k

If (a) holds, then by Definition 21[1], L[e'] = L[x]*. If (b) holds, then by Definition 21[2], L[e'] = L[x] · L[y]. If (c) holds, then by Definition 21[3]], L[e'] = L[x] ∪ L[y]. In each case, by our induction hypothesis, L[x] and L[y] are regular and thus, in each case, by Theorem 4 it follows that L[e'] is regular. This concludes the proof.

Theorem 7 has established that L[e] is regular for every regular expression e. We now state and prove that the converse is also true, demonstrating that regular expressions define another way to characterize or define regular languages over some given alphabet.

Theorem 8a. Let R ⊆ Σ* be a regular language. Then there is a regular expression, e_R ∈ **E**(Σ) for which L[e_R] = R.

Proof (sketch).

Since R is regular, R = L(M) for some DFA, M = (Q, Σ, δ, q_0, A). The method we shall introduce for converting a DFA to an equivalent regular expression is based on the idea that a regular expression is a description of the set of possible "paths" from the initial state of the DFA to some given destination state. Let n = |Q|, the number of states of M. Then for each state k, 1 ≤ k ≤ n, define

$$L_k = \{ x \in \Sigma^* \mid \delta^{\cdot}_M(q_0, x) = k\}$$

L_k is simply the set of input strings that leave M in state k. From this it easy to see that L(M) is just the union of the sets L_k, where $k \in A$, the set of accept states of M. We state this formally

$$L(M) = \bigcup_{k \in A} L_k$$

by

Now, the goal of our approach is to construct a regular expression e_k such that $L[e_k] = L_k$, for each state k of M. To do this we define a system of n equations in n unknowns, where n is the number of states of M and the unknowns are the regular expressions e_k , $1 \le k \le n$. The equation for e_k is generally a recursive equation that describes how to obtain "new" strings in L_k in terms of existing strings in $L_1, L_2, \ldots, L_n$ concatenated with symbols $a \in \Sigma$ that satisfy the relationship: $k = \delta(j, a)$. This is illustrated by the example DFA given below in the form of a state transition table.

Example 26. Transition Table for DFA, M

δ	a	b
→1	1	2
2*	3	4
3	4	1
4*	2	3

Consider L_3 as an example. Clearly $ba \in L_3$ because $\delta^*_M(1,ba) = 3$. Also observe from the TT that $\delta(2,a) = 3$ and $\delta(4,b) = 3$. So, this implies that $L_3 = L_2\{a\} \cup L_4\{b\}$. In other words, any string that will leave M in state 3 must be formed by concatenating "a" on the right end of some string that leaves M in state 2, or by concatenating "b" on the right end of some string that leaves M in state 4. We can define a similar relationship for each of the four states of M. Specifically,

$L_1 = L_1\{a\} \cup L_3\{b\} \cup \{\lambda\}$.
$L_2 = L_1\{b\} \cup L_4\{a\}$.
$L_3 = L_2\{a\} \cup L_4\{b\}$.
$L_4 = L_2\{b\} \cup L_3\{a\}$.

The equation for L_1 has an extra term because it is the start state - state 1 can be reached by the null string because that is the state in which M begins. Observe that $L(M) = L_2 \cup L_4$. This relationship holds because states 2 and 4 are the accept states of M.

This set of equations above can be translated into an Inductive Definition for L(M) as follows.

Basis: $\lambda \in L_1$. $L_2 = L_3 = L_4 = \Phi$.
Inductive Rules:
[1] if x belongs to L_1, then $xa \in L_1$ and $xb \in L_2$;
[2] if x belongs to L_2, then $xa \in L_3$ and $xb \in L_4$;

[3] if x belongs to L_3, then xa ∈ L_4 and xb ∈ L_1 ;
[4] if x belongs to L_4, then xa ∈ L_2 and xb ∈ L_3 ;
Completion Rule: Nothing belongs to L(M) that cannot be added to L_2 or L_4 by finite application of the Basis and Inductive Rules.

Returning to our goal of constructing a regular expression for L(M) we translate the set equations given earlier into equivalent "regular expression" equations (*regular equations* for short) by replacing L_k be e_k, {a} by a, for each a ∈Σ, {λ} by λ, and finally, "∪" by "+". For our example we obtain

$$
\begin{aligned}
& e_1 = e_1a + e_3b + \lambda. \\
[1] \quad & e_2 = e_1b + e_4a. \\
& e_3 = e_2a + e_4b. \\
& e_4 = e_2b + e_3a.
\end{aligned}
$$

Furthermore $e_M = e_2 + e_4$, where L(M) = L[e_M]. Our problem reduces to solving this system of equations for e_1, e_2, e_3, and e_4 purely in terms of the symbols {a, b, λ} and the operators {+,·, ∗}. This can be done with two algebraic tools. First, ordinary algebraic substitution of equals for equals. For example e_2 can be replaced in the right side of equations for e_1,e_3 and e_4 by the expression ($e_1b + e_4a$) to obtain:

$$
\begin{aligned}
& e_1 = e_1a + e_3b + \lambda. \\
[2] \quad & e_3 = (e_1b + e_4a)a + e_4b = e_1ba + e_4aa + e_4b = e_1ba + e_4(aa+b) \\
& e_4 = (e_1b + e_4a)b + e_3a = e_1bb + e_4ab + e_3a
\end{aligned}
$$

After substitution, the expressions can be simplified by applying the law that concatenation (·) distributes over union(+). Factoring out common expressions is also a useful technique at times by applying this distributive law in reverse.

By making the above substitution for e_2, we have reduced the set of equations to just three involving only the unknowns e_1, e_3 and e_4. Note that reducing the number of unknowns is only possible if the expression replacing a given variable does not involve that variable - that is, the expression is not recursive in the variable being replaced. Applying this principle to the remaining three equations we see that only the equation [2] for e_3 is non-recursive. So, by replacing e_3 we can reduce the set to just two equations and two unknowns as shown below.

$$
\begin{aligned}
e_1 &= e_1a + (e_1ba + e_4aa + e_4b)b + \lambda &&= e_1a + e_1bab + e_4aab + e_4bb + \lambda \\
e_4 &= e_1bb + e_4ab + (e_1ba + e_4aa + e_4b)a &&= e_1bb + e_4ab + e_1baa + e_4aaa + e_4ba
\end{aligned}
$$

Collecting terms for the same variables we have:

$$
\begin{aligned}
e_1 &= e_1a + e_1bab + e_4aab + e_4bb + \lambda &&= e_1(a+bab)+e_4(aab+bb)+\lambda \\
e_4 &= e_1bb + e_4ab + e_1baa + e_4aaa + e_4ba &&= e_4(ab+aaa+ba)+e_1(bb+baa)
\end{aligned}
$$

The resulting set of two equations will not simplify by algebraic substitution since both equations are recursive. To break the recursion and be able to solve for the remaining unknowns, we must apply the result of the following lemma.

Lemma 4. Let A,B,X be any subsets of Σ^* satisfying $X = XA \cup B$. Then

(a) if $\lambda \notin A$, then $X = BA^*$
(b) otherwise, BA^* is a subset of any X satisfying this relation.

Proof. See Solutions to Problem Set #1.

In terms of regular expressions, Lemma 4 can be rephrased as Lemma 4'.

Lemma 4'. Let A,B,X be regular expressions over Σ satisfying $X = XA + B$. Then

(a) if $\lambda \notin L[A]$, then $L[X] = L[B]L[A]^* = L[B(A)^*]$
(b) otherwise, $L[B(A)^*]$ is a subset of $L[X]$ for any X satisfying this relation.

Applying Lemma 4' to the equation for e_1, where $X = e_1$, $A = (a+bab)$ and $B = e_4(aab+bb)+\lambda$ we obtain:

[3] $e_1 = (e_4(aab+bb)+\lambda)\ (a+bab)^* = e_4(aab+bb)(a+bab)^* + (a+bab)^*$

Now we can substitute this expression for e_1 into the equation for e_4 to obtain:

$$\begin{aligned} e_4 &= e_4(ab+aaa+ba)+(e_4(aab+bb)(a+bab)^* + (a+bab)^*)\ (bb+baa) \\ &= e_4(ab+aaa+ba)+ e_4(aab+bb)(a+bab)^*(bb+baa) + (a+bab)^*(bb+baa) \\ &= e_4((ab+aaa+ba)+ (aab+bb)(a+bab)^*(bb+baa)) + (a+bab)^*(bb+baa) \end{aligned}$$

Finally, applying Lemma 4' to this equation we obtain:

[4] $e_4 = (a+bab)^*(bb+baa)((ab+aaa+ba)+ (aab+bb)(a+bab)^*(bb+baa))^*$

This is the final expression for e_4. To obtain the final expression for e_1 we must substitute the right side of equation [4] back into the right side of equation [3] . To obtain e_3, we substitute the final expressions for e_1 and e_4 into the right side of equation [2]. Then finally, to obtain the final expression for e_2, we substitute the final expressions for e_1 and e_4 into the right side of equation [1]. The final expression for e_M can be obtained in a similar fashion from the final expressions for e_2 and e_4.

The form of regular expressions given in Definition 20 is somewhat cumbersome because every operator occurrence must be pair with a matched set of parenthesis. By defining the following rules of operator precedence and associativity, and by making the concatenation operator implicit, we can greatly simplify the form of regular expressions.

Operator Precedence and Associativity Laws for Regular Expressions

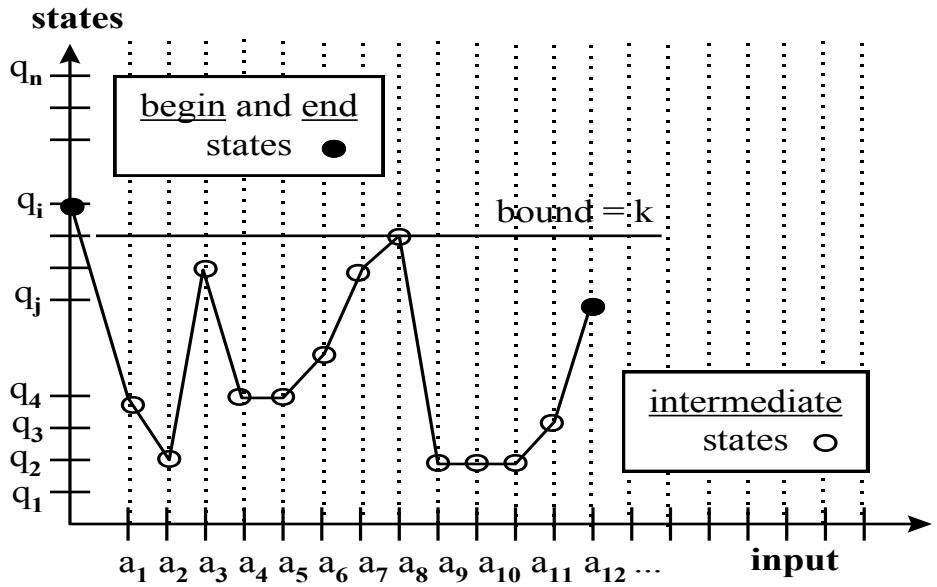
[1] "+" < "·" < "∗" ("+" has the lowest binding strength, "∗" has the highest)
[2] all operators are left associative

Definition 22. A regular expression, r, is said to be reduce if and only if r = ϕ or r has no occurrence of ϕ, and, subject to the above rules of operator precedence and associativity, r has no unnecessary parentheses.

Theorem 8b. (Kleene's Theorem) Let M = (Q, Σ , δ , q_1 , A) be a DFA. Then there is a reduced regular expression e_M such that $L_r[e_M] = L(M)$.

Proof. Without loss of generality we can assume the following about M. Q = {q_1, q_2, …, q_n}, where n ≥ 1, q_1 is the initial state, and for some k', 1≤ k'<n, #A = k'. In other words, A ≠Φ and A ≠ Q. For in the former case, L(M) = Φ, and e_M = ϕ. In the latter case, L(M) = Σ* , for some Σ = {a_1, a_2, …, a_m}, then for m ≥ 2, e_M = ((…((a_1+a_2)+ a_3)… +a_m))*, and for m = 1, e_M = (a_1)*.

For 1 ≤ i, j ≤ n and 0 ≤ k ≤ n define the set $R^k_{i,j}$ = { x ∈Σ* | (q_i, x) $\overset{M}{\Rightarrow}^+$ (q_s, x') $\overset{M}{\Rightarrow}^+$ (q_j,λ) implies s ≤ k.}. In words, $R^k_{i,j}$ = the set of all strings that cause M to transition from state q_i to q_j under the constraint that if M enters an *intermediate state*, q_s, then the index **s** cannot exceed the parameter k. The graph below illustrates a typical sequence of transitions for some x ∈ $R^k_{i,j}$.



We make three claims that will be proved by induction on the parameter k.

(1) $R_{i,j}^k$ is regular

(2) $R_{i,j}^k = R_{i,j}^{k-1} \cup R_{i,k}^{k-1}(R_{k,k}^{k-1})^* R_{k,j}^{k-1}$, for k > 0.

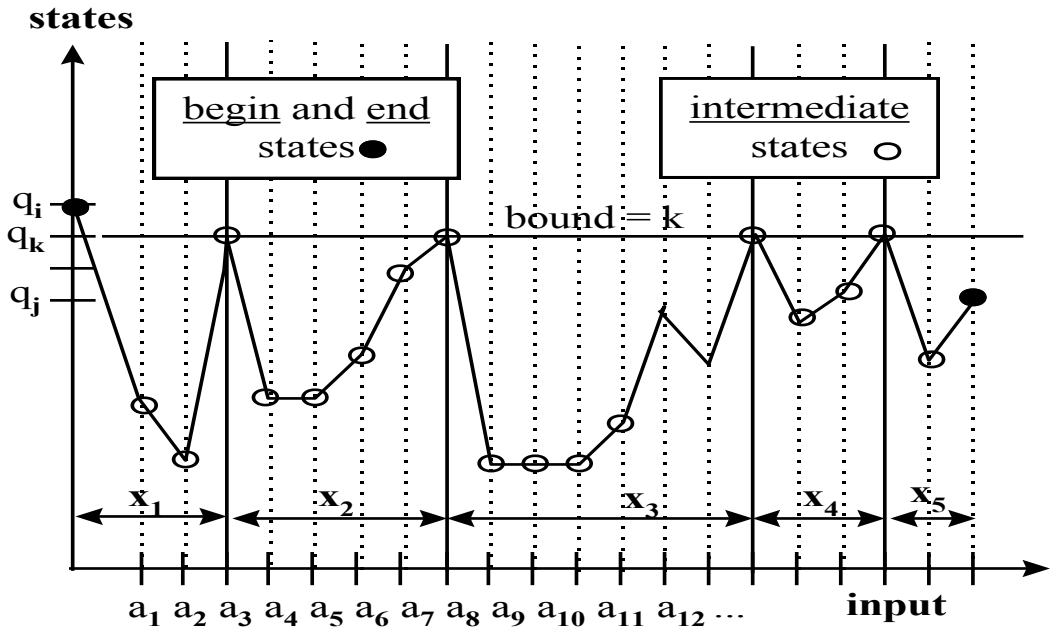
(3) $R_{i,j}^k = L[e_{i,j}^k]$, for some regular expression, $e_{i,j}^k$.

Basis: $R_{i,j}^0 = \{ a \in \Sigma \mid \delta(q_i, a) = q_j \}$, if $i \neq j$, and $R_{i,i}^0 = \{\lambda\} \cup \{ a \in \Sigma \mid \delta(q_i, a) = q_i \}$. Since both these sets are finite, it follows that both are regular. It is straightforward to show that,

$L[e_{i,j}^0] = R_{i,j}^0$, where $e_{i,j}^0 = a'$, or $= (\ldots(a_{1'} + a_{2'}) + \ldots a_{m'})$, where $R_{i,j}^0 = \{a'\}$ or $= \{a_{1'}, a_{2'}, \ldots, a_{m'}\}$ for some $m' \geq 2$.

$L[e_{i,i}^0] = R_{i,i}^0$, where $e_{i,i}^0 = \lambda$, or $= (\ldots(\lambda + a_{1'}) + \ldots a_{m'})$, where $R_{i,i}^0 = \{\lambda\}$ or $= \{a_{1'}, a_{2'}, \ldots, a_{m'}\}$ for some $m' \geq 1$.

Inductive case: To see that (2) holds when k > 0, consider $x \in R_{i,j}^k$. Clearly, $R_{i,j}^{k-1} \subseteq R_{i,j}^k$ follows by definition of $R_{i,j}^k$. So, consider $x \in R_{i,j}^k$ for which M enters state q_k one or more times. This situation is depicted in the graph shown below.

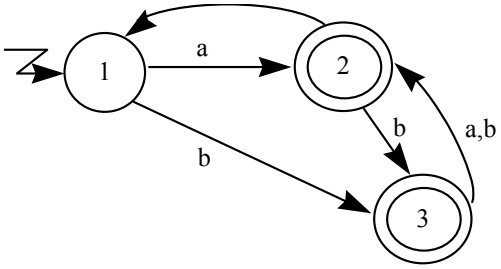


It is easy to see that $x_1 \in R_{i,k}^{k-1}$, x_2, x_3, and $x_4 \in R_{k,k}^{k-1}$, and $x_5 \in R_{k,j}^{k-1}$. Clearly, then, if q_k is entered as an intermediate state only once, then $x \in R_{i,k}^{k-1} R_{k,j}^{k-1}$ and if entered more than once $x \in R_{i,k}^{k-1}(R_{k,k}^{k-1})^+ R_{k,j}^{k-1}$. Thus $x \in R_{i,k}^{k-1}(R_{k,k}^{k-1})^* R_{k,j}^{k-1}$ if q_k is entered one or more times as an intermediate state. If it is not entered at all as an intermediate state, the $x \in R_{i,j}^{k-1}$. Thus claim (2) is established.

By our induction assumption that (1) and (3) hold for k-1, it follows that $R_{i,j}^k$ must be regular by TTheoremT 4 (regular languages are closed under finite concatenation, Kleene-*, and finite union) and can therefore be expressed by the regular expression:
$e_{i,j}^k = (e_{i,j}^{k-1} + ((e_{i,k}^{k-1} \cdot (e_{k,k}^{k-1})^*) \cdot e_{k,j}^{k-1}))$. Thus our claims are established.

To complete the proof of the TtheoremT, we observe that L(M) = $\bigcup_{f \in I_A} R^n_{1,f}$, where I_A is the index set for A. Thus the regular expression, $e_M = (\sum_{f \in I_A} (e^n_{1,f}))$, and the TtheoremT is proved.

Example 27. Consider the DFA shown belowF[1]F.



The table below shows the values of $R^k_{i,j}$ for values of k = 0,1,2 and all values of i and j.

	k = 0	k=1	k=2
$R^k_{1,1}$	λ	λ	(aa)*
$R^k_{1,2}$	a	a	a(aa)*
$R^k_{1,3}$	b	b	a*b
$R^k_{2,1}$	a	a	a(aa)*
$R^k_{2,2}$	λ	λ+aa	(aa)*
$R^k_{2,3}$	b	(λ+a)b	a*b
$R^k_{3,1}$	ϕ	ϕ	(a+b)(aa)*a
$R^k_{3,2}$	a+b	a+b	(a+b)(aa)*
$R^k_{3,3}$	λ	λ	λ+(a+b)a*b

Now L(M) = $R^3_{1,2} \cup R^3_{1,3}$ (M has accepting states 2 and 3), where

$R^3_{1,2} = R^2_{1,2} \cup R^2_{1,3}(R^2_{3,3})^* R^2_{3,2}$ = L_p[(a(aa)* + a*b(λ+(a+b)a*b)*(a+b)(aa)*)]

= L_p[(a + a*b((a+b)a*b)*(a+b))(aa)*]

$R^3_{1,3} = R^2_{1,3} \cup R^2_{1,3}(R^2_{3,3})^* R^2_{3,3} = R^2_{1,3}(\lambda + (R^2_{3,3})^+) = R^2_{1,3}(R^2_{3,3})^*$

= L_p[a*b((a+b)a*b)*]

e_M = ((a + a*b((a+b)a*b)*(a+b))(aa)* + a*b((a+b)a*b)*)

[1] This example was adapted from *Formal Languages and Automata Theory*, by Hopcroft and Ullman.

Important Applications of Regular Expressions

Regular languages and consequently, Regular expressions, have many applications in string processing, decoding or parsing. Perhaps the most obvious and familiar application is in the specification of programming languages. Specifically, the structure or syntax of programming languages is defined at the lexical level (atomic level) in terms of *tokens*. A "token" is an abstraction of a lexical feature, such as, *identifier*, *reserved word*, *real literal*, *operator*, etc. In short, a token may be thought of as the set of all strings, permitted by the programming language to represent a valid lexical element of a given kind. In the design of modern programming languages, tokens are always Regular languages !

Compiler designers have used this fact to advantage in developing tools to facilitate writing and testing compilers. The diagram below illustrates two important steps in the process of compiler development: design and implementation of the lexical analyzer, design and implementation of the syntax analyzer.

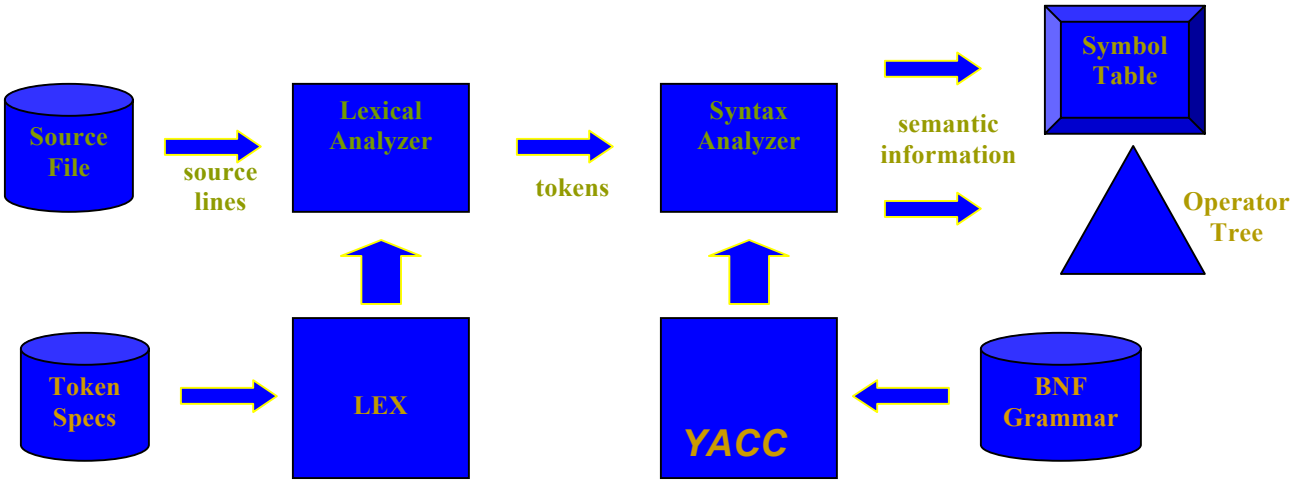


Figure 5. Compiler Development Using Lex and Yacc

The standard UNIX tools, Lex and Yacc, are examples of *program generators*. Program generators are programs that take as input a description of a computational process, and output the actual code of a program that realizes that process. Lex takes as input the specification of all tokens defined by a given programming language, and outputs a C program that will translate source text into a stream of those tokens. For example, if the source text is " X = A + B ; ", then the Lexical Analyzer produced by Lex might translate this string into IDENT(X) OP(=) IDENT(A) OP(+) IDENT(B) DELIMETER(;). Token specifications are input to Lex in the form of Regular Expressions. Similarly, Yacc is a program generator for syntax analyzers or parsers. Yacc takes as input a specification of the programming language in the form of a Type-2 (BNF) Grammar, where the terminal alphabet is the set of tokens identified by a Lex generated program.

The use of such tools greatly reduces the amount of effort necessary to build the front-end components of a compiler (Lexical Analyzer and Parser). The gcc compiler system is designed to exploit this philosophy to the fullest. Gcc provides Lexical Analyzers and Parsers for a variety of languages (C, C++, Ada, …), all of which translate their source language into C. Without tools like Lex and Yacc, rapid development of new languages would be much more difficult.

Pumping Lemma for Regular Languages

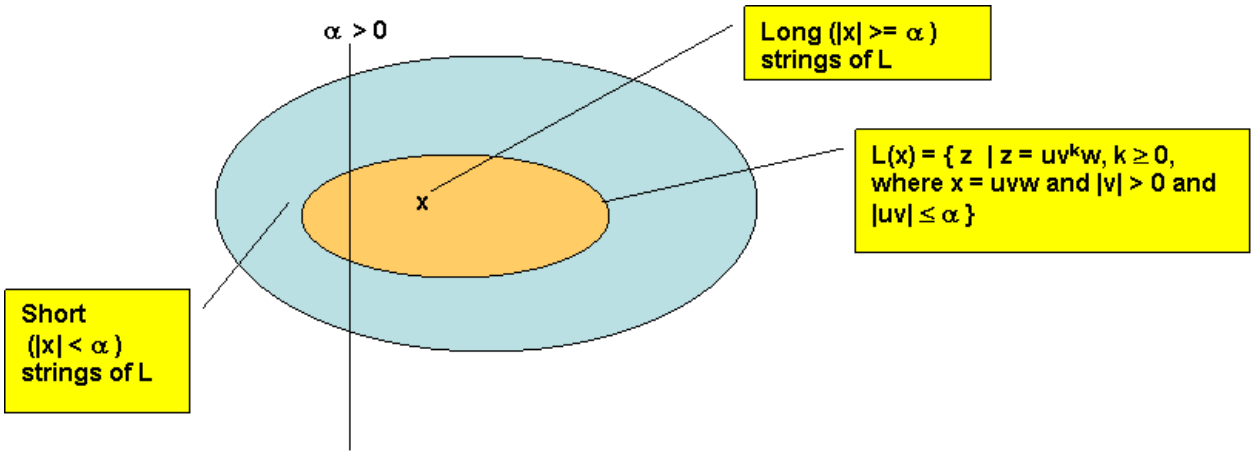
We have completed our study of closure properties of Regular Languages and various formal systems for defining or describing them. The question we wish to consider next is, how can non-Regular languages be identified? The first such tool is the *pumping lemma for Regular languages*. It defines a key property possessed by all Regular languages, specifically all infinite Regular languages (for it is only the infinite languages that can be non-Regular). As a logical assertion, it states that, "L is regular ⇒ L has property P". Negating this implication we have, "L does not have property P ⇒ L is not regular." It is the contra-positive form that is useful in proving languages to be non-Regular. That is, typical proofs using the Pumping Lemma assert that a given language, L, is Regular, and then proceed to obtain a contradiction by showing L does not have property, P. We'll explain what property "P" is after the statement of our next TtheoremT.

T**Theorem**T **9 (Pumping Lemma for Regular Languages(PLR)).** Let L be a Regular language over Σ. There exits a positive integer **α,** depending only on L, such that for every x ∈ L with
$|x| \geq \alpha$, then there exists a decomposition of x in the form **uvw**, where $|uv| \leq \alpha$ and $|v| \geq 1$, such that for every value of $k \geq 0$, $uv^k w \in L$.

The "property P" of Regular languages described in the PLR is: there is a fixed number **α** associated with each regular language, L, such that all strings, x, belonging to L and having length at least **α**, define an infinite subset of L, denoted L_x, defined by

$L_x = \{ uv^k w \mid k \geq 0$, where x = uvw for some strings u,v,w satisfying $|uv| \leq \alpha$ and $|v| \geq 1 \}$

The figure below illustrates this concept.



NOTE: You may be wondering about finite languages. Since the PLR applies to all Regular languages, how does the result hold for finite Regular languages? The answer is that the value of

α for finite languages, L, is always greater than the length of the longest string in L; all members of a finite language are therefore considered to be "short" strings.

Before proving the PLR we apply it to an example. We had stated earlier that certain languages were known to be non-Regular without giving any proof. One of these is $L = \{a^n b^n | \ n \geq 0 \}$. Using the PLR we now supply the proof.

Corollary PLR-1: $L = \{ a^n b^n \mid \ n \geq 0 \}$ is non-Regular.

Proof. The proof will proceed by contradiction. Suppose L is Regular, then the conclusion of the PLR must hold for L. In particular, consider $x = a^\alpha b^\alpha \in L$ (a *long* member of L), where α is the parameter defined for L by the PLR. To obtain a contradiction we must show that <u>for each decomposition</u> of the selected long string x in the form uvw, <u>there exists a k</u> ≥ 0, such that $uv^k w \notin L$.

It is really easier to think of applying the contra-positive of the PLR,
"If ~P(L), then L $\notin$ R(Σ)." The contra-positive Ttheorem T takes the following form; that is, show:
(a) If for all $\alpha > 0$,
(b) there exists $x \in L$ for which $|x| \geq \alpha$, and
(c) for every decomposition of x in the form uvw, where $|uv| \leq \alpha$ and $v \neq \lambda$,
(d) there exists a $k \geq 0$, such that $uv^k w \notin L$,
(e) Then L is not Regular!

Applying this proof template to our example we have:

(a) Let $\alpha > 0$ be given, and consider
(b) $x \in L$ where $x = a^\alpha b^\alpha$ ($|x| = 2\alpha$)
(c) Let x = uvw be any decomposition of x, where $|uv| \leq \alpha$ and $|v| > 0$. Then $v = a^p$ for some p, $1 \leq p \leq \alpha$. Furthermore, $uv^k w = a^{\alpha+(k-1)p} b^\alpha$ for all $k \geq 0$.
(d) For any value of $p > 0$ and any $k \neq 1$, $uv^k w \notin L$. Thus for every decomposition of x = uvw there exists a $k \geq 0$ for which $uv^k w \notin L$ and we conclude,
(e) L is not Regular!

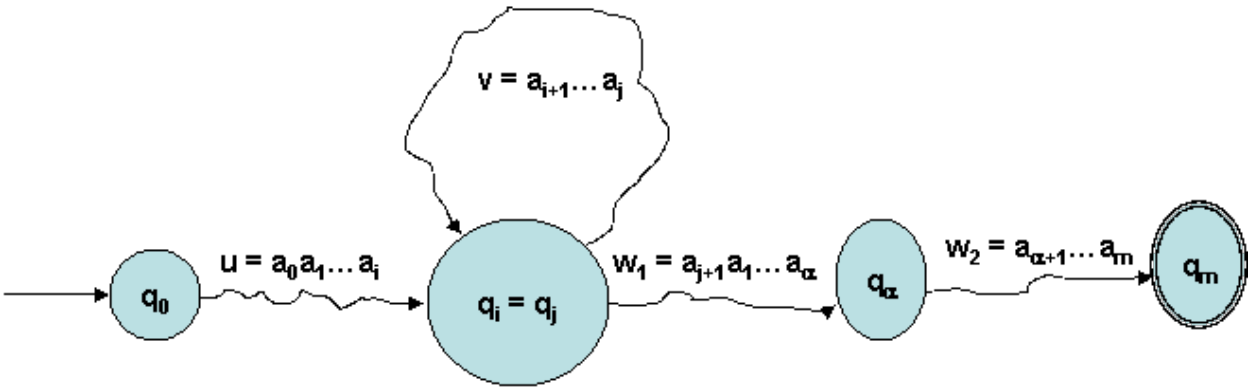
Proof of PLR.

If L is Regular, then L = L(M) for some DFA, M = (Q, Σ, δ, q_0, A). As a candidate for the parameter "α" associated with L we choose the number of states of M, that is, consider α = |Q|. If x ∈L and |x| ≥ α, then let x = $a_1a_2\ldots a_m$, where m ≥ α. Consider the computation of M given by

$$(q_0, a_1a_2\ldots a_m) \Rightarrow_M (q_1, a_2\ldots a_m) \Rightarrow_M \ldots (q_\alpha, a_{\alpha+1}\ldots a_m) (\Rightarrow_M)^* (q_m, \lambda)$$

Because M has only α states, one of the states in the sequence $q_0, q_1, \ldots, q_\alpha$ must be repeated. Let q_i be the first such state and q_j be its second occurrence in this sequence. Then the above computation can be written in the following form (see the figure below), where w = w_1w_2 and $w_2 = a_{\alpha+1}\ldots a_m$ in the following.

$$(q_0, uvw) (\Rightarrow_M)^* (q_i, vw) (\Rightarrow_M)^+ (q_j, w_1w_2) (\Rightarrow_M)^* (q_\alpha, a_{\alpha+1}\ldots a_m) (\Rightarrow_M)^* (q_m, \lambda)$$



Observing that the state subscript corresponds to the number of symbols M must read to reach that state, it follows that |uv| = j ≤ α. Furthermore, because q_m is an accepting state, and by our definition of q_i and q_j, it follows that 1 ≤ |v| ≤ α and each of the following is an accepting computation of M.

$$(q_0, uw) (\Rightarrow_M)^* (q_i, w) = (q_j, w) (\Rightarrow_M)^* (q_\alpha, a_{\alpha+1}\ldots a_m) (\Rightarrow_M)^* (q_m, \lambda)$$

For k > 1,

$$(q_0, uv^kw) (\Rightarrow_M)^* (q_i, v^kw) (\Rightarrow_M)^+ (q_j, v^{k-1}w) = (q_i, v^{k-1}w) (\Rightarrow_M)^* (q_j, w)$$
$$(\Rightarrow_M)^* (q_\alpha, a_{\alpha+1}\ldots a_m) (\Rightarrow_M)^* (q_m, \lambda)$$

Thus for all values of k ≥ 0, it holds that uv^kw ∈L. To complete the proof we note that the above argument is not dependent upon any properties of M other than the number of its states, α. So, we could have chosen the smallest such M. Consequently, the parameter α associated with L is the number of states in the smallest DFA accepting L.

Corollary PLR-2. L = { $a^{f(i)}$ | where $i \geq 0$ and $f(i) = i^2$. } = { λ, a, aaaa, aaaaaaaa, … } is not Regular.

Proof. Once again let us use the template of the contra-positive version of the PLR.
(a) If for all $\alpha > 0$,
(b) there exists $x \in L$ for which $|x| \geq \alpha$, and
(c) for every decomposition of x in the form uvw, where $|uv| \leq \alpha$ and $v \neq \lambda$,
(d) there exists a $k \geq 0$, such that $uv^k w \notin L$,
(e) Then L is not Regular!

(a) Let **$\alpha > 0$ be given, and consider**
(b) $x = a^{f(i)}$ where i is chosen so that $2i + 1 > \alpha$,
(c) Let uvw be any decomposition of x, where $1 \leq |v| \leq \alpha$. Then clearly $v = a^p$ for some p satifying: $1 \leq p \leq \alpha$, and $uv^k w = a^{f(i)+(k-1)p}$.
(d) For k = 2, we have $uv^2w = a^{f(i)+p}$. But $f(i) = i^2 < f(i) + |v| \leq i^2 + \alpha < f(i+1) = (i+1)^2 = i^2 + 2i + 1 = f(i) + 2i + 1$. Thus $uv^2w \notin L$ for any value of p satisfying: $1 \leq p \leq \alpha$.
(e) We may thus conclude that L cannot be Regular.

Exercise 12. Show that each of the following languages is non-Regular using the PLR.

(a) $L = \{ a^n b^m \mid 0 \leq n < m \}$
(b) $L = \{ a^n b^m \mid m \neq n, 0 \leq n, m \}$
(c) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = |x|_a \text{ or } k = |x|_b \} \subseteq \{a,b,c,\#\}^*$
(d) $L = \{ x\#x \mid x \in \{0,1\}^+ \}$
(e) $L = \{ x \mid x \in \{a,b,c\}^* \ |x|_a = |x|_b \}$
(f) $L = \{a,b\}^* \cup \{c\}^+\{a^n b^n \mid n \geq 0 \}$
(g) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = 2|x|_a \} \subseteq \{a,b,c,\#\}^*$
(h) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = 2|x|_a + 5|x|_b \} \subseteq \{a,b,c,\#\}^*$
(i) $L = \{ 0^n 1 0^m 1 0^k \mid n,m,k \geq 0 \text{ and } n+m = k \}$
(j) $L = \{ 0^n 1 0^m 1 0^k \mid n,m,k \geq 0 \text{ and } n+k = m \}$
(k) $L = \{a^n b^n \mid n \text{ is odd} \}$

Proof of (b)
We copy the proof template and fill in the "blanks".

(a) Let $\alpha > 0$ be given,
(b) consider $x = a^\alpha b^{\alpha!+\alpha} \in L$. Clearly, $|x| \geq \alpha$, and
(c) every decomposition of x in the form uvw, where $|uv| \leq \alpha$ and $v \neq \lambda$, implies $v = a^p$, for some p, $1 \leq p \leq \alpha$. Furthermore, for every $k \geq 0$, $uv^k w = a^{\alpha+(k-1)p} b^{\alpha!+\alpha}$
(d) Choose $k = 1 + \frac{\alpha!}{p}$ (k must exist because p is a factor of $\alpha!$). Then $(k-1)p = \alpha!$ and $uv^k w = a^{\alpha+\alpha!} b^{\alpha!+\alpha} \notin L$.
(e) Thus, L is not Regular!

Another tool for identifying non-Regular languages is the use of closure operations on Regular languages. This is stated formally in our next TheoremT.

TheoremT **10.** Let L be a known non-Regular language, let $R_1, R_2, \ldots, R_{n-1}$ be Regular languages for $n \geq 1$, and finally, let L' be some language for which $L = \theta(L', R_1, R_2, \ldots, R_{n-1})$, where
θ: $\mathbf{R}\times\mathbf{R}\times\ldots\times\mathbf{R} \rightarrow \mathbf{R}$ is an n-*ary* closure operation on Regular languages, then L' is not Regular.

Proof. If L' is Regular, then $L = \theta(L', R_1, R_2, \ldots, R_{n-1})$ must be Regular, but this is a contradiction to the assumption that L is known to be non-Regular. Thus L' must be non-Regular.

Example 28.

(a) Let $L = \{ a^n b^n \mid n \geq 0 \}$ and let $R \subseteq L$ be finite. Then by Theorem 10, L - R is non-Regular. This follow trivially from the fact that $L = (L - R) \cup R$. R is Regular (it is finite) and Regular languages are closed under union. Since L is a known non-Regular language by Corollary PLR-1, it follows that L-R must be non-Regular.

(b) Let $L = \{ a^n b^n \mid n \geq 0 \}$ and $L' = \{ x \in \{a,b\}^* \mid |x|_a = |x|_b \}$. Then L' is non-Regular since $L = L' \cap R$, where $R = \{a\}^*\{b\}^*$. It is obvious that R is Regular, and L is known to be non-Regular. Since Regular languages are closed under intersection, it must follow by Theorem 10 that L' is non-Regular.

Exercise 13. Show that each of the following languages is non-Regular using only Theorem 10, Example 28, and Corollary PLR-1.

(a) $L = \{ 0^n b^m \mid 0 \leq n < m \}$
(b) $L = \{ a^n 0^m \mid m \neq n, 0 \leq n, m \}$
(c) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = |x|_a \text{ or } k = |x|_b \} \subseteq \{a,b,c,\#\}^*$
(d) $L = \{ x\#x \mid x \in \{0,1\}^+ \}$
(e) $L = \{ x \mid x \in \{a,b,c\}^* \ |x|_a = |x|_b \}$
(f) $L = \{a,b\}^* \cup \{c\}^+\{a^n b^n \mid n \geq 0 \}$
(g) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = 2|x|_a \} \subseteq \{a,b,c,\#\}^*$
(h) $L = \{ x\#c^k \mid x \in \{a,b\}^* \text{ and } k = 2|x|_a + 5|x|_b \} \subseteq \{a,b,c,\#\}^*$
(i) $L = \{ 0^n 1 0^m 1 0^k \mid n,m,k \geq 0 \text{ and } n+m = k \}$
(j) $L = \{ 0^n 1 0^m 1 0^k \mid n,m,k \geq 0 \text{ and } n+k = m \}$
(k) $L = \{a^n b^n \mid n \text{ is odd} \}$

Proof of (f). L(f) is a language that cannot be proven to be non-Regular using the PLR!

$L_1 = L_f \cap \{c\}^+\{a\}^*\{b\}^* = \{c\}^+\{a^n b^n \mid n \geq 0 \}$
$L_2 = h(L_1)$, where h is the homomorphism given by $h(a) = a$, $h(b) = b$, $h(c) = \lambda$
$L_2 = \{a^n b^n \mid n \geq 0\}$

Myhill-Nerode: An Algebraic Characterization of Regular Languages

We have characterized Regular languages in terms of NFAs (DFAs) and Regular Expressions. In Section A we also stated (presently without proof) that Regular languages are also be defined by Left- and Right-linear grammars. With so many equivalent representations for the family of Regular languages, one is led to wonder if there are any other ways they might be characterized. Also, as we have seen from Exercise 12(f), the Pumping Lemma is not an absolute test for Regular languages, that is, there are some non-Regular languages that cannot be proven so using the PLR.
This raises the question: is there a property P' such that, "R is Regular if and only if P'(R)"?

Myhill and Nerode have given us an affirmative answer to this question! Their celebrated result is stated below as Theorem 10 and is one of the most elegant results of language theory. Theorem 10 is important for three primary reasons: (1) it gives a purely algebraic characterization of Regular Languages distinct from grammars and automata, (2) it gives an absolute test for distinguishing Regular from non-Regular languages, and (3) it provides the basis for constructing the minimal-state DFA recognizing a given Regular language.

To help build your intuition about the concepts central to the proof of Theorem 10, consider the problem of reducing the number of states in a DFA, $M = (Q, \Sigma, \delta, q_0, A)$. We know intuitively that not all equivalent DFAs have the same number of states. This is demonstrated by a trivial example. Consider M and M' illustrated in the figure below. It is clear that $L(M) = L(M') = \Sigma^*$.

For any DFA, M, and any state $q \in Q$, we can define two sets of strings determined by q. The first set we shall denote as $R_q = \{ x \mid \delta^*(q_0,x) = q \}$. This is just the strings that leave M in state q - you were introduced to this set in Theorem 7 when we solved a system of regular equations to obtain the regular expression e_q that satisfies $L[e_q] = R_q$. The second set determined by q is S_q and is given by $S_q = \{x \mid \delta^*(q, x) \in A \}$. These are the strings that would allow M to accept, once M has reached state q. Given these sets, the following is true about L(M).

$$L(M) = \bigcup_{q \in Q} R_q S_q$$

Now suppose that $S_q = S_{q'}$ for two distinct state q and q'. To see that this is possible, refer to our machine M above. Clearly $S_1 = S_2 = \Sigma^*$. However, assuming q and q' are both reachable states of M, then it is never true that $R_q = R_{q'}$, in fact $R_q \cap R_{q'} = \Phi$. For suppose that $x \in R_q \cap R_{q'}$. Because δ_M is a function we have $\delta_M(q_0,x) = q = q'$, contradicting our assumption that q and q' are distinct. The consequence of the fact that $S_q = S_{q'}$ is that the equation for L(M) can now be

$$L(M) = \left(\bigcup_{p \in Q - \{q,q'\}} R_p S_p \right) \cup \left(R_q \cup R_{q'} \right) S_q$$

written as:

This says that, with respect to what M will eventually accepts, there is no difference in the

behavior of M once it reaches q or q'. We say that q and q' are *indistinguishable*. Restating, we can merge q and q' into one state by "rewiring" M in the following way. If $p \neq q'$ is any state in Q and $\delta(p, a) = q'$, then redefine the transition to go to q instead; that is, redefine $\delta(p,a) = q$. Then, remove q' from M and any other states that might become unreachable as a result. If one of q and q' is q_0, then remove the state that is not q_0. After this rewiring, $R_q(new) = R_q(old) \cup R_{q'}(old)$. $S_q(new) = S_q(old)$ and $L(M_{new}) = L(M_{old})$.

This notion of indistinguishability ($S_q = S_{q'}$) is central to Myhill-Nerode and to the algorithm we present later for merging indistinguishable states.

Theorem 11 (Myhill-Nerode). The following statements are equivalent:

(a) L is a regular language over Σ.
(b) L is the union of some of the equivalence classes of some right-invariant equivalence relation of finite index.
(c) The equivalence relation $\mathbf{R_L}$ defined on Σ* by: $x \mathbf{R_L} y$ if and only if $\forall(z \in \Sigma^*)$ $[xz \in L \Leftrightarrow yz \in L]$, has finite index.

The proof will be established by showing that (a) ⇒ (b) ⇒ (c) ⇒ (a). But before doing the proof we review definitions and properties of equivalence relations on sets.

Definition 23. Let S be a set and $E \subseteq S \times S$ (*E* is a relation on S). Then *E* is said to be an equivalence relation if and only if:
(1) *E* is *reflexive*: for all $x \in S$, $(x,x) \in E$;
(2) *E* is *symetric*: $(x,y) \in E \Rightarrow (y,x) \in E$; and
(3) *E* is *transitive*: $(x,y) \in E$ and $(y,z) \in E \Rightarrow (x,z) \in E$.

An equivalence relation *E* is said to be right-invariant with respect to a binary operation $\theta: S \times S \rightarrow S$ iff $(x,y) \in E \Rightarrow \forall z \in S, (x\theta z, y\theta z) \in E$. *Left-invariance* can be defined in a similar fashion. Finally, *E* is said to be a *congruence relation* with respect to θ if it is both left- and right-invariant with respect to θ.

An equivalence relation *E* on S partitions S into *equivalence classes*. That is, S can be expressed as the union of the equivalence classes with respect to *E*. An equivalence class is denoted $[x]_E$ and $= \{ y \mid (x,y) \in E \}$. Since *E* is symetric and transitive, $[x]_E = [y]_E$ where x and y are any two distinct members of the same class. Finally, if $(x,y) \notin E$ then $[x]_E \cap [y]_E = \Phi$.

Example 29. M*odulo n* is a congruence relation on the set of integers. That is $(x,y) \in \equiv_n$ if and only if there is an integer k such that $x = y + k \times n$. The equivalence classes are [0], [1], … [n-1]. Observe that $(0,3) \in \equiv_3$ that is, $0 \equiv_3 3$ because $3 = 0 + 1 \times 3$. The concept of right-invariance with respect to multiplication implies that if $(x,y) \in \equiv_3$ and z is any integer, then $(x \times z, y \times z) \in \equiv_3$. To illustrate let z = 7, then $(0,3) \in \equiv_3$ implies $(0 \times 7, 3 \times 7) \in \equiv_3$ or $(0,21) \in \equiv_3$. Since $21 = 7 \times 3$, it follows that $21 = 0 + 7 \times 3$ and thus $(0,21) \in \equiv_3$.

An equivalence relation is said to have *finite index* if the number of distinct classes is finite, else the relation is of *infinite index*. Mod-n has finite index.

Lemma 5. Let $L \subseteq \Sigma^*$, then the relation R_L is a right invariant equivalence relation on Σ^*, where $(x,y) \in R_L$ if and only if $\forall z \in \Sigma^*[\, xz \in L \Leftrightarrow yz \in L]$.

Proof.
(a) R_L is reflexive: this is a trivial consequence of the definition of R_L - replacing "y" by "x" produces a tautology.

(b) R_L is symetric: that $(x,y) \in R_L$ and $(y,x) \in R_L$ should again be obvious since logical equivalence ($\Leftrightarrow$) is symetric. Specifically, $\forall z \in \Sigma^*[\, xz \in L \Leftrightarrow yz \in L]$ holds if and only if $\forall z \in \Sigma^*[\, yz \in L \Leftrightarrow xz \in L]$. But this implies $(y,x) \in R_L$.

(c) R_L is transitive: if (x,y) and $(y,z) \in R_L$ then for all $w \in \Sigma^*$, xw, yw, and zw are all in L or all are not in L. In particular, it is true that xw and zw are both in L or are both not in L for all w. Thus $(x,z) \in R_L$.

(d) R_L is right invariant with respect to string concatenation: to show right invariance we must show that for any $u \in \Sigma^*$, $(x,y) \in R_L$ implies $(xu, yu) \in R_L$; to this end let $w \in \Sigma^*$ and consider xuw and yuw. Since $(x,y) \in R_L$ if follows that for any $z \in \Sigma^*$, xz and yz are both in L or both not in L. In particular this is true for all z of the form uw, where $w \in \Sigma^*$ is arbitrary and u is fixed. Thus, for all $w \in \Sigma^*$, xuw and yuw are both in L or both not in L and we have established $(xu,yu) \in R_L$.

Proof of Myhill-Nerode.

(a) ⇒(b)
If L is regular, then L = L(M) for some DFA $M = (Q, \Sigma, \delta, q_0, A)$. Define the relation R_M on Σ^* as follows, $(x,y) \in R_M$ if and only if $\delta^*_M(q_0,x) = \delta^*_M(q_0,y)$. That is, x and y are related by R_M if and only if both define computations of M that end in the same state. We now show that R_M is a right invariant equivalence relation with respect to string concatenation.
(1) R_M is reflexive: $\delta^*_M(q0,x) = \delta^*_M(q_0,x)$. Follows from reflexive property of equality (=).
(2) R_M is symetric: $\delta^*_M(q_0,x) = \delta^*_M(q_0,y)$ iff $\delta^*_M(q_0,y) = \delta^*_M(q_0,x)$ by symetry of "=."
(3) R_M is transitive: $\delta^*_M(q_0,x) = \delta^*_M(q_0,y)$ and $\delta^*_M(q_0,y) = \delta^*_M(q_0,z)$ imply $\delta^*_M(q_0,x) = \delta^*_M(q_0,z)$ by the transitivity of "=".
(4) R_M is right invariant: for any $z \in \Sigma^*$, $\delta^*_M(q_0,xz) = \delta^*_M(\delta^*_M(q_0,x),z) = \delta^*_M(\delta^*_M(q_0,y),z)$ $= \delta^*_M(q_0,yz)$, if $\delta^*_M(q_0,x) = \delta^*_M(q_0,y)$. Thus $(x,y) \in R_M$ implies $(xz,yz) \in R_M$, for all $z \in \Sigma^*$.

To complete this part of the proof we note that R_M is of finite index. Specifically, each state of M defines an equivalence class, $[y]_q = \{\, y \mid \Delta_M(q_0,y) = q\}$. So, R_M is of finite index and we finally observe that $L = \bigcup_{q \in A} [y]_q$.

(b)⇒(c) We assume that L is the union of some of the equivalence classes of a right invariant equivalence relation of finite index, say E. From Lemma 5 we have already established that R_L is a right invariant equivalence relation, so what remains is to show that that R_L is of finite index. We do this by showing that for all $x \in \Sigma^*$, $[x]_E \subseteq [x]_{R_L}$. If we can show this, then the index of R_L can be no larger than the index of E. This holds because each equivalence class, $[x]_{R_L}$, of

R_L contains one or more equivalence classes $[x]_E$ of E.

To the contrary suppose there is some $[x]_E$ such that $[x]_E \cap [x]_{R_L} \neq \Phi$ and $[x]_E - [x]_{R_L} \neq \Phi$. Then let $u \in [x]_E - [x]_{R_L}$ and $v \in [x]_E \cap [x]_{R_L}$. Because u and v are not in the same equivalence class under R_L, it follows that there is a $z \in \Sigma^*$ such that $uz \in L$ but $vz \notin L$. Now, by assumption, L is the union of some equivalence classes under E. Thus uz and uv must belong to different equivalence classes under E. But because E was assumed to be right invariant, and because $u,v \in [x]_E$, it must be the case that uz and uv belong to the same equivalence class under E. Thus we have a contradiction and it must follow that for every x, that $[x]_E \cap [x]_{R_L} = \Phi$ or $[x]_E - [x]_{R_L} = \Phi$. This implies that E is a refinement of R_L and must therefore have an index no smaller than R_L. Thus R_L is of finite index.

(c) ⇒ (a) Now we assume that R_L is of finite index. We define a DFA $M_L = (Q, \Sigma, \delta, q_0, A)$ where $Q = \{ [x]_{R_L} \mid x \in \Sigma^* \}$ the set of equivalence classes under R_L is a finite set, $q_0 = [\lambda]_{R_L}$, the class containing the null string, $A = \{ [x]_{R_L} \mid [x]_{R_L} \cap L \neq \Phi \}$, $\delta([x]_{R_L}, a) = [xa]_{R_L}$, for all $[x]_{R_L} \in Q$ and $a \in \Sigma$.

To show this definition is valid, we must show that L is the union of some of the equivalence classes of R_L and that δ is a function and not a multi-valued relation. To show that L is the union of some of the equivalence classes, we observe that if u,v are any two distinct members of $[x]_{R_L}$, then uz and vz are both in L or both not in L for all $z \in \Sigma^*$. In particular then, this holds for $z = \lambda$. Thus either u and v are both in L or both are not in L. Consequently, $[x]_{R_L} \subseteq L$ or $[x]_{R_L} \cap L = \Phi$, for all $[x]_{R_L} \in Q$.

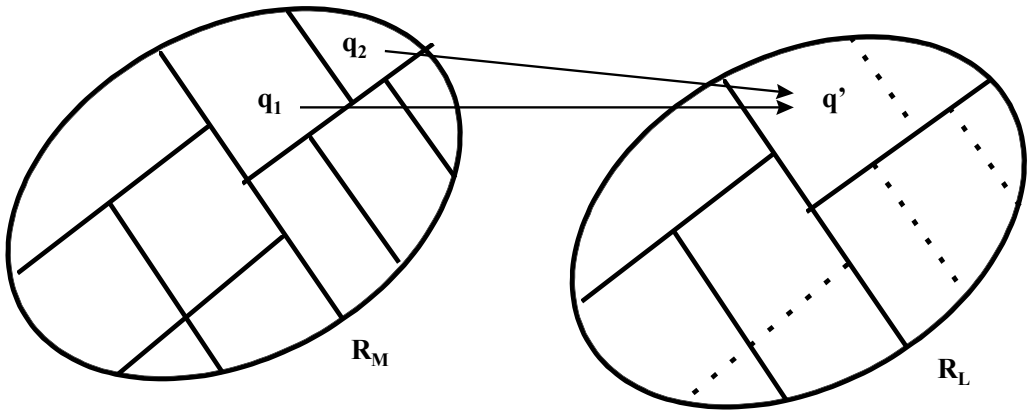
To show δ is a function, let u and v be any two distinct members of $[x]_{R_L}$ and let $a \in \Sigma$. By the right invariance of R_L (Lemma 5) we have $[ua]_{R_L} = [va]_{R_L}$. Thus δ assigns only one state to a given member of $Q \times \Sigma$ and is therefore a function.

To complete the proof we note that $L(M_L) = L$. This follows from the fact that $\delta^*(q_0, x) = \delta^*([\lambda]_{R_L}, x) = [x]_{R_L} \in A$ if and only if $[x]_{R_L} \subseteq L$ if and only if $x \in L$. This fact follows by induction on |x| from the definition of δ.

Finally, we note that from the proof of **(a) ⇒(b)** that every DFA for L has no fewer states than the number of states in M_L. This follows from the fact that the number of states in M_L is exactly the index of R_L together with the fact that the proof of **(a) ⇒(b)** implies that for any DFA M for which L(M) = L, the index of R_M = number of states of M ≥ index of R_L. ***Thus M_L is the smallest DFA that accepts L!!!!!!!!!!!!***

Constructing M_L

To construct M_L we observe that given any DFA, M, accepting L the equivalence relation R_M is a refinement or "approximation" to the equivalence relation R_L. Thus we have to identify equivalence classes under R_M that are subsets of the same equivalence class under R_L. The diagram below illustrates graphically what this means. For example, states q_1 and q_2 in the machine M define equivalence classes of R_M that merged into one equivalence class of R_L represented by the single state q' of the minimal machine M_L.



Given that equivalence classes of these two relations really represent abstract states of their corresponding DFAs, it follows that we want to determine when two (or more) states of M are equivalent to the same state of M_L. This suggests our next definition.

Definition 24. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA, and let $q_1, q_2 \in Q$. Then the relation $\equiv$ is defined on Q as follows: $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(q_1, z) \in A \Leftrightarrow \delta^*_M(q_2, z) \in A]$.

Notice the parallel with the definition of R_L. That is, $(x,y) \in R_L$ if and only if $\forall z \in \Sigma^*$, xz and yz are both in L, or both are not in L. Here $q_1 = \delta^*_M(q_0,x)$ and $q_2 = \delta^*_M(q_0,y)$.

Corollary-24a. $\equiv$ is an equivalence relation on Q. Furthermore, $q_1 \equiv q_2$ if and only if $\forall x \in \Sigma^*$, $\delta^*_M(q_1, x) \equiv \delta^*_M(q_2, x)$.

Proof. It is trivial to show that $\equiv$ is an equivalence relation on Q. This we leave to the reader. By Definition 24, $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$.

This holds if and only if $\forall x,y \in \Sigma^*[\delta^*_M(q_1,xy) \in A \Leftrightarrow \delta^*_M(q_2,xy) \in A]$ if and only if $\forall x \in \Sigma^*$ $[\forall y \in \Sigma^*[\delta^*_M(\delta^*_M(q_1,x),y) \in A \Leftrightarrow \delta^*_M(\delta^*_M(q_2,x),y) \in A]]$ if and only if
$\forall x \in \Sigma^*[\delta^*_M(q_1,x) \equiv \delta^*_M(q_2,x)]$. QED

Corollary-24b. If $q_1 \equiv q_2$ and x and y are any strings in Σ^* for which $\delta^*_M(q_0,x) = q_1$ and $\delta^*_M(q_0,y) = q_2$, then $(x, y) \in R_L$. The converse is also true.

Proof. Again by Definition 24, $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$. By assumption $\delta^*_M(q_0,x) = q_1$ and $\delta^*_M(q_0,y) = q_2$, so we have $q_1 \equiv q_2$ if and only if $\forall z \in \Sigma^* [\delta^*_M(\delta^*_M(q_0,x),z) \in A \Leftrightarrow \delta^*_M(\delta^*_M(q_0,y),z) \in A]$. But by properties of δ^*_M , this holds if and only if $\forall z \in \Sigma^* [\delta^*_M(q_0,xz) \in A \Leftrightarrow \delta^*_M(q_0,yz) \in A]$. And this holds by definition of membership to L = L(M) if and only if $\forall z \in \Sigma^* [xz \in L \Leftrightarrow yz \in L]$. But this last statement is just the definition of $(x, y) \in R_L$. QED

The next Theorem gives us an iterative algorithm for computing the relation $\equiv$ on the state set of a given DFA, M. It also establishes that the minimal state DFA accepting L(M) can be derived from $\equiv$ by treating equivalence classes $[q]_{\equiv}$ as states of this machine.

Theorem 12. Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA and for all $k \geq 0$ define the relation $\equiv_k$ on Q as follows: $q_1 \equiv_k q_2$ if and only if $\forall z \in \Sigma^{*k} [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$. (We say that q_1 and q_2 are *indistinguishable by strings of length k or less*.) Then

(a) $\equiv_0 = \{Q-A, A\}$ and for each $k \geq 0$, $q_1 \equiv_{k+1} q_2$ if and only $q_1 \equiv_k q_2$ and if for every $a \in \Sigma$, $\delta(q_1,a) \equiv_k \delta(q_2,a)$; and

(b) for some $k \leq n-2$, where n is the number of states in M, $\equiv = \equiv_k$; and finally,

(c) $M_L = (Q', \Sigma, \delta', [q_0]_{\equiv} , A')$, where $Q' = \{ [q]_{\equiv} \mid q \in Q\}$, $A' = \{ [q]_{\equiv} \mid q \in A\}$, and for all $a \in \Sigma$, $\delta'([q]_{\equiv}, a) = [\delta(q,a)]_{\equiv}$.

Proof (a). Applying the definition for k = 0, we see that $q_1 \equiv_0 q_2$ if and only if $[q_1 = \delta^*_M(q_1,\lambda) \in A \Leftrightarrow q_2 = \delta^*_M(q_2,\lambda) \in A]$. Thus $\equiv_0 = \{Q-A, A\}$. Now consider $q_1 \equiv_{k+1} q_2$. Clearly if $\forall z \in \Sigma^{*(k+1)} [\delta^*_M(q_1, z) \in A \Leftrightarrow \delta^*_M(q_2, z) \in A]$, then $\forall z \in \Sigma^{*k} [\delta^*_M(q_1,z) \in A \Leftrightarrow \delta^*_M(q_2,z) \in A]$. Thus $q_1 \equiv_k q_2$. But because $\Sigma^{*(k+1)} = \Sigma\Sigma^{*k}$, $q_1 \equiv_{k+1} q_2$ holds if and only if $\forall a \in \Sigma [\forall z \in \Sigma^{*k} [\delta^*_M(q_1, az) \in A \Leftrightarrow \delta^*_M(q_2, az) \in A]$. And this holds if and only if $\forall a \in \Sigma [\forall z \in \Sigma^{*k} [\delta^*_M(\delta(q_1,a),z) \in A \Leftrightarrow \delta^*_M(\delta(q_2,a),z) \in A]$. And finally this holds if and only if $\forall a \in \Sigma [\delta(q_1,a) \equiv_k \delta(q_2,a)]$.

(b) We first show that if $\equiv_k = \equiv_{k+1}$, then $\equiv_k = \equiv$. It should be clear that $q_1 \equiv q_2$ if and only if for all $k \geq 0$, $q_1 \equiv_k q_2$. What we will show is that if $\equiv_k = \equiv_{k+1}$, then $\equiv_k = \equiv_{k+n}$, for all $n \geq 1$. This is easily established by induction on n. The basis (n = 1) is given by assumption. Now suppose $q_1 \equiv_{k+n+1} q_2$. Then by part (a) applied to $\equiv_{k+n+1}$ we have, $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_{k+n} q_2$ and for all $a \in \Sigma$ [$\delta(q_1,a) \equiv_{k+n} \delta(q_2,a)$]. But by our induction hypothesis, $\equiv_k = \equiv_{k+n}$, and we have $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_k q_2$ and for all $a \in \Sigma$ [$\delta(q_1,a) \equiv_k \delta(q_2,a)$]. But this is the same as saying $q_1 \equiv_{k+1} q_2$. And then, since $\equiv_k = \equiv_{k+1}$, it follows that $q_1 \equiv_{k+n+1} q_2$ if and only if $q_1 \equiv_k q_2$. Thus $\equiv_k = \equiv_{k+n+1}$ and the induction is complete.

To finish (b) suppose M has n states and assume both Q-A and A are non-empty. Then by part (a), $\equiv_0 = \{ Q-A, A\}$. Suppose that $\equiv_0 \neq \equiv_1$, then $\equiv_1$ must have at least one more equivalence class than $\equiv_0$ (at least 3). Furthermore, because $q_1 \equiv_{k+1} q_2$ implies $q_1 \equiv_k q_2$, then at least one class of $\equiv_1$ was formed by splitting one of the classes of $\equiv_0$. Thus if we consider the progression of

equivalence relations, $\equiv_0, \equiv_1, \ldots, \equiv_k$, where $|\equiv_0| = 2$ and $|\equiv_{j+1}| \geq |\equiv_j|+1$, then since the $|\equiv_k|$ cannot exceed the number of states of M, it follows that $k \leq n-2$ and $\equiv_k = \equiv_{k+1}$. Thus for any $j > n-2$, it must follow that $\equiv_j = \equiv_k = \equiv$.

STATE MINIMIZATION ALGORITHM

Let $M = (Q, \Sigma, \delta, q_0, A)$ be a DFA. The minimal state DFA equivalent to M can be computed in the following way.

Step 1: Eliminate the unreachable states of M to obtain $M' = (Q', \Sigma, \delta, q_0, A')$. The only difference between M and M' will be that $Q' \subseteq Q$ and $A' \subseteq A$.

Step 2: If $A' = \Phi$, then $L(M) = L(M') = \Phi$, and both are equivalent to the 1-state DFA, $M'' = (\{q_0\}, \Sigma, \delta'', q_0, \Phi)$, where $\delta''(q_0,a) = q_0$, for all $a \in \Sigma$. On the other hand, if $A' = Q'$, then $L(M) = L(M') = \Sigma^*$, and both are equivalent to $M'' = (\{q_0\}, \Sigma, \delta'', q_0, \{q_0\})$, where $\delta''(q_0,a) = q_0$, for all $a \in \Sigma$.

Step 3: If $A' \neq \Phi$ and $Q'-A' \neq \Phi$, then do the following.
For $k = 0, 1, \ldots$, compute $\equiv_k$ until $\equiv_k = \equiv_{k+1}$. The minimal state DFA equivalent to M' is the machine, $M_L = (Q', \Sigma, \delta', [q_0]., A')$, where $Q' = \{ [q]. \mid q \in Q\}$, $A' = \{ [q]. \mid q \in A\}$, and for all $a \in \Sigma$, $\delta'([q]., a) = [\delta(q,a)]..$ Use $\equiv_k$ as $\equiv$.

The above is essentially what we did earlier. We can also approach this from the concept of finding incompatible states, a technique discussed earlier.

Chomsky's Hierarchy

One of the fundamental questions computer scientists (language theorists) attempt to answer is: "What general structural properties can be ascribed to computable languages accepted by programs (algorithms) having given properties?" For example, can we write programs whose input language is English (and whose output language is French?) In other words, are computers capable of understanding and processing natural languages?

This last question, or something similar to it, motivated a natural linguist, Noam Chomsky, to devise a formal mathematical system for describing (defining) English in the mid-50's (in fact, he may still be following this pursuit today.) Chomsky called his formal system a *phrase structure grammar* and is defined formally in Definition 5.

DEFINITION 5. A general *phrase structure grammar* **(psg)** is a 4-tuple, G = (N, Σ, P, S), where
N is an alphabet of *non-terminal symbols* (also called *syntax variables*),
S ∈**N** is a unique non-terminal called the *start symbol*, and
Σ is an alphabet of *terminal symbols* (also called *syntax constants*). It is required that N ∩ Σ = **Φ**.
$V_G = N \cup \Sigma$ is called the *vocabulary of G*, and
P is a finite subset of $(V_G)^*N(V_G)^* \times (V_G)^*$ called the *set of productions* (also called re-writing rules). A member (u,v) of P is denoted by $u \rightarrow v$, where u is called the *leftpart* and v is called the *rightpart* of the rule.

The term "re-writing rule" is intended to connote an operation, denoted $\Rightarrow_G$, defined on strings of symbols over the vocabulary of G, called *sentential forms*, whereby a given sentential form, α, can be rewritten to form another sentential β. This operation or relation is called the *directly produces* (*directly derives*)(*rewrites as*) relation and is defined formally by:

For $\alpha, \beta \in (V_G)^*$: $\alpha \Rightarrow_G \beta$ if and only if $\alpha = \alpha_1 u \alpha_2$ and $\beta = \alpha_1 v \alpha_2$, where $u \rightarrow v \in P$.

We denote the transitive closure of $\Rightarrow_G$ by $(\Rightarrow_G)^+$ and its reflexive-transitive closure by $(\Rightarrow_G)^*$ [$\alpha (\Rightarrow_G)^* \beta$ is read α "derives" ("produces")("generates") β in G**]**. The relation $\alpha (\Rightarrow_G)^+ \beta$ tells us that β can be derived from α by a sequence of one or more rewriting operations of G; $\alpha (\Rightarrow_G)^* \beta$ implies that α = β or $\alpha (\Rightarrow_G)^+ \beta$. Sometimes we write $\overset{r}{\Rightarrow}_G$ to indicate that a particular rule r: $u \rightarrow v \in P$ was used in the re-writing operation. Finally, when $\alpha (\Rightarrow_G)^+ \beta$ and we are interested in a specific sequence of rules (π) used in obtaining β from α, we write $\alpha \overset{\pi}{\Rightarrow}_G \beta$.

DEFINITION 6. Let G = (N, Σ, P, S) be a phrase structure grammar. The *language generated(defined)(produced) by G* is the set

$$L(G) = \{ x \in \Sigma^* \mid S (\Rightarrow_G)^+ x \}$$

Example 2. Consider G = (N, Σ, P, S) where N = {S, X}, Σ = {0, 1}, and

P = { 1: S → 1S,
2: S → λ, "λ" denotes the null string
3: S → 0X,
4: X → 1X,
5: X → 0S }

$S \overset{1}{\Rightarrow}_G 1S \overset{1}{\Rightarrow}_G 11S \overset{3}{\Rightarrow}_G 110X \overset{4}{\Rightarrow}_G 1101X \overset{5}{\Rightarrow}_G 11010S \overset{2}{\Rightarrow}_G 11010 \in L(G)$.

Exercise: Prove that $L(G) = \{ x \in \Sigma^* \mid |x|_0$ is even $\}$.

Example 3a. Consider G = (N, Σ, P, S) where N = {S}, Σ = {0, 1}, and

P = { 1: S → 1S0,
2: S → λ, "λ" denotes the null string
}

$S \overset{1}{\Rightarrow}_G 1S0 \overset{1}{\Rightarrow}_G 11S00 \overset{2}{\Rightarrow}_G 1100 \in L(G)$.

Exercise: Prove that $L(G) = \{ 1^n0^n \mid n \geq 0 \}$.

Example 3b. Consider G = (N, Σ, P, E) where N = {E, T, F, X}, Σ = {n,v, +, -, *, /, (,)} and where

P = { 1: E → E + T,
2: E → E – T,
3: E → T,
4: T → T∗ F,
5: T → T/ F,
6: T → F,
7: F → +X
8: F → –X
9: F → X
10: X → n,
11: X → v,
12: X → (E),
}

G is an example of a Context-free grammar that is not a Linear CFG. It is also an example of grammar for generating arithmetic expressions over binary operators {+, -, , /} and operands {n , v} ("n" denotes a number, "v" denotes a variable.)

$E \overset{1}{\Rightarrow}_G E+T \overset{3}{\Rightarrow}_G T+T \overset{4}{\Rightarrow}_G T+T*F \overset{66}{\Rightarrow}_G F+F*F \overset{(11)(11)(10)}{\Rightarrow}_G v+n*v \in L(G)$.

Exercise: Give a derivation for: v*(n+v/n)/n

Example 4. General psg. $G = (N, \Sigma, P, S')$ where $N = \{S, S', X, Y, Z, Z'\}$, $\Sigma = \{a,b,c\}$, and

P = {
1: S' → abc,
2: S' → SXYZ',
3: S → SXYZ,
4: S → XYZ,
5: YX → XY,
6: ZX → XZ,
7: ZY → YZ,
8: Z' → c,
9: Xa → aa,
10: Xb → ab,
11: Yb → bb,
12: Yc → bc,
13: Zc → cc }

$S' \overset{2}{\Rightarrow}_G SXYZ' \overset{3}{\Rightarrow}_G SXYZ\,XYZ' \overset{4}{\Rightarrow}_G XYZXYZ\,XYZ' \overset{65}{\Rightarrow}_G XXYZYZXYZ' \overset{6565}{\Rightarrow}_G$
$XXXYZYZYZ' \overset{777}{\Rightarrow}_G XXXYYYZZZ' \overset{8(13)(13)}{\Rightarrow}_G XXXYYYccc \overset{(12)(11)(11)}{\Rightarrow}_G$
$XXXbbbccc$
$\overset{(10)99}{\Rightarrow}_G aaabbbccc = a^3b^3c^3 \in L(G).$

Exercise 4. Prove that $L(G) = \{ a^nb^nc^n \mid n \geq 1 \}$

Example 5. Consider $G = (N, \Sigma, P, S)$ where $N = \{S, X, Y, Z\}$, $\Sigma = \{a,b,c\}$, and

P = {
1: S → SXYZ,
2: S → λ, "λ" denotes the null string
3: YX → XY,
4: ZX → XZ,
5: ZY → YZ,
6: X → a,
7: aX → aa,
8: aY → ab,
9: bY → bb,
10: bZ → bc,
11: cZ → cc }

$S \overset{1}{\Rightarrow}_G SXYZ \overset{2}{\Rightarrow}_G XYZ \overset{6}{\Rightarrow}_G aYZ \overset{8}{\Rightarrow}_G abZ \overset{10}{\Rightarrow}_G abc \in L(G).$

Exercise 5. Give a characterization of L(G) in terms ordinary set notation. In other words, describe mathematically the general form of strings in L(G). What form do members of L(G) have? **Hint:** consider derivations (rule sequences) that begin with sequences in the following set $D = \{ 1^k26^k \mid k \geq 2\}$. *What can you say about the set of strings in L(G) produced by derivations of this form?*

In studying the properties of psgs, Chomsky imposed certain restrictions on the form rules could take. Chomsky (and others) were able to show that these restrictions were very important because they altered the *family of languages* that could be described by grammars conforming to a given type of restriction.

> **Terminology**. A *family of languages* is a set or collection of languages all defined by the same formal system (e.g. Phrase Structure Grammar).

The restrictions defined by Chomsky and the family of languages they define are given in the next definition.

DEFINITION 7. Let $G = (N, \Sigma, P, S)$ be a general psg (PSG). Then define,

(a) A *Type-0 grammar* is a PSG with no restrictions;

(b) A *Type-1 grammar* (*context-sensitive grammars*)(CSG) is a PSG for which $|u| \leq |v|$ for all $u \rightarrow v \in P$ (erasing rules are not allowed);

(c) A*Type-2 grammar* (*context-free grammar*)(CFG) is a PSG for which $P \subseteq N \times V_G^*$; that is, $u \rightarrow v \in P$ implies u must be a single non-terminal symbol, but the rightpart v can be any string, including λ.

(d) A *Type-3 grammar* (*right-linear grammar)(RLG*) is a PSG for which $P \subseteq N \times (\Sigma^*)(N \cup \{\lambda\})$; that is, $u \rightarrow v \in P$ implies u must be a single non-terminal symbol, and the rightpart v, if it has a non-terminal, it can only have one, and that non-terminal must be the rightmost symbol of v; otherwise, v can begin with zero or more terminals.

Among the major results we shall establish this term are the following:

1. The family of languages defined by Type-3 grammars (Right-linear; **see Example 2**) (and Left-linear Context-free grammars[2]) is precisely the family of *Regular languages*; and this family is precisely the languages recognized by *non-deterministic (and deterministic) finite automata*. FSA's are algorithms that use a fixed and bounded amount of memory (the set of states) independent of the length of the input. Because all computers are ultimately limited in the amount of accessible memory space, one could argue that the input language of all existing programs is, in a practical sense, a regular language!

2. The family of languages defined by Type-2 grammars(**see Example 3**), or the *Context-free languages*, properly includes the Regular languages and can be recognized by non-deterministic push-down automata (finite automata with a pushdown store or stack). The Context-free family includes a subfamily called the *Deterministic Context-free languages*. This family is particularly important because it is the family from which all programming languages are defined. DCFLs can be accepted by deterministic pushdown automata and defined by LR(k) grammars.

3. The family of languages defined by Type-1 grammars(**see Example 4**), or the *Context-sensitive languages*, properly includes the Context-free languages and can be recognized by *Linear-bounded automata* (Turing machines with memory space bounded by the length of the input).

4. The family of languages defined by Type-0 grammars(**see Example 5**), or the *Recursively Enumerable* languages, properly includes the Recursive languages, which properly includes the Context-sensitive languages, and can be recognized by non-deterministic (and deterministic) *Turing machines*.

Figure A-2 illustrates graphically the inclusion relationships among these language families. In the remainder of this course, we will study the family of Regular languages in depth and introduce the fundamental properties of the Context-free family. The sequel to this course, COT 5310, is a study of the remaining families of the Chomsky Hierarchy.

[2] A left-linear grammar(LLG) is a PSG for which $P \subseteq N \times (N \cup \{\lambda\})(\mathbf{\Sigma}^*)$. It will be shown later that the LLGs and the RLGs define exactly the family of Regular Languages. For this reason, the LLGs and RLGs are commonly (and collectively) called *Regular grammars*. LLGs and RLGs are properly included in a larger subfamily of Type-2 grammars called the Linear Context-free grammars (LCFG). A linear CFG (**see Example 3**) is one for which $P \subseteq \mathbf{N \times (\Sigma^*)}\ (N \cup \{\lambda\})(\mathbf{\Sigma}^*)$; that is $u \rightarrow v \in P$ implies u is a single non-terminal and v can have at most one non-terminal, but it can occur anywhere relative to the other terminals (if any).

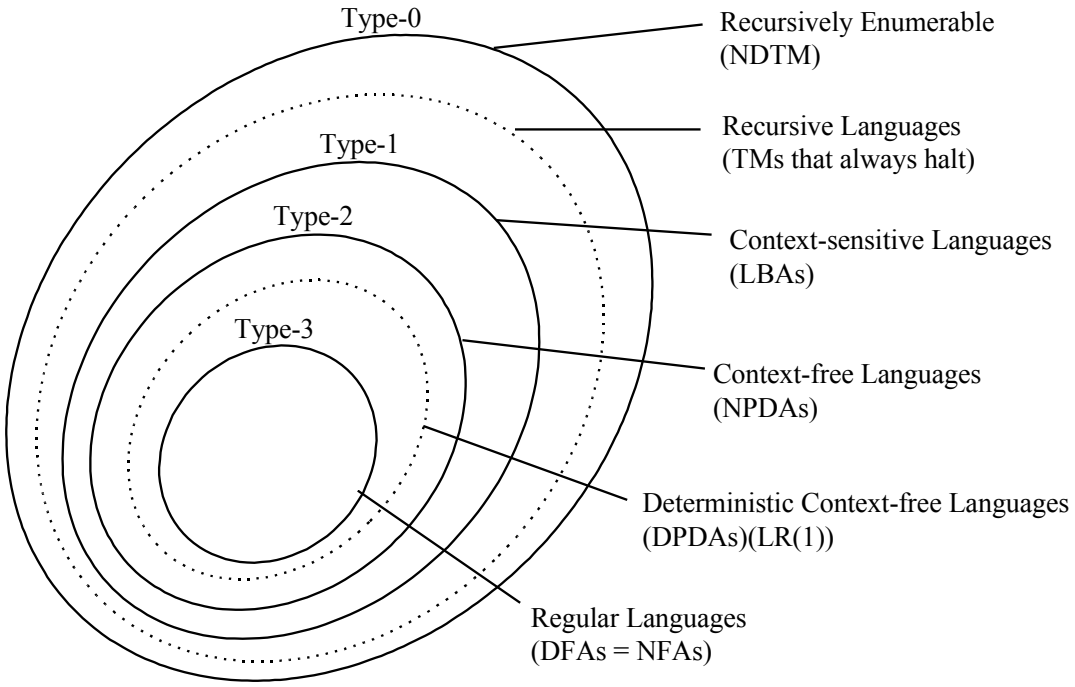


Figure A-2. Chomsky Hierarchy

NOTE: Technically, Context-sensitive languages cannot contain the null string. So, it is somewhat incorrect to depict the Context-free languages as being a proper subfamily of the Context-sensitive family (Context-free languages may include λ). However, this relationship holds for all Type-1 and Type-2 languages NOT containing λ, a "small" error, and so we illustrate their relationship as shown in Figure A-2.

Equivalence of NFAs to Right-linear Grammars

To establish the first equivalence suggested by the Chomsky Hierarchy, we now show that the Regular languages (defined by DFAs and NFAs) is exactly the same family of languages that can be defined by Type-3 or Right-linear grammars. This is the subject of our next theorem.

Theorem 3. Let $M = (Q, \Sigma, \delta, Q_0, A)$ be an arbitrary NFA, then there is a Right-linear grammar $G = (N, \Sigma, P, S)$, such that $L(G) = L(M)$. Conversely, if G is an arbitrary RLG, then one can construct an NFA, M, for which $L(M) = L(G)$.

Proof. Let M be given. We construct G from M essentially by (a) introducing a non-terminal for each state of M, and (b) introducing one production for each transition of M. Formally, we define $N = Q \cup \{S\}$, where S denotes the start symbol of G and is distinct from all symbols denoting states of M. P is then defined to be the union of three sets of rules denoted P[1] , P[2], and P[3].

$P[1] = \{\ S \rightarrow q \quad | \ q \in Q_0\ \},$
$P[2] = \{\ q \rightarrow \sigma q' \ | \ q' \in \delta_M(q, \sigma)$ for some $\sigma \in \Sigma \cup \{\Lambda\}\ \}$, and
$P[3] = \{\ q \rightarrow \lambda \ | \ q \in A\ \}$

To show that $L(G) = L(M)$, we have to show that $x \in L(G)$ if and only if $x \in L(M)$. Suppose that $x \in L(M)$. Then there is some accepting computation of x, that is there is some sequence of configurations: $(q_0, y_0) \underset{M}{\Rightarrow} (q_1, y_1) \underset{M}{\Rightarrow} \ldots \underset{M}{\Rightarrow} (q_m, y_m) = (q_m, \lambda)$, where $q_0 \in Q_0$, $y_0 = x$, and $q_m \in A$. Then in G we have the following derivation: $S \underset{G}{\Rightarrow} q_0 \underset{G}{\Rightarrow}^{*} y_0 q_m \underset{G}{\Rightarrow} y_0 = x$. The first step of the derivation applies a rule in P[1] to rewrite S as the particular initial state (q_0) of M that determines an accepting computation. Each rule of π corresponds to a move of M in the accepting computation – whatever input, σ, is consumed by M on any given move, σ will be produced as output by the corresponding rule in P[2]; if $\sigma = \Lambda$, then no input is consumed by M and nothing is written to the sentential form by G. Thus, if M consumes y_0, G will generate y_0 by mimicking the same transitions, but opposite in the IO sense. Finally, the last rule of the derivation is a rule in P[3] – these rules permit the derivation in G to terminate if and only if M is in an accept state.

Using a similar argument it is easily shown that any string x generated by G can also be accepted by M. Thus M and G are equivalent specifications for the same language.

The converse result is established in a similar fashion, but there are some details that are different. So, let us be given an arbitrary RLG, $G = (N, \Sigma, P, S)$. To construct M we will introduce a state for each non-terminal, analogous to our previous construction of G from M. Consistent with the previous construction one might infer that we will also define a transition of M for each rule of G. The problem with this idea is that a rule of G has one of the general forms: $X \rightarrow wY$, and $X \rightarrow w$, where $w \in \Sigma^*$ and X,Y denote non-terminals. If $|w| > 1$, then M would have to consume 2 or more symbols on a given transition, say from state X to state Y – this violates what an NFA can do. To solve this problem, it is necessary to introduce some additional intermediate states between X and Y, in fact we must introduce $|w| - 1$ unique intermediate states (states other than X or Y) when $|w| > 1$. If $|w| \leq 1$, then a transition in M can be defined that

exactly mimics the given rule, $X \rightarrow wY$. Furthermore, for terminating rules of G of the form, $X \rightarrow w$, what is the "next state"? To deal with these rules we introduce a unique accept state Ω. Using this idea, we can think of terminating rules as being of the form, $X \rightarrow w\Omega$, and then treating them exactly like rules of the form $X \rightarrow wY$. Putting this all together formally, we have:

$Q = N \cup N' \cup \{\Omega\}$, where N' is the set of intermediate states needed to support all the *long* rules (rules with $|w| > 1$);

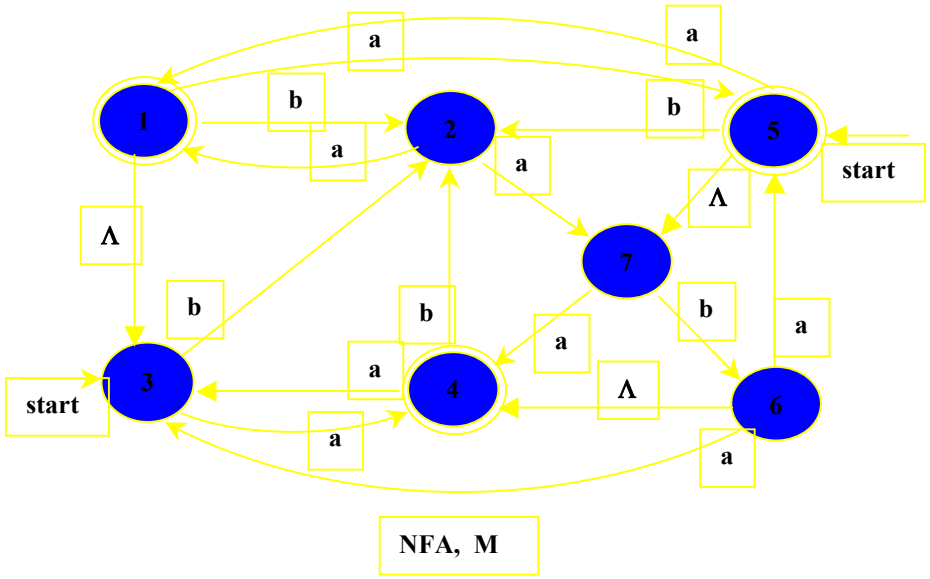
$Q_0 = \{S\}$, the set consisting of the start symbol of G;
$A = \{\Omega\}$, and δ, the transition relation of M is defined by,
[1] if $X \rightarrow wY \in P$ then
case ($w = \lambda$) : define $Y \in \delta(X, \Lambda)$,
case ($w \in \Sigma$) : define $Y \in \delta(X, w)$,
case($w = a_1a_2\ldots a_n \in \Sigma^n$, for some $n > 1$): let $Z_1, Z_2, \ldots Z_{n-1}$ be unique new states in N'
and define $Z_1 \in \delta(X, a_1)$, $Y \in \delta(Z_{n-1}, a_n)$, and for all $1 \leq k < n$ define
$Z_{k+1} \in \delta(Z_k, a_{k+1})$.
[2] if $X \rightarrow w \in P$ then apply [1] to $X \rightarrow w\Omega$.

Note that there are no transitions defined for state Ω. The NFA M will halt precisely when the derivation in G terminates with a rule of the form [2].

Like before, we must prove that L(M) is exactly the same language as L(G). However, since the argument follows much the same line of thought as that given for the converse result, we will not present it here. Nevertheless, there is one detail worth noting: M must complete exactly the sequence of transitions defined through intermediate states (N') to simulate the effect of a given long rule of G. To guarantee this will happen, it is essential in [1] that the set of intermediate states defined for long rules is unique to that rule – these states cannot be "reused" by different long rules, for otherwise M might be able to make unintended transitions leading to accepting strings that could not be generated by G.

We apply the construction techniques introduced by Theorem 3 in Examples 14 and 15 below.

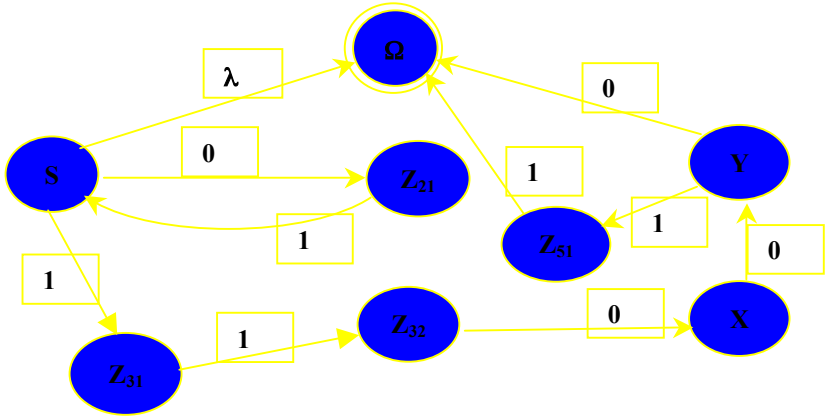
Example 14. Converting an NFA to a Right-linear Grammar.



From the NFA, M, shown in the diagram above we note that there are two initial states, {3,5}and three accept states {1, 4, 5}. Applying the construction of TTheoremT 3 we obtain the following RLG, G = (N, Σ, P, S), where N = { S, A(=1), B(=2), C(=3), D(=4), X(=5), Y(=6), Z(=7) }, Σ = {a, b}, and P is given by:

P[1] = { 1: S → C, 2: S → X }
P[2] = { 3: C → bB, 4: C → aD, 5: A → bB, 6: A → C, 7: A → aX, 8: B → aA, 9: B → aZ,
10: D → aC, 11: D → bB, 12: X → aA, 13: X → bB, 14: X → Z, 15: Y → aX,
16: Y → aC, 17: Y → D, 18: Z → aD, 19: Z → bY }
P[3] = { 20: A → λ, 21: D → λ, 22: X → λ }

Example 15. Converting a Right-linear grammar to an NFA. Let G be given by:
N = {S, X, Y}, Σ = {0, 1}, and P = {1: S –> λ, 2: S –>01S, 3: S –> 110X, 4: X –>0Y,
5: Y –>11, 6: Y –> 0 }. The corresponding NFA is illustrated by the transition diagram below. Pay particular attention to the new intermediate states, they are labeled Zij, where i denotes the rule of G and j is a unique index.



Closure Properties of Regular Languages

Definition 17. Let A, B be arbitrary languages over Σ, and let Δ be an arbitrary alphabet, then we define:

(a) A - B (the *Difference* of A and B) = { x | x ∈ A and x ∉ B }
(b) ~A = Σ* - A (the *Complement of A with respect to Σ*)
(c) A^{rev} (*Reverse of* A) = { rev(x) | x ∈A }, where
rev(λ) = λ, and rev(ax) = rev(x)a, for x ∈Σ* and a ∈Σ.
(d) A *homomorphism* is a mapping, h :Σ* → Δ*, defined as follows:
h(λ) = λ,
h(a) ∈ Δ*, for all a ∈Σ, and
for x = $a_1a_2 \ldots a_n \in \Sigma^+$, n > 1, h(x) = h($a_1$)h($a_2$) … h($a_n$).
Finally, for A ⊆ Σ*, we define h(A) = { h(x) | x ∈ A }
(e) A *substitution from Σ* to Δ** is a mapping, σ: Σ* → $2^{(\Delta^*)}$, where
σ(λ) = {λ}
For each a ∈Σ, σ(a) = $R_a \subseteq \Delta^*$ is a language over Δ, and
for x ∈Σ^*, and a ∈Σ, σ(ax) = σ(a)σ(x) and σ(xa) = σ(x)σ(a).
Finally, for A ⊆ Σ*, we define σ(A) = $\bigcup_{x \in A} \sigma(x)$.
A substitution map is said to be a <u>*regular substitution*</u> from Σ* to Δ* if for every a ∈Σ, σ(a) = R_a is a regular language over Δ. Finally, if Δ = Σ, we say *σ is a substitution <u>on Σ*</u>.*

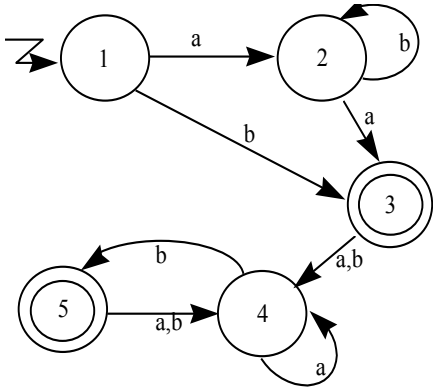
Theorem 4. $\mathbf{R}_\Sigma$ is closed under finite application of the operations listed below. That is, if L ∈ $\mathbf{R}_\Sigma$ and L'∈ $\mathbf{R}_\Sigma$, h is a homomorphism on Σ', and σ is a regular substitution on Σ', then each of the following languages belongs to $\mathbf{R}_\Sigma$.

(a) ~L
(b) L^{rev}
(c) h(L)
(d) σ(L)
(e) L – L'
(f) L ∩ L'
(g) L ∪ L'
(h) L* (Kleene-*)
(h) LL' (concatenation of L with L')

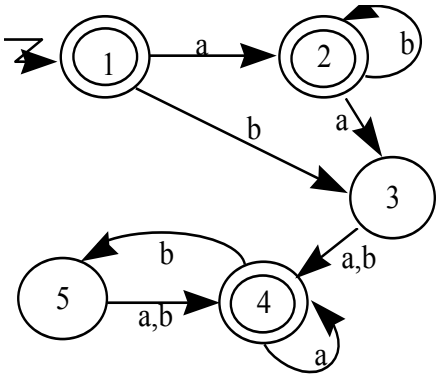
Proof (sketch). For all parts we assume L = L(M_1) and L' = L(M_2), where M_1 = (Q_1, Σ_1, δ_1,α_1, A_1) and M_2 = (Q_2, Σ_2, δ_2, α_2, A_2) are DFAs. We further assume that their state sets are disjoint. The objective in establishing each part is to define an NFA, M' = (Q', Σ', δ', Q_0', A'), such that L(M') is the desired target language and M' is constructed from M_1 and M_2 in some fashion. Alternatively, in light of Theorem 3, we can also assume L and L' are defined by appropriate RLGs, G_1 and G_2, respectively, and then attempt to construct an RLG, G', such that L(G') is the desired language. Finally, each of the construction techniques will be illustrated by example.

(a) L(M') = ~L(M_1): Given $M_1 = (Q_1, \Sigma_1, \delta_1, \alpha_1, A_1)$, define $M' = (Q', \Sigma', \delta', Q_0', A')$, where $Q' = Q_1$, $\Sigma' = \Sigma_1$, $\delta' = \delta_1$, $Q_0' = \{ \alpha_1 \}$ and $A' = Q_1 - A_1$. M' is essentially M_1 with a different set of accepting states: an accepting state of M_1 is non-accepting in M' and vice versa. M' is a DFA and it is essential to this construction that L be defined by a DFA, and not an NFA. The reason is that if the accepting states of an NFA are complemented, the resulting NFA may still be able to accept strings it did before. The first example illustrates the correct construction. The second example illustrates an incorrect construction using an NFA.

Example 16. (Complementing a DFA)

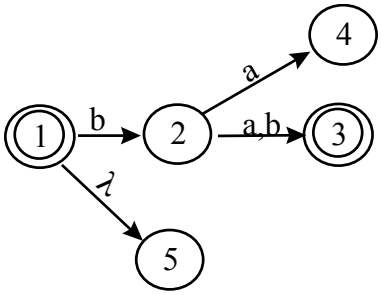


M_1

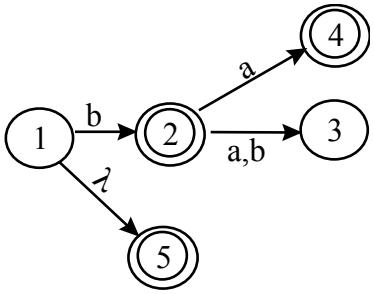


M'

Example 17. (Complementing an NFA – BAD!!!!)



M_1



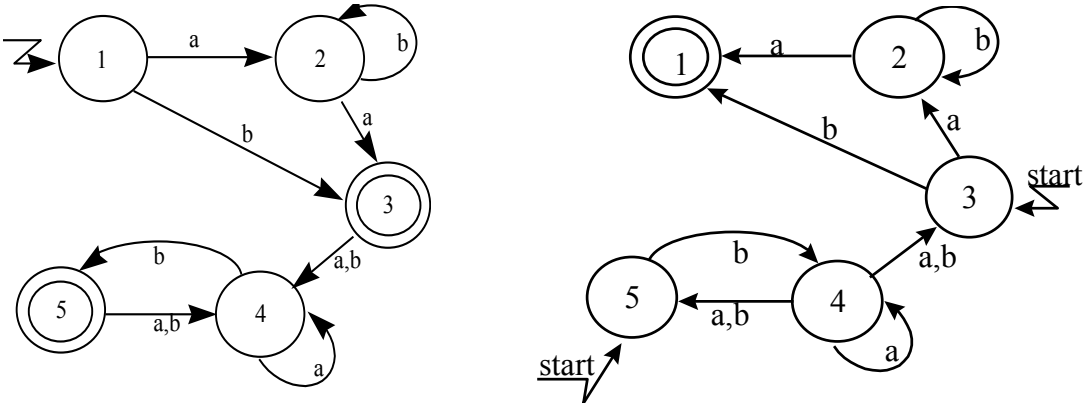
M'

Observe that NFA M_1 accepts λ and ba, as does $M' = \sim M_1$.

(b) $L(M') = L^{rev} = L(M_1)^{rev}$.

To obtain M' from M_1 we simply reverse the direction of transitions and interchange start states with accepting states; that is, the start state of M_1 becomes the only accept state of M' and each accept state of M_1 becomes a start state for M' Formally we have, $M_1 = (Q_1, \Sigma_1, \delta_1, \alpha_1, A_1)$, $M' = (Q', \Sigma', \delta', Q_0', A')$, where $Q' = Q_1$, $\Sigma' = \Sigma_1$, $Q_0' = A_1$, $A' = \{\alpha_1\}$, and for each $q \in Q'$ and $a \in \Sigma'$, $\delta'(q, a) = \{q' \mid \delta_1(q', a) = q\}$.

Example 18. M' the reverse of M_1.



M_1 **M'**

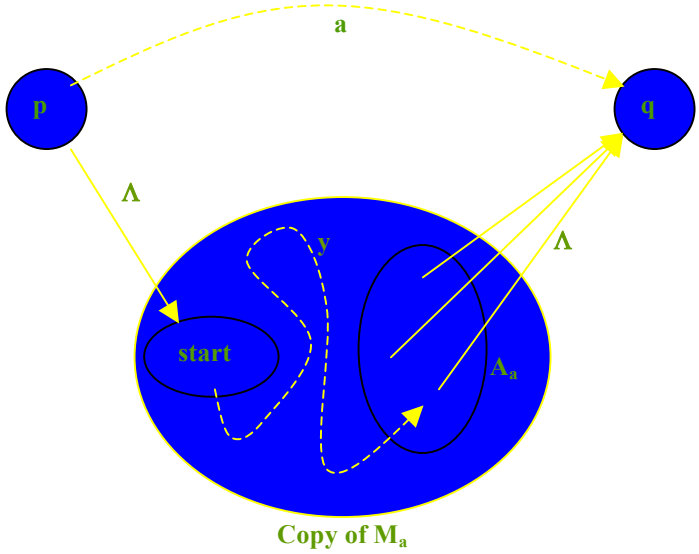
To prove that $L(M') = L(M_1)$, we first observe that M' has no spontaneous transitions. Consequently, every move of M' must consume one input symbol. Next we observe that if $x \in L(M_1)$, then there is some computation π, $(\alpha_1, x) \underset{M1}{\Rightarrow}^{*} (q, \lambda)$, where $q \in A_1$. Because $\delta' = \delta_1^{-1}$, the computation π is reversible in M'. Specifically, $(q, rev(x)) \underset{M'}{\Rightarrow}^{rev(\pi)} (\alpha_1, \lambda)$. This can be formally established by induction on $|x| = |\pi|$. It follows, then, that $rev(x) \in L(M')$ and that $L^{rev} \subseteq L(M')$. The reverse inclusion can be established by reversing this argument.

(c) $L(M') = h(L) = h(L(M_1))$.
(d) $L(M') = \sigma(L) = \sigma(L(M_1))$.

First we observe that every homomorphism on Σ^* is a special type of regular substitution on Σ^*. Specifically, $h(L) = \sigma_h(L)$, where $\sigma_h(a) = \{h(a)\}$, for each $a \in \Sigma$. Since all finite languages over Σ are Regular, σ_h is a regular substitution. Consequently, if we can demonstrate that $\mathbf{R}_\Sigma$ is closed under regular substitution, then it will follow as a corollary that $\mathbf{R}_\Sigma$ is closed under homomorphism.

To this end, for each $a \in \Sigma$, let $\sigma(a) = R_a \subseteq \Sigma^*$ be regular. Let $M_a = (Q_a, \Sigma, \delta_a, \alpha_a, A_a)$ be a DFA that accepts R_a. We will assume that the state sets $\{Q_a \mid a \in \Sigma\}$ are pair-wise disjoint. To illustrate our method for constructing the NFA, M', consider our objective: $x = y_1 y_2 \ldots y_n$ is to be accepted by M', precisely when $a_1 a_2 \ldots a_n$ is accepted by M_1, where $y_k \in \sigma(a_k) = Ra_k$. We build M' by rewiring the transitions of M_1 in the following way. Let $q = \delta_1(p, a)$. In M', we will introduce a Λ-transition from p to the start state of a copy of M_a, and then from each accept state of M_a, we wire a Λ-transition back to q. The diagram below depicts this construction. M', then,

will only be able to reach state q from p by reading some string y $\in\Sigma^*$ that would be accepted by the copy of M_a. To implement this construction, we will need m copies of M_a, for each a $\in\Sigma$, where m is the number of states in M_1. The size of Q' will be $m + m\sum_{a\in\Sigma} |Q_a|$.



Formally then, M' = (Q', Σ, δ', α', A') where $Q' = Q_1 \cup \bigcup_{i=1}^{m} \bigcup_{a\in\Sigma} Q_a^i$, with m = $|Q_1|$ and the superscript **i** designates the **ith** copy of Q_a; α' = α_1 (start state of M' is the start state of M_1); A' = A_1 (the accept states of M' are the accept states of M_1). To define δ', let $Q_1 = \{p_1, p_2, \ldots, p_m\}$. Then for each transition, $p_j = \delta_1(p_k, a)$ define δ' by

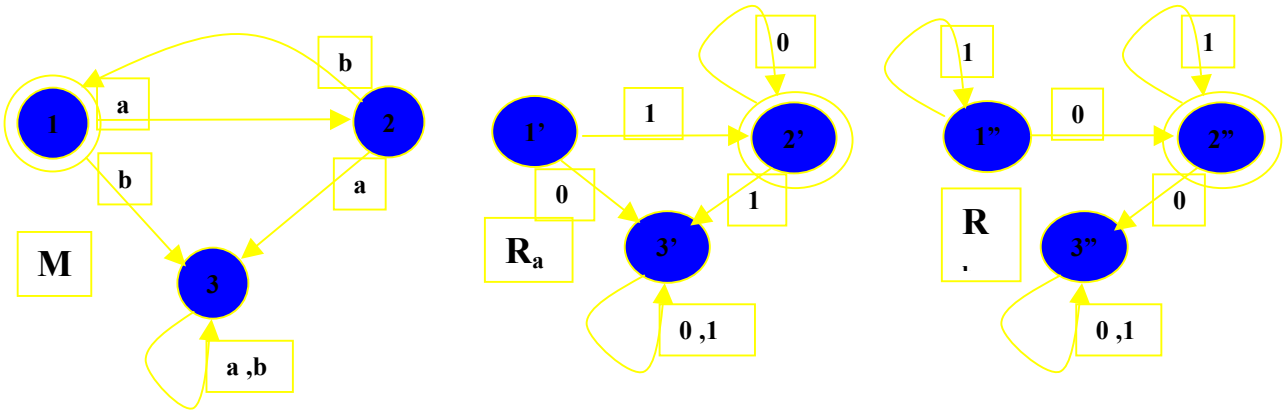
$$\delta'(p_k, \Lambda) = \alpha_a^k;$$

$$\delta'(q, b) = \delta_a^k(q, b) \text{ for all } q \in Q_a^k \text{ and } b \in \Sigma;$$

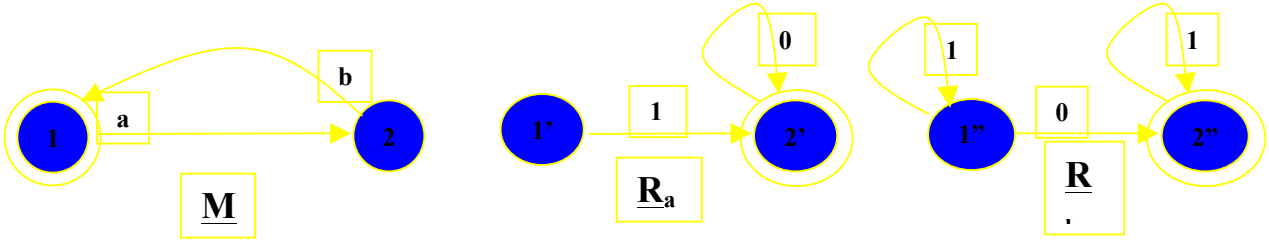
$$\delta'(q', \Lambda) = p_j, \text{ for all } q' \in A_a^k.$$

It may now be argued by induction on |x|, that if $x = a_1a_2\ldots a_n$, n ≥1, is accepted by M_1, then y is accepted by M', where $y = y_1y_2\ldots y_n \in \sigma(x)$ and each $y_k \in \sigma(a_k)$. The converse may be established by decomposing an accepting computation of M' for y into n segments beginning and ending with states of Q_1 (each such intermediate configuration must be reachable only by Λ-transitions from accept states of some copy of M_a, for some a∈Σ.) An induction on the number of such segments can then be used to establish there is a string $x = a_1a_2\ldots a_n$, accepted by M_1, where $y = y_1y_2\ldots y_n \in \sigma(x)$ and each $y_k \in \sigma(a_k)$.

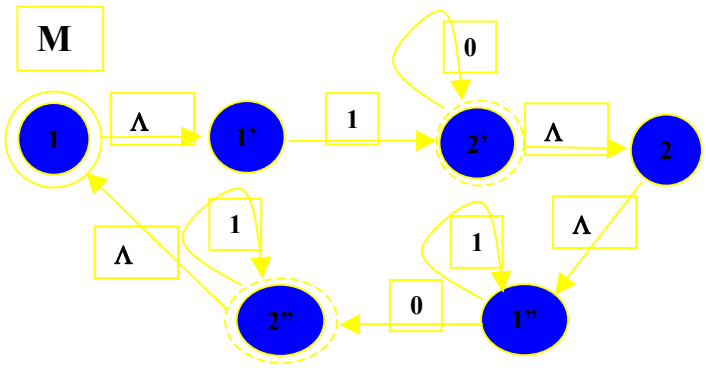
Example 19. Building a substitution NFA from constituent DFAs. Shown below are DFAs M, R_a, and R_b. L(M) = {ab}*, L(R_a) = {1}{0}* and L(R_b) = {1}*{0}{1}*. We want to construct an NFA, M', that accepts σ(L(M)), where σ:{a,b}* → {0,1}* is the regular substitution defined by: σ(a) = L(R_a) and σ(b) = L(R_b).



Before constructing M' we observe we can make some simplifications because M' is, in general, an NFA rather than a DFA. Specifically, we can eliminate transitions that cannot possibly lead to acceptance. This yields a legal NFA, but would not be legal if we were building a DFA. In M we can immediately remove state 3 and the transitions that lead to it. In like fashion, and for the same reason, we can eliminate state 3' in R_a and 3" in R_b. The resulting NFAs are reproduced below.



Now to construct M' we disconnect transitions in M and rewire them by spontaneously transitions to the start states of a copy of the appropriate machine Ra or Rb. Since M only has one transition for each symbol, we only need one copy of Ra and Rb. The resulting NFA is shown below.



(e) L(M') = L – L'
(f) L(M') = L ∩ L'
(g) L(M') = L ∪ L'

As with all previous cases, let $M_1 = (Q_1, \Sigma_1, \delta_1, \alpha_1, A_1)$ and $M_2 = (Q_2, \Sigma_2, \delta_2, \alpha_2, A_2)$. Without any loss of generality we assume $\Sigma = \Sigma_1 = \Sigma_2$. To demonstrate each of these closure properties, M' will be a *cross-product machine*. That is, M' will be a DFA that simulates, in parallel, the behavior of each component DFA, M_1 and M_2, on the next input. Specifically, define $M' = (Q_1 \times Q_2, \Sigma, \delta', \alpha_1 \times \alpha_2, A')$, where $\delta' = \delta_1 \times \delta_2$, that is, $\delta'((q_1,q_2), a) = (\delta_1(q_1,a),\delta_2(q_2,a))$, for all $a \in \Sigma$.

(e) Now, to design M' so that it will accept $L - L' = L(M_1) - L(M_2)$, we define
$A' = \{ (q_1,q_2) \mid q_1 \in A_1 \text{ and } q_2 \in (Q_2 - A_2) \} = A_1 \times (Q_2 - A_2)$.

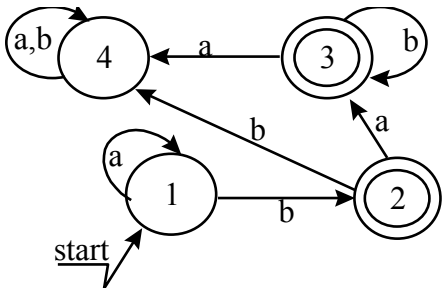
(f) To design M' to accept the intersection of these two languages, we define
$A' = \{ (q_1,q_2) \mid q_1 \in A_1 \text{ and } q_2 \in A_2 \} = A_1 \times A_1$.

(g) Finally, to design M' to accept the union of these languages, define
$A' = \{ (q_1,q_2) \mid q_1 \in A_1 \text{ or } q_2 \in A_2 \} = A_1 \times Q_2 \cup Q_1 \times A_2$.

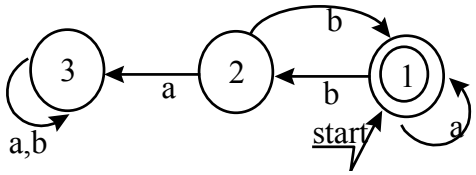
To prove L(M') is the desired combination of $L(M_1)$ and $L(M_2)$, one first proves that for any $q \in Q_1$ and any $p \in Q_2$, $\delta^*_{M'}((q, p), x) = (\delta^*_{M1}(q,x), \delta^*_{M2}(p,x))$. This can be done easily by induction on |x|. *We leave this as an exercise.*

To complete the proof for case (e) as an example, let $x \in L(M')$, then $\delta^*_{M'}((\alpha_1, \alpha_2), x) \in A'$, but this holds if and only if $\delta^*_{M'}((q, p), x) = (\delta^*_{M1}(\alpha_1,x), \delta^*_{M2}(\alpha_2,x)) \in A'$. This holds if and only if $\delta^*_{M1}(\alpha_1,x) \in A_1$ and $\delta^*_{M2}(\alpha_2,x) \in (Q_2 - A_2)$. But this is true if and only if $x \in L(M_1)$ and $x \notin L(M_2)$. The last conclusion depends on the fact that M_2 is a DFA, as we illustrated in the proof of complement (~) in part (a).

Example 20. (Cross Product Machine) Let M_1 and M_2 be the DFAs shown below.



M_1



M_2

The transition tables below define M' for the **difference** and **intersection**, respectively, of L_1 and L_2. Accepting states are marked with an "*", and transitions in the table that enter an accepting

state are shaded.

δ'	a	b
(1,1)	(1,1)	(2,2)
(2,1)	(3,1)	(4,2)
(3,1)	(4,1)	(3,2)
(4,1)	(4,1)	(4,2)
(1,2)	(1,3)	(2,1)
*(2,2)	(3,3)	(4,1)
*(3,2)	(4,3)	(3,1)
(4,2)	(4,3)	(4,1)
(1,3)	(1,3)	(2,3)
*(2,3)	(3,3)	(4,3)
*(3,3)	(4,3)	(3,3)
(4,3)	(4,3)	(4,3)

$L_1 - L_2$

δ'	a	b
(1,1)	(1,1)	(2,2)
*(2,1)	(3,1)	(4,2)
*(3,1)	(4,1)	(3,2)
(4,1)	(4,1)	(4,2)
(1,2)	(1,3)	(2,1)
(2,2)	(3,3)	(4,1)
(3,2)	(4,3)	(3,1)
(4,2)	(4,3)	(4,1)
(1,3)	(1,3)	(2,3)
(2,3)	(3,3)	(4,3)
(3,3)	(4,3)	(3,3)
(4,3)	(4,3)	(4,3)

$L_1 \cap L_2$

Exercise 10. Define the set of accept states for M' that will accept each of the following languages.

(a) $L_1 \cup L_2$
(b) $L_2 - L_1$

(h) L* (Kleene-*)

To show closure under Kleene-* it is easier, perhaps, to use a grammar construction. So, if L is Regular, then L = L(G), for some Right-linear grammar, G = (N, Σ, P, S). We wish to construct an RLG, G' = (N', Σ, P', S'), such that L(G') = L*. If this can be done, then by TTheoremT 3, L* must be Regular.

Define G as follows, N' = {S'} ∪ N and P' = { 1: S' → S, 2: S' → λ } ∪
{ X → wY | X,Y ∈ N and X → wY ∈P } ∪ { X → wS' | X → w ∈ P, a terminating rule of P}.

Suppose $x_1, x_2, \ldots x_n \in L$, for any $n \geq 1$. Then we must show that $x_1x_2 \ldots x_n \in L(G')$ and that $\lambda \in L(G')$. This will established that $L^* \subseteq L(G')$. Rule 2 of P' ensures that $\lambda \in L(G')$. So, suppose that $\pi_1, \pi_2, \ldots, \pi_n$ are derivations, respectively, of $x_1, x_2, \ldots x_n \in L$ in G. Furthermore, let $\mu_1, \mu_2, \ldots, \mu_n$ be corresponding derivations in G' constructed from $\pi_1, \pi_2, \ldots, \pi_n$, respectively, by replacing the last rule of the form, X →w, by X →wS'. Then the following is a derivation in G': $S' \overset{1}{\Rightarrow} S \overset{\mu_1}{\Rightarrow} x_1S' \overset{2}{\underset{\mu_2}{\Rightarrow}} x_1x_2S' \ldots \overset{\mu_n}{\Rightarrow} x_1x_2 \ldots x_nS' \overset{2}{\Rightarrow} x_1x_2 \ldots x_n$. Thus $x_1x_2 \ldots x_n \in L(G')$. Showing that $L(G') \subseteq L^*$ is straightforward and essentially involves reversing the argument we just presented. The details are left to the interested reader.

Example 21. Closure under Kleene-*.

Let L = L(G), where G = (N, Σ, P, S) is given by, N = {S, X, Y }, Σ = {a, b}, and P = {1: S → aX, 2: S → Y, 3: X → bbX, 4: X → λ , 5: Y → baS, 6: Y → b }. G' is given by: N' = {S', S, X, Y } and P' = {1: S' → λ, 2: S' → S } ∪ {3: S → aX, 4: S → Y, 5: X → bbX, 7: Y → baS } ∪ { 6: X → S' , 8: Y → bS' }. Rules 1 & 2 are unique to P'. Rules 3 – 8 of P' are obtained directly from rules 1 – 6 of G, respectively. Rules 1,2,3,5 of G are copied exactly, while rules 4 and 6 of G are modified by appending the start symbol S' to the right end to obtain rules 6 and 8 of G', respectively.

Observe that 1334 and 25256 are derivations in G of "abbbb" and "babab", respectively. Then the corresponding derivation of "abbbbbabab" in G' would be: 23556474781. Verify this for you self.

(i) LL' (Concatenation)

Concatenation follows a construction similar to that of Kleene-*. Let L = L(G_1) and L' = L(G_2), where $G_1 = (N_1, \Sigma_1, P_1, S_1)$ and $G_2 = (N_2, \Sigma_2, P_2, S_2)$ are Right-linear grammars. We also assume that their non-terminal alphabets are disjoint – note, too, that they may have different terminal alphabets. We wish to construct a Right-linear grammar, G' = (N', Σ', P', S') such that L(G') = LL'. To this end define $N' = N_1 \cup N_2$, $\Sigma' = \Sigma_1 \cup \Sigma_2$, $S' = S_1$, and $P' = P_2 \cup$ $\{ X \to wY \mid X,Y \in N_1 \text{ and } X \to wY \in P_1 \} \cup \{ X \to wS_2 \mid X \to w \in P_1$, a terminating rule of P_1 and S_2 is the start symbol of $G_2\}$. A derivation in G' must begin with the start symbol, S_1 of G_1 and continue in P_1 until a rule of the form r': $X \to wS_2$ is used. This sequence of rules duplicates a derivation of some string $x \in L(G_1) = L$, because r' is based on a rule r: $X \to w$ in G_1. Now the derivation in G' must continue using only rules of P_2. This sequence of rules must also define a derivation of some string y in $L(G_2) = L'$. The complete derivation in G' then results in a string $xy \in LL'$. Thus $L(G') \subseteq LL'$. Showing the reverse inclusion is left as an exercise for the interested reader.

Example 22. Closure under Concatenation.

Let L = {a}* and L' = {b}*. $G_1 = (\ \{S_1\}, \{a\}, \{1: S_1 \to aS_1, 2: S_1 \to \lambda\ \}, S_1)$ and $G_2 = (\ \{S_2\}, \{b\}, \{1': S_2 \to bS_2, 2': S_2 \to \lambda\ \}, S_2)$.
Then G' = ($\{S_1, S_2\}$, {a,b}, {1: $S_1 \to aS_1$, 2: $S_1 \to S_2$, 3: $S_2 \to bS_2$, 4: $S_2 \to \lambda$ }, S_1).

Observe that 1112 is a derivation of "aaa" in G_1 and 1'1'2' is a derivation of "bb" in G_2. In G' the corresponding derivation is 1112334, and the resulting string is "aaabb".

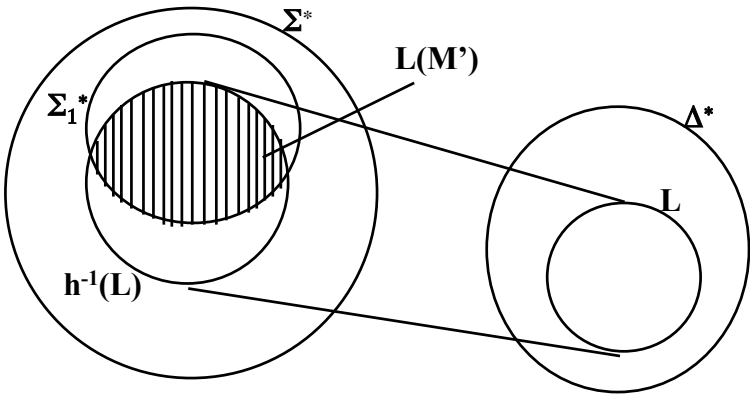
T**Theorem**T **5.** Let h: $\Sigma^* \to \Delta^*$ be a homomorphism. Then the *inverse homomorphism defined by h* is a mapping, h^{-1}: $\Delta^* \to \Sigma^*$, defined by: for all $x \in \Delta^*$, $h^{-1}(x) = \{ y \in \Sigma^* \mid h(y) = x \}$. h^{-1} is extended to subsets of Δ* in the obvious way. Then, $\mathbf{R}_\varepsilon$ is closed under inverse homomorphism

on Σ.

Proof. A homomorphism h is said to be *erasing* if and only if $h(a) = \lambda$, for some $a \in \Sigma$.

To illustrate, let h: $\{a,b,c\}^* \rightarrow \{0,1\}^*$ be given by $h(a) = \lambda$, $h(b) = 1010$, $h(c) = 10$. Then $h^{-1}(101010) =$ $\{ ucvcwcx \mid u,v,w,x \in \{a\}^* \} \cup \{ ucvbx \mid u,v,x \in \{a\}^* \} \cup \{ ubvcx \mid u,v,x \in \{a\}^* \}$. Thus, if h is an erasing homomorphism, $h^{-1}(x)$ can be an infinite set. In fact, $h^{-1}(x)$ is infinite if and only if h is erasing. Also note that $h^{-1}(\lambda) = \{\lambda\}$ if and only if h is non-erasing, that is, $h(a) \neq \lambda$, for all $a \in \Sigma$.

To show that $h^{-1}(L)$ is regular if L is regular we assume that L = L(M), for some DFA, $M = (Q, \Delta, \delta, q_0, A)$, and let h: $\Sigma^* \rightarrow \Delta^*$ be a homomorphism. Define $\Sigma_0 = \{ a \in \Sigma \mid h(a) = \lambda \}$ and $\Sigma_1 = \Sigma - \Sigma_0$. We want to define an DFA, M' = (Q', Σ1, δ', q'0, A'), such that $L(M') = h^{-1}(L) \cap (\Sigma_1)^+$. First we define, M1 = (Q $\cup \{\alpha\}$, Σ1, δ1, A), where α is a new start state, $\alpha \notin A$, and $\delta 1(\alpha,a) = \delta(q0,a)$, for all $a \in \Sigma$. It should be clear that L(M1) = L(M) - $\{\lambda\}$. Now we construct M' from M1 as follows: Q' = $\{ q \in Q1 \mid$ q is reachable from $\alpha \}$, A' = Q' $\cap$ A, q'0 = α, and δ': Q' $\times \Sigma \rightarrow$ Q' is defined by, $\delta'(q, a) = \Delta_{M_1}(q, h(a))$, for all $a \in \Sigma 1$. Since the start state of Q' is not an accepting state, it follows that L(M') does not contain λ. Furthermore, $x = a1a2\ldots an \in (\Sigma_1)^+$, is accepted by M' if and only if $\Delta_{M_1}(\alpha, h(a1)h(a2)\ldots h(an)) \in A$ if and only if $h(x) \in L(M1) = L(M) - \{\lambda\}$. Thus $L(M') = h^{-1}(L) \cap (\Sigma_1)^+$. The diagram below illustrates what has been established so far.



The strings in $h^{-1}(L) - L(M')$ include $h^{-1}(\lambda) = (\Sigma_0)^*$, if $\lambda \in L$, and all strings of the form $x_0a_1x_1a_2x_2\ldots a_nx_n$ where $x_i \in (\Sigma_0)^+$ and $a_1a_2\ldots a_n \in L(M')$. Thus

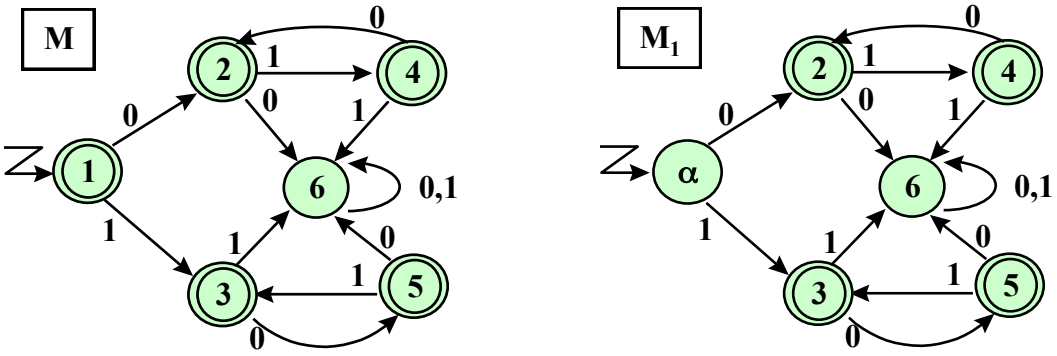
[1] $h^{-1}(L) = \sigma(L(M')) \cup (\Sigma_0)^*$, if $\lambda \in L$, or
[2] $h^{-1}(L) = \sigma(L(M'))$, if $\lambda \notin L$,

where σ: $(\Sigma_1) \rightarrow \Sigma^*$ is the substitution given by $\sigma(a) = (\Sigma_0)^*\{a\}(\Sigma_0)^*$, for all $a \in \Sigma_1$. Since the regular languages are closed under union and substitution, it follows that $h^{-1}(L)$ is regular. (NOTE: the representation for $h^{-1}(L)$ in terms of σ and Σ_0 is valid even in the case Σ_0 is empty.)

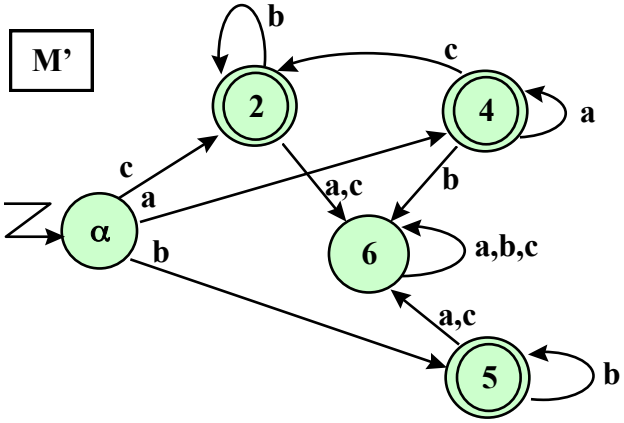
To complete the proof, we define DFA, M'', such that $L(M'') = h^{-1}(L)$. If [1] applies, then M'' can be obtained from M' by making two simple changes: (a) make α an accept state, and (b) extend δ' to $\Sigma = \Sigma_1 \cup \Sigma_0$ by defining $\delta'(q,b) = q$, for all $q \in Q'$ and all $b \in \Sigma_0$. If [2] applies, then we must ensure that $(\Sigma_0)^* \notin L(M'')$. To this end, apply the construction as in [1] but do not make α an accept state; that is, do only part (b).

Example 23. (Inverse Homomorphism)
Let h: {a,b,c,d,e}* → {0,1}* be the homomorphism defined by: h(d) = h(e) = λ, h(a) = 01, h(b) = 10, h(c) = 0. Let L = { x ∈{0,1}* | x = uvw and |v| = 2 ⇒ v ∈{01,10}}. Construct an NFA or DFA that accepts $h^{-1}(L)$. **Note:** Σ_0 = {d,e} and Σ_1 = {a,b,c}

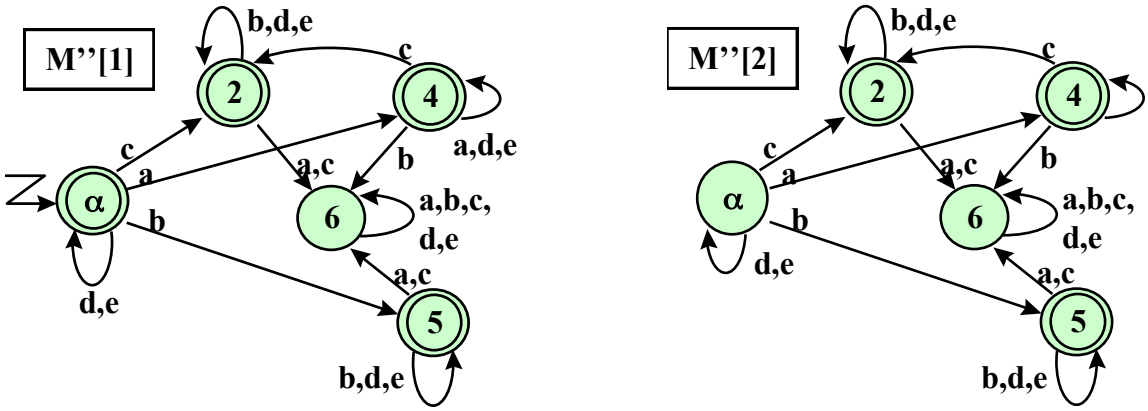
Step 1: Construct a DFA, M, for L. **Note:** λ ∈L.



Step 2: Construct the DFA for M'. Note that state 3 of M_1 becomes inaccessible in M'.

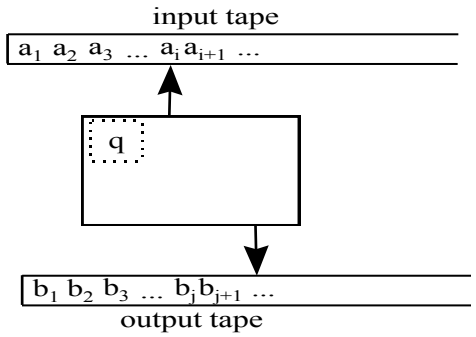


Step 3: $h^{-1}(L) = \sigma(L(M')) \cup (\Sigma_0)^*$, where $\sigma(a) = (d+e)^*a(d+e)^*$, $\sigma(b) = (d+e)^*b(d+e)^*$, and $\sigma(c) = (d+e)^*c(d+e)^*$. Note: $(\Sigma_0)^* = L[(d+e)^*]$. To obtain the DFA, M'', simply make α an accept state and for each symbol in $\Sigma_0 = \{d,e\}$ and each state add a transition that leaves M'' in the same state (a self-loop). This machine is shown below as M''[1]. If $\lambda \notin L$, then we obtain the machine illustrated in M''[2].



The next definition introduces the notion of a finite state machine that produces output. *Sequential Transducer*s, as they are called, model a class of translators or encoding devices that translate strings over some input alphabet into strings over some output alphabet. Sequential Transducers (STs) are, in their most general form, non-deterministic finite state automata that can output a string on each transition (spontaneous and reading). Deterministic versions, called *Mealy* machines and *Moore* machines, can be defined by imposing restrictions on the ST model. The reader will also note that substitutions and homomorphisms accomplish essentially the same result, but, as we shall see, are really special instances of sequential transducer maps.

Definition 19. A ***Sequential Transducer*** is a 6-tuple, S = (Q, Σ, Δ, δ, Q_0, A), where Q = finite, non-empty set of states; Σ = input alphabet; Δ = output alphabet; $Q_0 \subseteq Q$, the set of initial states; $A \subseteq Q$, the set of accepting states; and $\delta : Q \times (\Sigma \cup \{\Lambda\}) \rightarrow Q \times \Delta^*$ is the transition relation. Specifically, if S is in state q, then it can choose either to read a $\in\Sigma$ from its input tape, or ignore its input (Λ), in making a transition to one of several possible next states. In addition, the result of making either type of transition is to write some string y (possibly λ)to its output tape. If S can enter an accepting state after having read all its input (just like an NFA), then the contents of its output tape is a valid translation of that input. This is illustrated in the figure below.



More formally, we define a configuration of S to be a 3-tuple, (q, x, y), where q denotes the current state of S, x denotes the remaining input (left most symbol next to be read), and y the current contents of its output tape (right end most recently written).

Similar to NFAs, we define the move relation ($\Rightarrow_S$) on configurations of S as follows:
[1] $(q, x, y) \Rightarrow_S (q', x, yu)$ if and only if $(q', u) \in \delta(q, \Lambda)$;
[2] for $a \in\Sigma$, $(q, ax, y) \Rightarrow_S (q', x, yu)$ if and only if $(q',u) \in \delta(q, a)$.
Finally, we define the language recognized by S to be

$$L(S) = \{ x \in\Sigma^* \mid (p, x, \lambda) (\Rightarrow_S)^* (f, \lambda, y) \text{ for some } p \in Q_0 \text{ and } f \in A \}$$

For $x \in L(S)$, we define the translation of x produced by S to be

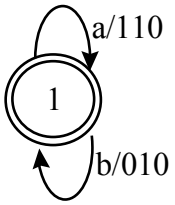
$$S(x) = \{ y \in\Delta^* \mid (p, x, \lambda) (\Rightarrow s)^* (f, \lambda, y) \text{ for some } p \in Q_0 \text{ and } f \in A \}$$

Finally, for some $L \subseteq \Sigma^*$ we define the translation of L to be

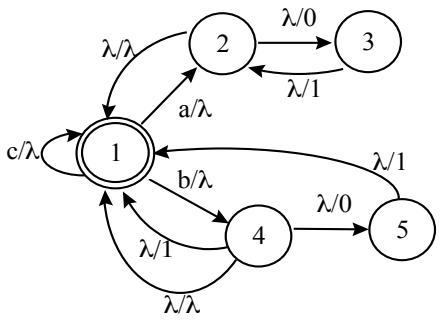
$$S(L) = \bigcup_{x \in L \cap L(S)} S(x)$$

Example 24. Consider the homomorphism h:{a,b}* → {0,1}* defined by h(a) = 110, h(b) = 010. Then the following sequential transducer realizes the same mapping as h.

S = ({q_1},{a,b},{0,1}, δ, {q_1}, {q_1}), where δ is illustrated by the state transition diagram below. Observe how transitions are represented in this diagram. Since S accepts all of {a,b}* it behaves exactly like a homomorphism by writing h(a) to its output tape whenever a is read from its input tape and writing h(b) whenever b is read from the input.



Example 25. Consider the regular substitution, σ: {a,b,c}* → {0,1}*, where σ(a) = {01}*, σ(b) = {λ,1,01}, σ(c) = {λ}. Observe that the sequential transducer can produce an unbounded sequence of output symbols for every input symbol by making any number of transitions that ignore the input and write some string to its output tape. Thus sequential transducers can also emulate regular substitutions.



Theorem 6. $\mathbf{R}_\Sigma$ is closed under sequential transducer maps. That is, if L is a regular subset of Σ^* and S: $\Sigma^* \rightarrow \Delta^*$ is a sequential transducer map, then S(L) is regular.

Proof. Because a formal proof is somewhat involved, we only sketch the salient details. First observe that we can describe a transition of S as a 4-tuple, $t = (q_1, \sigma, y, q_2)$, where q_1 denotes the current state of S, $\sigma \in \Sigma \cup \{\Lambda\}$ the input read (if any), $y \in \Delta^*$ the output written, and q_2 the next state. If we think of these "tuples" as symbols in a third alphabet, Ω, then we can see that a string $z \in \Omega^*$ represents a valid sequence of transitions in S if and only if $z = (q_1, a_1, y_1, q_2)(q_2, a_2, y_2, q_3) \ldots (q_n, a_n, y_n, q_{n+1})$. That is, for each tuple the current-state component must agree with next-state of the preceding tuple, and the current-state component of the first tuple must be one of the initial states of S. If z is to represent an accepting computation of S, then, of course, the next-state component of the last tuple must be an accepting state of SF^3F. It should therefore not be difficult to see how to define a DFA, $M = (Q', \Omega, \delta', q_0, A')$, that accepts only strings in Ω^* that represent accepting computations of S. Clearly, L(M) is regular.

Next we define a homomorphism, h: $\Omega^* \rightarrow \Delta^*$, given by $h(t) = y$, where $t = (q_1, a, y, q_2) \in \Omega$. In other words, h simply replaces t by its output-component, y. And, we define a homomorphism, g: $\Omega^* \rightarrow \Sigma^*$, given by $g(t) = a$, where $t = (q_1, a, y, q_2) \in \Omega$.

Given some regular set $L \subseteq \Sigma^*$, we now claim that $S(L) = h(L(M) \cap g^{-1}(L \cap L(S)))$. $L \cap L(S)$ is the subset of L accepted by S and is a regular set because L was assumed to be regular and L(S), the language accepted by S, is regular. Applying g^{-1} to this set gives all strings in Ω^* that would map to an input accepted by S. Intersecting this set with L(M) yields all the accepting computations of S corresponding to some accepted input. Finally, applying the homomorphism h to this set of computations yields the set of outputs produced by accepting computations of S only for inputs from $L \cap L(S)$. To conclude that S(L) is regular requires closure of $\mathbf{R}_\Sigma$ for intersection, homomorphism, and inverse homomorphism.

Exercise 11. We have seen that sequential transducer maps can emulate homomorphism and regular substitution. What other closure operations can sequential transducers emulate?

[3] Actually, acceptance is a bit more complicated than this. We require that the next-state component be an accept state of S sometime after the last read transition. A DFA can check this by remembering when an accept state has been reached, and then whether any read transitions of S occur thereafter.

Context-free Grammars and Languages

Definition 25. A *Context-free grammar (CFG)* is a PSG, G = (N, Σ, P, S), where $P \subseteq N \times V_G^*$. A production $r \in P$ is denoted r: $X \rightarrow w$ and means that an occurrence of X can be replaced by w in any context in which X occurs - this is where the term "context free" originates - rewriting X "free of any contextual constraints." **NOTE**: Context-free grammars were defined by Chomsky to be Phrase Structure grammars with the ***Type-2*** restriction. *CFG* and *Type-2* grammars are synonymous terms.

The family of all languages defined by Context-free grammars is called the family of Context-free Languages (CFLs).

Example 31. G_e = ({E, T, F, X}, {n,v, +, -, *, /, (,)}, P, E), where

E = { 1: E → E + T,
2: E → E – T,
3: E → T,
4: T → T∗ F,
5: T → T/ F,
6: T → F,
7: F → +X
8: F → –X
9: F → X
10: X → n,
11: X → v,
12: X → (E),
}

$L(G_e)$ = { x ∈{n, v, +, -, *, /, (,)}* | x denotes a well-formed arithmetic expression over the operator symbols {+, –, ∗, /} and operand symbols {v, n}allowing parenthesized sub-expressions nested arbitrarily deep.}

A derivation of x = (n+v)∗n ∈ $L(G_e)$ is illustrated below. In general, x ∈ L(G) may have several distinct derivations. **Note**: ⇒ refers to G_e

[1] $E \overset{3}{\Rightarrow} T \overset{4}{\Rightarrow} T*F \overset{6}{\Rightarrow} F*F \overset{9}{\Rightarrow} X*F \overset{12}{\Rightarrow} (E)*F \overset{1}{\Rightarrow} (E+T)* F \overset{3}{\Rightarrow} (T+T)*F \overset{6}{\Rightarrow} (F+T)*F \overset{9}{\Rightarrow} (X+T)*F \overset{10}{\Rightarrow} (n+T)*F \overset{6}{\Rightarrow} (n+F)*F \overset{9}{\Rightarrow} (n+X)*F \overset{11}{\Rightarrow} (n+v)*F \overset{9}{\Rightarrow} (n+v)*X \overset{10}{\Rightarrow} (n+v)*n$

Exercise 14: Show that 3469(10)9(12)169(11)369(10) is also a derivation for (n+v)∗n in G_e. Can you construct yet another derivation for (n+v)∗n in G_e that is distinct from either of the ones already identified?

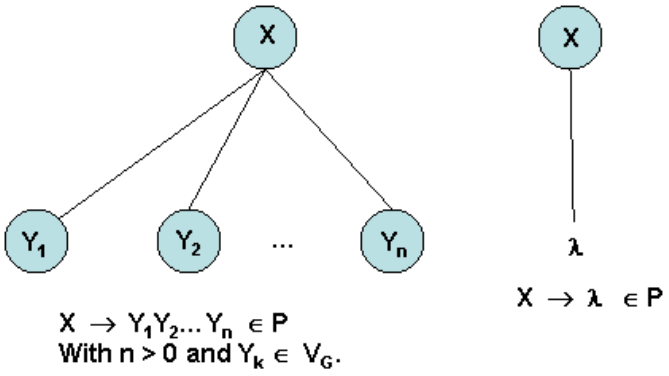
Definition 26. Let G = (N,Σ,P,S) be a Context-free grammar. A derivation, $\pi \in P^*$ in G is said to be *leftmost (rightmost)* if and only if every rule of π rewrites the leftmost (rightmost) nonterminal occurring in the sentential form defined at each step.

Example 32. Verify that derivation [1] above is a leftmost derivation of (n+v)*n, and [2] is a rightmost derivation of (n+v)∗n in G_e.

[2] $E \overset{3}{\Rightarrow} T \overset{4}{\Rightarrow} T*F \overset{9}{\Rightarrow} T*X \overset{10}{\Rightarrow} T*n \overset{6}{\Rightarrow} F*n \overset{9}{\Rightarrow} X*n \overset{12}{\Rightarrow} (E)*n \overset{1}{\Rightarrow} (E+T)*n \overset{6}{\Rightarrow}$
$(E+F)*n \overset{9}{\Rightarrow} (E+X)*n \overset{11}{\Rightarrow} (E+v)*n \overset{3}{\Rightarrow} (T+v)*n \overset{6}{\Rightarrow} (F+v)*n \overset{9}{\Rightarrow} (X+v)*n \overset{10}{\Rightarrow} (n+v)*n$

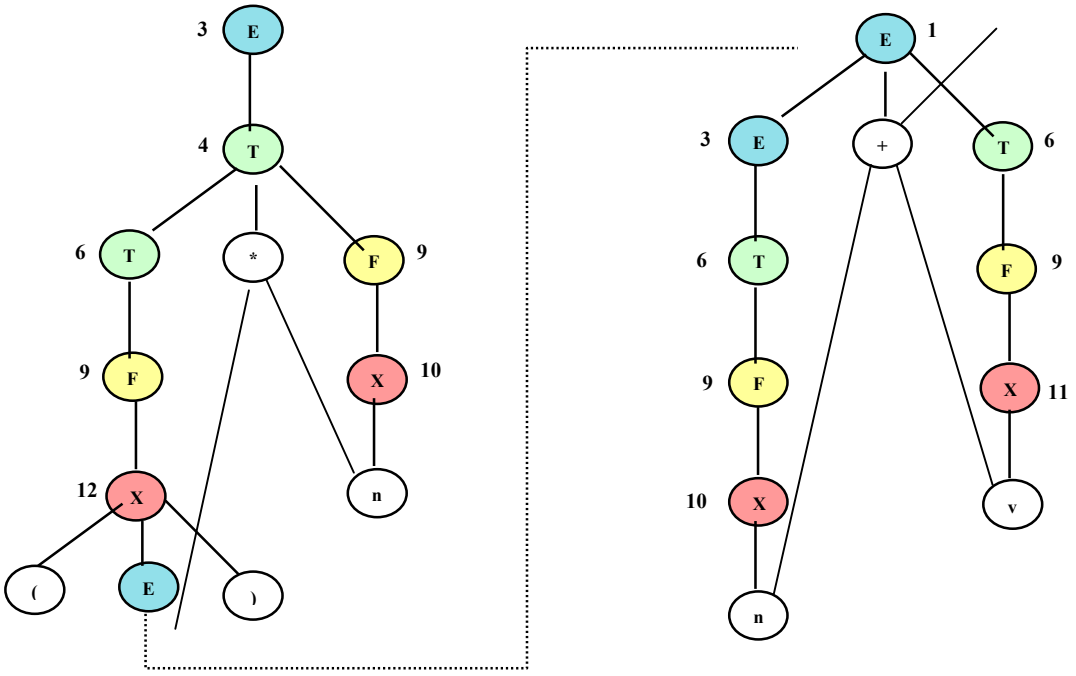
π(rightmost) = 349(10)69(12)169(11)369(10)

Defintion 27. Let G = (N,Σ,P,S) be a Context-free grammar. Let $x \in L(G)$. A syntax tree for x is a two-dimensional representation of a derivation of x as illustrated in the diagram below. Each rule in the derivation of x that is used to rewrite a non-terminal is expanded as a subtree rooted at the nonterminal (in the figure, X is expanded by the rule: $X \rightarrow Y_1Y_2\ldots Y_n$ or $X \rightarrow \lambda$).



The syntax tree is complete and valid if the root node is the start symbol of G and each nonterminal in the tree is expanded by one of its rules in G. The *frontier of a syntax tree* is the sequence of leaves (terminal symbols or λ) enumerated in a left-to-right order.

Example 33. A syntax tree for the expression: (n+v)∗n, as defined by the grammar, G, in Example 31, is given below.



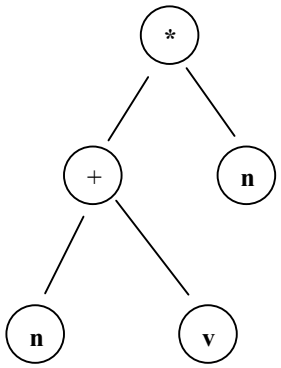
Observe that when given a syntax tree for an expression, it is very easy to generate a leftmost and rightmost derivation of the expression. A leftmost derivation is generated by a pre-order traversal of the tree, top-down, left-to-right, listing the rule used to expand each nonterminal on the first visit to that node. A rightmost derivation is generated by a top-down, right-to-left post-order traversal in the same fashion.

Syntax trees are important in compiling and translating programming languages. Compilers, for example, typically construct an *Operator tree* from (or instead of) a syntax tree and use the Operator tree for allocating hardware registers or temporary variables needed to evaluate the associated expression. The shape and structure of the Operator tree also determines the order in which operands and sub-expressions will be evaluated.

Definition 28. An ***Operator Grammar,*** is a CFG, G = (N, Σ,P,S), where $\Sigma = \Sigma_{opr} \cup \Sigma_{opnd} \cup \Sigma_{brk}$. The terminals are respectively, the ***operators***, the ***operands***, and the ***brackets***. Brackets are matching pairs of one or more types of parentheses. Every rule has one of the following forms:

[1] **X → Y θ Z**, where $\theta \in \Sigma_{opr}$ and Y, Z ∈ N (binary operators)
[2] **X → Y θ**, where $\theta \in \Sigma_{opr}$ and Y∈ N (postfix unary operators)
[3] **X → θ Y,** where $\theta \in \Sigma_{opr}$ and Y∈ N (prefix unary operators)
[4] **X → Y,** where Y∈ N
[5] **X → a,** where $a \in \Sigma_{opnd}$,
[6] **X → (Y),** where **(,)** are matching pairs in Σ_{brk}.

If T is a syntax defined by an Operator Grammar, then it is frequently desirable to construct its equivalent ***Operator Tree(OT).*** The Operator Tree constructed from the syntax tree shown in Example 33 is illustrated below.



The shape of the operator tree determines the order in which operators are applied to their operands. The rules for evaluating an operator tree top-down are the following.

Operator Tree Evaluation: Value(T)
Let T be an operator tree.
(1) If root(T) is an operand, then Value(T) = content(T).
(2) If root(T) is an operator, θ, then let S_L and S_R be the operator subtrees denoting, respectively, the left operand expression and the right operand expression. Let x = Value(S_L) and y = Value(S_R). Then Value(T) = θ(x,y). **NOTE:** if θ is a unary operator, only S_L is defined and evaluated.

The operator tree associated with a given expression defines its *evaluation semantics*. Since the operator tree is extracted from the syntax tree, it is logical to conclude that the structure of the grammar somehow encodes the evaluation semantics. Indeed this is the case. In fact, the *relative precedence* of operators (sometimes called *binding strength*) and their *associativity properties* are encoded in the order and form of rules used to introduce operators. Specifically,

X + Y * Z (+ has lower binding strength than *)(+ has lower precedence compared to *)
X + Y + Z (Left associativity defines relative binding strength for operators with the same precedence)

Rule 1: All operators having the same binding strength (precedence) should be introduced by the same nonterminal. Therefore, if the operator set is partitioned into k strength (precedence) groups, then k distinct nonterminals should be defined, one to introduce each group of operators. For example, the grammar G_e of Example 31 has 3 precedence groups binary{+, -}, binary{*,/} and unary{+,-}. The nonterminal (and start symbol), E, introduces the first group, T introduces the second group, and F introduces the third group.

Rule 2: The set rules introducing each precedence group should be linked by a single unit rule (X → Y) and ordered so that the precedence group having the lowest binding strength is introduced by the start symbol, and the precedence group having the highest binding strength is

introduced by the last nonterminal. The unit rule, X → Y, suggests that the operator group introduced by Y has precedence level one higher than the group introduced by X.

Rule 3: Within each precedence group, all operators should have the same associativity*: Left, Right, None*. Left associativity is achieved by using left-recursive rules to introduce the operators, while right-associativity is achieved using right-recursive rules. For example, E → E + T, is a left recursive rule because the nonterminal in the leftpart is also the leftmost symbol of the rightpart. Some operators are not associative, e.g., negation. Non-associative operators should be introduced by non-recursive rules. Note: For unary operators, associativity depends on whether they are prefix or posfix operators – associative prefix operators are always right-associative, while associative postfix operators are left-associative.

X ^ Y ^ Z = (X ^ Y)^Z = $(X^Y)^Z = X^{Y*Z}$ (X ^ (Y ^ Z)) = X^{Y^Z}
Boolean-exp ? X1 : X2 B ? (B' ? Y1 : Y2) : X1 : X2
-X (do not want ------X)

Rule 4: Finally, a unique nonterminal should be defined to introduce all operand forms including nested subexpressions. For example, in G_e, X is used to introduce operands n,v, and subexpressions (E).

Example. 34. Consider designing an Operator Grammar that realizes the following operator precedence and associativity properties (L, R, N).

Precedence (Low to High): BL{ &, | } < BN{ <, >, = } < UR{*, # } < UN{ $, @ }.
Operand symbols: a, b.
P = {
1: S → S **&** T //Binary Left Associative of lowest precedence
2: S → S | T
3: S → T
4: T → F < F //Binary Non-associative of next lowest precedence
5: T → F > F
6: T → F = F
7: T → F
8 : F → *F //Unary Associative (prefix) operators of next higher precedence
9 : F → #F
10 : F → X
11 : X → Y$ //Unary non-associative (postfix) operators of next higher precedence
12: X → Y@
13: X → Y
14 : Y → a //Operands
15: Y → b
16: Y → (S) //nested subexpressions
}

Ambiguous vs. Unambiguous Grammars

Context-free languages and grammars have special significance to computer science because they provide the basis for the specification of programming languages. All well-known programming languages have a syntax defined by a member of the class of unambiguous Context-free grammars. Unambiguous grammars are necessary for defining programming languages for two primary reasons: (a) every string (program) in the language has exactly one *parse*; (b) syntax analysis can be performed in *linear time* (time proportional to the length of the source program.)

A parse is the inverse of a derivation. That is, parsing is the process a compiler uses to reconstruct a derivation of the source program with respect to the rules of the underlying Context-free grammar that defines the programming language in which the source program is written. When a compiler reports a syntax error, then it has failed in this process. If the parsing process succeeds, the compiler has implicitly reconstructed the derivation of the source program.

Parsing algorithms fall generally into two categories:

- **Top-down alorithms** are distinguished as algorithms that simulate a leftmost derivation in some underlying CFG that defines the language. Examples of such algorithms are *Recursive Descent* and *LL(k)*. The former uses backtracking (trial and error); this applies to a large family of grammars, but is very inefficient. LL(k) parsers are based on LL(k) grammars. These parsers are very efficient, but apply to a very limited class of grammars. (NOTE: LL(k) means **L**eft-to-right scan of input producing a **L**eftmost derivation using k symbols (tokens) of input lookahead.

- **Bottom-up algorithms** are distinguished as algorithms that simulate the reverse of a rightmost derivation. Examples are the Weak-precedence algorithms and the LR(k) algorithms, identified with grammar classes by the same name. WP algorithms are simple and efficient, but apply to a small class of grammars relative to the LR(k). WP parser use one (1) symbol of lookahead. LR(k) parsers use k symbols of lookahead, although all practical implementations use k = 1. The well-known UNIX tool, YACC, is a parser generator based for LR(1) grammars.

The semantics (meaning)(computational behavior) of a program is defined in terms of the rules comprising its derivation. Thus, if there is more than one derivation of a program, there could possibly be more than one meaning assigned to the program - and the meaning assigned by the compiler may not be the meaning the programmer intended - it is also possible that two different compilers might not assign the same meaning to the same program. Clearly, such “ambiguity” in the behavior of computer programs is not desired - it is a kind of nondeterminism we cannot tolerate. So, computer scientists have invested considerable research effort in identifying and classifying unambiguous Context-free grammars. The next definition formally defines this important class of grammars.

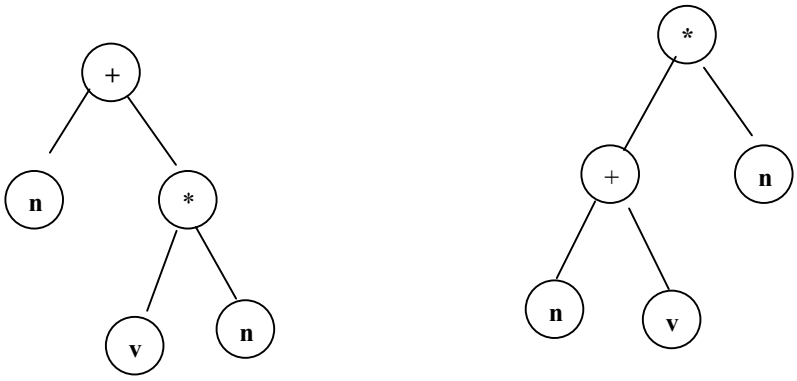
Definition 29. A Context-free grammar, G = (N,Σ,P,S), is said to be *unambiguous* if and only if for every x ∈L(G), x has exactly one (leftmost derivation)(rightmost derivation)(syntax tree).

To illustrate the principle that *syntactic ambiguity can lead to semantic ambiguity*, consider the following grammar.

Example 34. G' = (N, Σ, P', S), where P' =

{ 1: S → S + S,
2: S → S - S,
3: S → S * S,
4: S → S / S,
5: S → n,
6: S → v
7: S → (S) }

The expression, n+v*n, has two distinct leftmost derivations, π1 = 15365 and π2 = 31565. The operator trees that correspond to these derivations are shown below. The evaluation semantics is clearly different for the two trees.



Several *important families of unambiguous grammars* have been found, the *Weak Precedence* grammars, the *LR(k) grammars*, and the *LL(k) grammars*. Grammars from these families are almost always used as the basis for defining programming languages. However, there is some bad news. Not all Context-free languages have an unambiguous grammar that defines them (these are called *inherently ambiguous languages*) -clearly we do not want to uses these languages as models of programming languages. Worse yet, there does not exist an algorithm for detecting such languages. The good news is that the classes of unambiguous grammars mentioned above are relatively easy to design and so these negative results do not have any significant practical consequences.

Definition 30. A Context-free grammar, G = (N, Σ, P, S), is said to be *linear* if and only if P ⊆ N × Σ*(N∪{λ}) Σ*. G is said to be *right-linear* if and only if P ⊆ N × Σ*(N∪{λ}), and *left-linear* if and only if P ⊆ N × (N∪{λ}) Σ*.

Example 35. Linear, right-linear, and left-linear grammars for the language, L = aa*#(ba)*.

$G_1 = (N_1, \{a,b,\#\}, P_1, S)$ is a linear grammar for L. $G_2 = (N_2, \{a,b,\#\}, P_2, S)$ is a right-linear grammar for L.
$G_3 = (N_3, \{a,b,\#\}, P_3, S)$ is a left-linear grammar for L.

$P_1 = \{ 1: S \rightarrow aX, 2: X \rightarrow aX, 3: X \rightarrow Xba, 4: X \rightarrow \# \}$
$P_2 = \{ 1: S \rightarrow aX, 2: X \rightarrow aX, 3: X \rightarrow \#Y, 4: Y \rightarrow baY, 5: Y \rightarrow \lambda \}$
$P_3 = \{ 1: S \rightarrow Y, 2: Y \rightarrow Yba, 3: Y \rightarrow X\#, 4: X \rightarrow Xa , 5: X \rightarrow a \}$

Exercise 15: Prove that G_1, G_2, and G_3 are unambiguous grammars!

Theorem 14. Every left-linear Context-free grammar is equivalent to some right-linear Context-free grammar and conversely.

Proof. Let G = (N, Σ, P, S) be a right-linear CFG. We describe a construction that produces an equivalent left-linear grammar. In doing so, we establish that any language defined by a right-linear grammar can also be defined by a left-linear grammar. A similar and analogous construction can be used to reverse this argument. We leave the converse construction and proof as an exercise for the reader.

Let n = |P| and define G' = (N ∪ {S'}, Σ, P', S') from G, where |P'| = n+1, as follows.

$P' = \{k:S' \rightarrow X\,w \mid k:X \rightarrow w \in P, \text{ where } w \in \Sigma^* \}_{[1]}$
$\quad \cup \{k:Y \rightarrow Xw \mid k:X \rightarrow wY \in P, \text{ where } w \in \Sigma^*\}_{[2]}$
$\quad \cup \{ n+1: S \rightarrow \lambda \}_{[3]}$

Claim: $S \Rightarrow xY \overset{k}{\Rightarrow} xw$, where $xw \in \Sigma^*$, if and only if $S' \overset{k}{\Rightarrow} Yw \Rightarrow S\,xw \overset{[3]}{\Rightarrow} xw$, in G', where $\theta = \pi^{rev}$

Claim can be established by a straightforward induction on the $|\pi|$ noting that the terminating production **k** in G corresponds to a production k in set $P'_{[1]}$, and each G-production k appearing in π corresponds 1-1, but in the reverse order, with a G'-production k in $\theta = \pi^{rev} \in (P'_{[2]})^+$. The remaining details are left to the reader.

Exercise. Let G = (N, Σ, P, S) be a left linear grammar. Give a formal specification of the equivalent right linear grammar as was done in the proof of Theorem 14 above.

Definition 31. A production, k: X → w, of a Context-free grammar, G, is said to be *useless* if and only if one of the following is true about rule k:
(a) $\forall \pi \in P^*$, $S \Rightarrow \alpha$ implies $\alpha \neq uXv$, for any $u,v \in V_G^*$ (X is *unreachable*)
(b) $\forall \pi \in P^*$, $w \Rightarrow \beta$ implies $\beta \notin \Sigma^*$. (w is *non-terminating*)

Our next theorem gives an algorithm for identifying and removing all useless productions from a arbitrary Context-free grammar. The resulting grammar is said to be in *reduced normal form*.

Theorem 15. Let G = (N, Σ, P, S) be a Context-free grammar. Then G is equivalent to a Context-free grammar, G' = (N', Σ, P', S), where $N' \subseteq N$ and $P' \subseteq P$. G' is said to be the *reduced form of G*. Also observe that if G is linear (right linear)(left linear), then G' will be too.

Proof. Apply algorithm F-1a to G to obtain $G_1 = (N_1, \Sigma, P_1, S)$. Then either G_1 is the desired grammar G', or algorithm F-1b must be applied to G_1 to obtain G'.

ALGORITHM F-1a: Eliminate non-terminating (non-generating) rules.
Step 1: Set $Q_0 = \{ X \rightarrow w \in P \mid w \in \Sigma^* \}$.
Set $Z_0 = \{ X \in N \mid X \rightarrow w \in Q_0 \}$. Set k = 0.
Step 2: Set $Q_{k+1} = Q_k \cup \{ X \rightarrow w \in P \mid w \in (\Sigma \cup Z_k)^* \}$.
Set $Z_{k+1} = Z_k \cup \{ X \in N \mid X \rightarrow w \in Q_{k+1} \}$.
Step 3: **If** $(Q_{k+1} - Q_k) \neq \Phi$ **Then** { k := k+1; **goto Step 2**;} // Have we converged?
Step 4: **If** $S \in Z_k$ Then {G' = Algorithm 1b applied to: $G_1 = (Z_k, \Sigma, Q_k, S)$;}
else {G' = ({S}, Σ, Φ, {S});} // $L(G) = L(G') = \Phi$
Step 5: Halt!

ALGORITHM F-1b: Eliminate unreachable non-terminals.
Step 1: Set $Z_0 = \{S\}$. Set $Q_0 = \{ S \rightarrow w \in P_1 \}$. Set k = 0.
Step 2: Set $Z_{k+1} = Z_k \cup \{X \in N_1 \mid Y \rightarrow uXv \in Q_k$, for some strings u and v$\}$.
Set $Q_{k+1} = Q_k \cup \{ X \rightarrow w \in P_1 \mid X \in Z_{k+1} \}$.
Step 3: **If** $(Z_{k+1} - Z_k) \neq \Phi$ **Then** { k := k+1; **goto Step 2**;} **else** { G' = (Z_k, Σ, Q_k, S); Halt!}

Example 36. Consider the grammar G = (N, Σ, P, S), where N = {S, A, B, C, X, Y, Z}, Σ = {a, b}, and
P = {
1: $S \rightarrow aSX$
2: $S \rightarrow AB$
3: $X \rightarrow Y$
4: $Y \rightarrow bY$
5: $Y \rightarrow ZX$
6: $Z \rightarrow aa$
7: $Z \rightarrow \lambda$
8: $A \rightarrow Cb$
9: $C \rightarrow B$
10: $B \rightarrow BCa$
11: $B \rightarrow \lambda$
}

To construct the reduced form of G we first run Algorithm F-1a to identify and eliminate any non-terminating rules.
Step 1a: Set $\mathbf{Q_0} = \{ X \rightarrow w \in P \mid w \in \Sigma^* \} = \mathbf{\{6, 7, 11\}}$
Set $\mathbf{Z_0} = \{ X \in N \mid X \rightarrow w \in Q_0 \} = \mathbf{\{Z, B\}}$. Set k = 0.
Step 2a: Set $\mathbf{Q_1} = Q_0 \cup \{ X \rightarrow w \in P \mid w \in (\Sigma \cup Z_0)^* \} = Q_0 \cup \mathbf{\{9\}}$.
Set $\mathbf{Z_1} = Z_0 \cup \{ X \in N \mid X \rightarrow w \in Q_1 \} = Z_0 \cup \mathbf{\{C\}}$.
Set $\mathbf{Q_2} = Q_1 \cup \{ X \rightarrow w \in P \mid w \in (\Sigma \cup Z_1)^* \} = Q_1 \cup \mathbf{\{8,10\}}$.
Set $\mathbf{Z_2} = Z_1 \cup \{ X \in N \mid X \rightarrow w \in Q_2 \} = Z_1 \cup \mathbf{\{A\}}$.
Set $\mathbf{Q_3} = Q_2 \cup \{ X \rightarrow w \in P \mid w \in (\Sigma \cup Z_2)^* \} = Q_2 \cup \mathbf{\{2\}}$.
Set $\mathbf{Z_3} = Z_2 \cup \{ X \in N \mid X \rightarrow w \in Q_3 \} = Z_2 \cup \mathbf{\{S\}}$.
Step 3a: $Q_4 = Q_3$, so exit
Step 4a: $G_1 = (N_1 = \{S,A,B,C,Z\}, \{a,b\}, P_1 = \{2,6,7,8,9,10,11\})$, since $S \in N_1$ we must apply Algorithm F-1b.

Step 1b: Set $\mathbf{Z_0} = \mathbf{\{S\}}$. Set $\mathbf{Q_0} = \{ S \rightarrow w \in P_1 \} = \mathbf{\{2\}}$. Set k = 0.
Step 2b: Set $\mathbf{Z_1} = Z_0 \cup \{X \in N_1 \mid Y \rightarrow uXv \in Q_0$, for some strings u and v$\} = Z_0 \cup \mathbf{\{A, B\}}$.
Set $\mathbf{Q_1} = Q_0 \cup \{ X \rightarrow w \in P_1 \mid X \in Z_1 \} = Q_0 \cup \mathbf{\{8, 10, 11\}}$.
Set $\mathbf{Z_2} = Z_1 \cup \{X \in N_1 \mid Y \rightarrow uXv \in Q_1$, for some strings u and v$\} = Z_1 \cup \mathbf{\{C\}}$.
Set $\mathbf{Q_2} = Q_1 \cup \{ X \rightarrow w \in P_1 \mid X \in Z_2 \} = Q_1 \cup \mathbf{\{9\}}$.

Step 3b: $Z_3 = Z_2$, so halt!
Step 4b: G' (reduced form of G) **= (N'= {S,A,B,C}, {a,b}, P' = {2, 8, 9, 10, 11}).**

Definition 32. Let G = (N,Σ, P, S) be a CFG. $X \in N$ is said to be ***nullable*** if and only if $X \Rightarrow^{+} \lambda$. Furthermore, ***Nullable(G)*** $= \{ X \in N \mid X \text{ is a nullable} \}$.

ALGORITHM F-2: Compute the Nullable(G), where G = (N,Σ, P, S) is a CFG.

Step 1: Define $Z_0 = \{ X \in N \mid X \rightarrow \lambda \in P \}$. Set k = 0.
Step 2: Define $Z_{k+1} = Z_k \cup \{ X \in N \mid X \rightarrow \alpha \in P, \text{ where } \alpha \in (Z_k)^* \}$.
Step 3: If $Z_{k+1} \neq Z_k$ then { k = k+1. goto step 2. }
Halt: Nullable(G) = Z_k.

Example 37. Consider the reduced grammar G' obtained in Example 36. Compute Nullable(G').

G' = (N, Σ, P, S), where N = {S, A, B, C}, Σ = {a, b}, and
P = { 1: S → AB
2: A → Cb
3: C →B
4: B → BCa
5: B → λ
}
Step 2.1: Define $\mathbf{Z_0} = \{ X \in N \mid X \rightarrow \lambda \in P \}$ **= {B}.** Set k = 0.
Step 2.2: Define $\mathbf{Z_1} = Z_0 \cup \{ X \in N \mid X \rightarrow \alpha \in P, \text{ where } \alpha \in (Z_0)^* \} = Z_0 \cup$ **{C}**.
Step 2.3: $Z_2 = Z_1$, so halt. Nullable(G') = {B, C}

ALGORITHM F-3: Removing the λ-rules (erasing rules) from G = (N,Σ, P, S), a CFG.

Step 1: Compute Nullable(G).
Step 2: Define G' = (N', ,Σ, P', S), where P' = { $X \to \beta \mid \beta \neq \lambda$ and $\beta \in \sigma(\alpha)$ where $X \to \alpha \in P$ and σ is the substitution defined by: $\sigma(a) = \{a\}$ for each $a \in \Sigma$;
$\sigma(Y) = \{Y\}$ for each $Y \in N - Nullable(G)$; $\sigma(Y) = \{Y,\lambda\}$ for each $Y \in Nullable(G)$ }

Rationale: To eliminate λ-rules, we must add new rules that compensate for the effect of $X \Rightarrow^{+} \lambda$ occurring in some derivation. For example, suppose $S \overset{*}{\Rightarrow} \alpha X \beta \overset{r}{\Rightarrow} \alpha x Y z \beta \overset{*}{\Rightarrow} \alpha x z \beta$, where r: $X \to xYz \in P$ and $Y \overset{*}{\Rightarrow} \lambda$. Then the effect of the subderivation, rθ, can be accomplished by the single rule : $X \to xz$. This is exactly what the substitution σ is doing - it is replacing nullable nonterminals by λ in all combinations, if more than one appears in the same rightpart, to generate a set of rules that simulate the effect of a given rule followed by a derivation that erases one or more of the nullable nonterminals introduced by the given rule into a sentential form. The only effect that cannot be duplicated by adding rules, is the derivation $S \overset{*}{\Rightarrow} \lambda$. Thus, the new grammar may not generate λ as a member of the language.

Step 3: Reduce G' to obtain the desired grammar.

NOTE: L(G') = L(G), if $S \notin Nullable(G)$. Otherwise, L(G') = L(G) - {λ }.

Example 38. Consider the reduced grammar G' obtained in Example 37. Eliminate rule 5 to obtain a grammar without λ-rules equivalent to L(G') - {λ}.

G' = (N, Σ, P, S), where N = {S, A, B, C}, Σ = {a, b}, and
P = { 1: $S \to AB$
2: $A \to Cb$
3: $C \to B$
4: $B \to BCa$
5: $B \to \lambda$
}

Step 1: Compute Nullable(G'). This was done in Example 37. Nullable(G) = {B, C}.

Step 2: Construct P' = {
1: $S \to AB$ ⇒ $S \to AB$, $S \to A$ (A is not nullable)
2: $A \to Cb$ ⇒ $A \to Cb$, $A \to b$
3: $C \to B$ ⇒ $C \to B$ (eliminating B would produce a new λ-rule)
4: $B \to BCa$ ⇒ $B \to BCa$, $B \to Ca$, $B \to Ba$, $B \to a$
}

Step 3: Reduce P' to obtain G" = (N = {S, A, B, C}, {a,b}, P", S), where
P" = { 1: S → AB,
2: S → A,
3: A → Cb,
4: A → b,
5: C →B,
6: B → BCa,
7: B → Ca,
8: B → Ba,
9: B → a
} Observe that P" = P' (P' was already in reduced form)

NOTE: Since S was not nullable in G', then $\lambda \notin L(G')$. Therefore L(G") = L(G').

ALGORITHM F-4: Removing the unit rules (X →Y) from G = (N,Σ, P, S), a reduced CFG having no λ-rules.

Step 1: For each $X \in N$, compute *Unit(X)* = { $Y \in N \mid X \Rightarrow^* Y$ in G }.
NOTE: $X \in$ Unit(X).
Step 2: Define P_1 as follows.
P_1 = { $X \rightarrow \beta \mid \beta \in \sigma(\alpha)$ where $X \rightarrow \alpha \in P$ with $\alpha \notin N$, and σ is the substitution defined by: $\sigma(a) = \{a\}$ for each $a \in \Sigma$; $\sigma(Y) = U(Y)$ for each $Y \in N$ }
P_1 contains no unit rules and includes one or more copies of the non-unit rules in G where occurrences of nonterminals in the rightpart are replaced by members of their Unit-set.

Rationale: An approach similar to that used in Algorithm 3 for eliminating λ-rules works well for eliminating unit rules. Specifically, we are trying to simulate a derivation of the form, $S \overset{*}{\Rightarrow} \alpha X \beta \overset{r}{\Rightarrow} \alpha x Y z \beta \overset{\theta}{\Rightarrow} \alpha x Z z \beta$, where r: $X \rightarrow xYz \in P$ and $Y \overset{\theta}{\Rightarrow} Z$ where θ consists only of the application of unit rules. Clearly the single rule, r': $X \rightarrow xZz$ achieves the same effect as the derivation rθ, where Z is any nonterminal in Unit(Y).

For example, if $A \rightarrow aXbY \in P$ and Unit(X) = {X, B} and Unit(Y) = {Y, C} then the following productions would be added to P_1: {A → aXbY, A → aBbY, A → aXbC, A → aBbC}

Step 3: Define G' = (N', Σ, P', S) to be the reduced form of G_2 = (N, S, P_2, S) where
$P_2 = P_1 \cup \{ S \rightarrow \beta \mid X \rightarrow \beta \in P_1$ and $X \in U(S) \}$.
NOTE: L(G') = L(G) and G' has no unit rules.

Example 39. Eliminate the unit rules from G", the grammar produced in Example 38.

G" = (N = {S, A, B, C}, {a,b}, P", S), where

P" = { 1: S → AB,
2: S → A,
3: A → Cb,
4: A → b,
5: C →B,
6: B → BCa,
7: B → Ca,
8: B → Ba,
9: B → a}

Step 1: Compute Unit(X) for each nonterminal X.

Unit(S) = {S, A}
Unit(A) = {A}
Unit(B) = {B}
Unit(C) = {C, B}

Step 2: Compute $P"_1$ = { rules obtained by substituting Unit(X) in the non-unit rules of P" }

= {
S → AB ⇒ S → AB
A → Cb ⇒ A → Cb, A → Bb
A → b ⇒ A → b
B → BCa ⇒ B → BCa, B → BBa
B → Ca ⇒ B → Ca, B → Ba
B → Ba ⇒ B → Ba (redundant)
B → a ⇒ B → a }

Step 3: Construct $P"_2 = P"_1 \cup \{ S \rightarrow \beta \mid X \rightarrow \beta \in P"_1 \text{ and } X \in U(S) - \{S\} \} =$
$P"_1 \cup \{ \{ S \rightarrow \beta \mid A \rightarrow \beta \in P"_1 \} =$ **$P"_1 \cup$ { S → Cb, S → Bb, S → b }**

Step 4: Reduce $P"_2$ to obtain the final grammar. The reduced subset of $P"_2$ is the following:
{ **S → AB**, **S → Bb**, **S → b, A → Bb, A → b, B → BBa, B → Ba, B → a** }

Definition 33. Let G = (N, Σ, P, S) be a CFG. G is said to be in *Chomsky Normal Form* (CNF) if and only if $X \rightarrow \alpha \in P$ implies $\alpha \in \Sigma \cup NN$.

Example 40. The following is a CNF grammar for $L = \{ a^n b^n \mid n \geq 1 \}$. G = (N, Σ, P, S), where
N = {S, X, A, B }, Σ = {a, b}, and P =
{ 1: S → XA
2: X → BS
3: S → AB
4: A → a
5: B → b
}

Theorem 16. Every CFL without λ is defined by a reduced grammar in CNF.

Proof. Let $L = L(G) \subseteq \Sigma^*$ and assume that $\lambda \notin L$. Applying Algorithms F-3 (remove λ-rules), F-4 (remove unit rules) and F-1 (reduce) to G we can obtain a reduced and equivalent grammar G_1 having no erasing or unit rules. G_1 is equivalent to G because L(G) was assumed not to contain λ. Consequently, G_1 must contain only rules having one of the following forms:
[1] $X \rightarrow a$, where $a \in \Sigma$, or
[2] $X \rightarrow Y_1Y_2\ldots Y_n$, where $n \geq 2$ and $Y_k \in V_{G1}$.

Observe that rules of type [1] already satisfy CNF. It is rules of type [2] that we are concerned with. The first equivalence preserving transformation is to replace each terminal $Y_k = a \in \Sigma$ by a new nonterminal, Z_a in all rules of type [2]. Next we define a new set of rules $P_\Sigma = \{ Z_a \rightarrow a \mid a \in \Sigma \}$ of type [1] that rewrite the new nonterminals.

Let $G_2 = (N_2 , \Sigma, P_2, S)$ be the grammar obtained as a result of this transformation; $N_2 = N_1 \cup \{Z_a \mid a \in \Sigma \}$ and $P_2 = P_1 \cup P_\Sigma$. Observe that rules in P_2 are of type [1] or of type [3], where

[3] $X \rightarrow Y_1Y_2\ldots Y_n$, where $n \geq 2$ and $Y_k \in N_2$.

The only difference between the form of G_2 and CNF is that rules of type [3] may have rightparts longer than two! So, the final transformation is to replace each type [3] rule for which $n \geq 3$ by the following set of equivalent rules :

$\{ X \rightarrow Y_1A_1,$
$A_1 \rightarrow Y_2A_2 ,$
…
$A_{n-2} \rightarrow Y_{n-1}Y_n \}$

As part of this transformation we introduce new nonterminals $A_1, A_2, \ldots A_{n-2}$ unique to each replaced rule. This is necessary to ensure that any derivation that applies, $X \rightarrow Y_1A_1$, cannot terminate until $A_{n-2} \rightarrow Y_{n-1}Y_n$ has been applied. In short, the single rule $X \rightarrow Y_1Y_2\ldots Y_n$ in G_2 is simulated by the derivation $X \Rightarrow Y_1A_1 \Rightarrow Y_1Y_2A_2 \Rightarrow \ldots \Rightarrow$
$Y_1Y_2\ldots Y_{n-2}A_{n-2} \Rightarrow Y_1Y_2\ldots Y_n$ in the new grammar, call it G_3.

It should be clear that G_3 is in CNF and equivalent to G_1. It should also be apparent that G_3 will be a reduced grammar.

The significance of Chomsky Normal Form is derived from the following theorem due to Coche, Younger and Kasami.

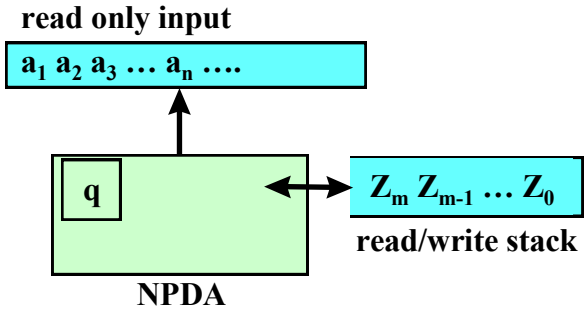
Theorem 17. Let $L \subseteq \Sigma^*$ be any Context-free language not containing the null string, λ. Then there exists an algorithm for answering the question, *For any $x \in \Sigma^*$ [Is $x \in L$]?,* in $O(|x|^3)$ steps.

Proof. The algorithm depends on the fact that L = L(G) for some grammar G in Chomsky Normal Form. The details are available upon request.

Pushdown Automata

We now answer the question: What kind of machine can accept Context-free languages? We show in this section that if an NFA is augmented with additional memory in the form of a stack, then it can recognize any Context-free language. We will also show that the converse of this statement is also true.

Definition 34. A *Non-deterministic Pushdown Automaton* (NPDA) is a 7-tuple,
$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, A)$, where
Q = finite set of states,
Σ = input alphabet,
Γ = stack alphabet,
$q_0 \in Q$ = the initial state
$Z_0 \in \Gamma$ = bottom of stack marker (or initial stack symbol),
δ: $Q \times (\Sigma \cup \{\lambda\}) \times \Gamma \rightarrow Q \times \Gamma^*$ = the transition relation. In any given state, M optionally reads the next input symbol, pops and examines the top stack symbol, pushes a string γ (perhaps null) back on the stack, and non-deterministically chooses its next state. The diagram below illustrates this behavior.



There are several sources of nondeterminism in this model. First, the NPDA can choose to ignore the input tape and make a transition based only on information provided by the top stack symbol. Second, it can choose one of several possible next states based on whatever information it has read (input and/or stack). And, third, it can push one of several possible strings before entering the next state.

A PDA must halt if its stack ever becomes empty after a transition to a new state. It can accept its input only if it has completely read the input (or it the input tape is initially λ), AND one of the following criteria is used.
(a) Its stack is empty, OR
(b) It has entered an accept state.

Thus there are two classes of NPDA depending on how acceptance is defined. One class accepts by empty stack, the other accepts by final state. It should be understood that an NPDA cannot accept one string by empty stack and another by final state. One criterion must be for all strings in Σ^*. When acceptance by empty stack is intended, A is chosen to be the empty set.

Like NFAs, computations of a PDA are defined in terms of *configurations and the move relation* $\Rightarrow_M$. Configurations of a PDA, however, have a third component - the stack contents. Formally we define,

For $Z \in \Gamma$, $(q, x, Z\theta) \Rightarrow_M (q', x, \gamma\theta)$ if and only if $(q',\gamma) \in \delta(q, \lambda, Z)$
For $a \in \Sigma$ and $Z \in \Gamma$, $(q, ax, Z\theta) \Rightarrow_M (q', x, \gamma\theta)$ if and only if $(q',\gamma) \in \delta(q,a,Z)$

Now we can give formal definitions for the language accepted by empty stack and final state.

Definition 35. Let $M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, A)$ denote a PDA that accepts by final state. Then the language accepted by M is

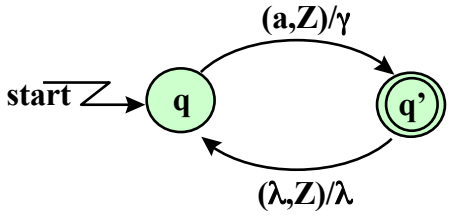
$L_{ste}(M) = \{x \in \Sigma^* \mid (q_0, x, Z_0) \Rightarrow_M^* (f, \lambda, \theta)$, for some $f \in A$ and $\theta \in \Gamma^*\}$.

Let M' = $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, \Phi)$ denote a PDA that accepts by empty stack. Then the language accepted by M' is

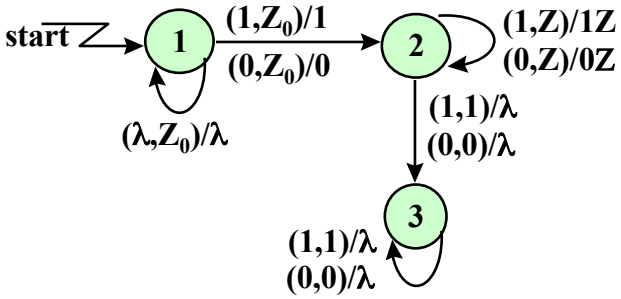
$L_{stk}(M') = \{x \in \Sigma^* \mid (q_0, x, Z_0) \Rightarrow_M^* (q', \lambda, \lambda)$, for some $q' \in Q\}$.

It turns out that ***the family of languages a PDA can accept by empty stack is exactly the same family that can accepted by final state***. This result is straightforward to prove and will be presented after we have considered a few examples.

A State Transition Diagram (STD) can be augmented to describe the behavior of a PDA as illustrated below. The transition from q to q' depicts a transition in which "a" is read from the input and "Z" is read from the stack, with the string γ replacing Z in the process. The transition from q' to q depicts a transition in which the input stream is ignored and the top stack symbol Z is replaced by the null string - this is equivalent to simply popping Z from the stack.



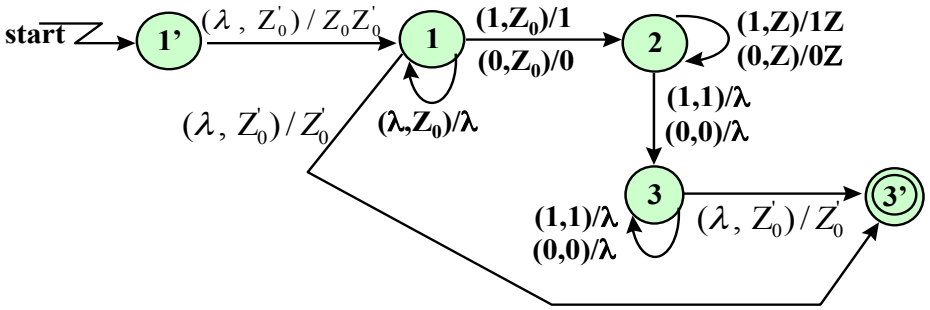
Example 41. Construct a PDA that accepts $L = \{ ww^{rev} \mid w \in \{0,1\}^*\}$ by empty stack. M begins by making a nondeterministic decision about whether the input is null or not. If it decides the input is λ, then it pops the stack marker Z_0 and accepts (the stack will be empty). If it decides the input is non-λ, then it reads the next symbol and pushes it on the stack to replace Z_0. If M guesses incorrectly that the input is null, then M halts not having read all of the input - acceptance is inconclusive in this case. If M guesses incorrectly that the input is not null - that is, by trying to read - then we must use the following analysis: if there is some other transition M could make (a Λ-move), then choose one nondeterministically and continue, otherwise halt with a non-empty stack.



Once M reaches state 2, it continues to read and push the input symbol onto the stack. At a point halfway through the input (M must "guess" when that occurs), M begins matching the input with the top symbol of the stack and then popping the stack. In the fortuitous circumstance when the input belongs to L and M makes all guesses correctly, the stack will be empty after having read the last symbol of the input - and M accepts. If the stack empties before all input has been read, then acceptance is undefined, or inconclusive. If the input is exhausted before the stack empties, then the input is not accepted. If a mismatch occurs in state 3, then M halts without an empty stack - and acceptance is either undefined, or the input is rejected outright.

Observe how important non-determinism is to the behavior of this PDA. Like the NFA, the PDA can make "wrong" guesses without penalty. An input string belongs to the language accepted by the PDA as long as there exists at least one accepting computation (sequence of moves)! What we cannot permit is a computation that rejects an input, when it belongs to L.

Example 42. To illustrate, informally, the construction used to establish that any PDA accepting by empty stack is equivalent to one that accepts by final state, we redo the previous example by constructing a PDA M' that accepts L = { ww^{rev} | w ∈{0,1}*}by final state. This is done by simulating M of the previous example as follows. M' will use a new bottom of stack marker, Z_0', that is different from that used by M, one M cannot recognize. M' starts the simulation by ignoring the input and pushing M's stack marker (Z_0) on the stack. It then gives control to M in its initial state. If the simulation of M ever reaches a configuration where Z_0' is the top stack symbol, then this must mean M has emptied its own stack and M' regains control by entering its one and only accepting state and halts. If input remains, then M emptied its stack prematurely and M' will neither accept nor reject - acceptance is inconclusive. On the other hand, if the input is empty, then M would have accepted and so M' will halt in an accepting configuration as well. If M halts for any other reason, then M' also halts (in a state of M - a non-accepting configuration for M').



PDA Accepting by Final State

Theorem 18. Let M = (Q, Σ, Γ, δ, q_0, Z_0, Φ) be a PDA that accepts by empty stack. Then there exists a PDA, M' = (Q', Σ, Γ', δ', q'_0, Z'_0, A') , such that $L_{ste}(M') = L_{stk}(M)$. The converse is also true.

Proof. Define Q' = Q ∪ {q'_0, Ω }, Γ' = Γ ∪ {Z'_0}, A' = {Ω}, and δ' = δ ∪ T', where T' are the following transitions: δ'(q'_0,Λ,Z'_0) = {(q_0, $Z_0Z'_0$)}, and for all q ∈ Q, δ'(q, Λ,Z'_0) = {(Ω, Z'_0)}. As we outlined in Example 42, M' simulates M until M empties its stack in some state, q. At this point Z'_0 is on top of the stack and only the new transition δ'(q, Λ,Z'_0) = (Ω, Z'_0) is defined. M' thus enters its accept state, Ω. If the input is exhausted when M emptied its stack, then M would have accepted. In this case M' can also accept. If the input was not consumed when M' enters Ω, then neither M nor M' will accept – acceptance is inconclusive for both PDAs. To state this more formally, suppose x ∈ $L_{stk}(M)$. Then (q_0, x, Z_0) $_M\Rightarrow^*$ (q, λ, λ) for some state q∈Q. Now in M' we have : (q'_0, x, Z'_0) $_{M'}\Rightarrow$ (q_0, x, $Z_0Z'_0$) $_{M'}\Rightarrow^*$ (q, λ, Z'_0) $_{M'}\Rightarrow$(Ω, λ, Z'_0). The first and last moves of M' are transitions in T'. Since δ' = δ ∪ T', the computation (q_0, x, Z_0) $_M\Rightarrow^*$ (q, λ, λ) can be duplicated in M' resulting in the configuration (q, λ, Z'_0) instead of
(q, λ, λ). Thus $L_{stk}(M) \subseteq L_{ste}(M')$. Since the argument is reversible, the reverse inclusion holds establishing the desired result.

To prove the converse, we adopt essentially the same approach. Let M = (Q, Σ, Γ, δ, q_0, Z_0, A) be a PDA that accepts by final state (we may assume A ≠Φ). We want to construct M' such that $L_{stk}(M') = L_{ste}(M)$. Once again let M' = (Q', Σ, Γ', δ', q'_0, Z'_0, Φ), where Q' = Q ∪ {q'_0, Ω}, Γ' = Γ ∪ {Z'_0}, and δ' = δ ∪ T'. Note that Ω is not an accept state. The additional transitions (T') are designed to enable M' to simulate M and empty its stack in any configuration of M that reaches an accept state. We also need to prevent M' from accepting accidentally during the simulation of M, should M happen to empty its stack in a non-accept state after consuming its input. This situation is prevented in our design by having M' push a new initial stack symbol on the stack before starting the simulation of M. Thus we have: δ'(q'_0, Λ,Z'_0) = {(q_0, $Z_0Z'_0$)} as before, and for every f ∈ A, and Z' ∈ Γ' define δ'(f, Λ,Z') = {(Ω,λ)}, finally, δ'(Ω, Λ,Z') = {(Ω,λ)} for every Z' ∈ Γ'. Every computation of M' begins by pushing Z_0, the initial stack symbol of M, onto the stack and entering the initial state of M to begin the simulation. Now, should M empty its stack on some given input, then the simulation of M by M' will result in Z'_0 appearing on top of the stack instead. If the final state of M is non-accepting in this case, then no transition to Ω will be defined in M' and M' will also halt, but with a non-empty stack. On the other hand, if M reaches an accept state when its stack becomes empty, then M' can make a transition to Ω emptying its stack. In this case both M and M' will accept if and only if

the input has been consumed. If M consumes its input and ends in an accepting configuration with a non-empty stack, then in the simulation of M, M' can make a transition to Ω and empty its stack while ignoring the input. Thus, M' can accept whenever M can accept. If M' prematurely empties its stack (when the input is not consumed), then M could not have accepted either. It follows that both PDAs are able to accept exactly when the other would accept. This concludes the proof.

Equivalence of PDAs to CFGs

We now address the question of equivalence between PDAs and CFGs.

The first key idea to understanding this equivalence is to view a PDA as a non-deterministic parser for a given Context-free language. Recall that a bottom-up parser attempts to reverse a rightmost derivation in the defining CFG when presented with some string over the terminal alphabet. On the other hand, a top-down parser attempts to construct a leftmost derivation in the defining CFG when presented its input. As we shall see, a PDA that accepts by empty stack provides the perfect model of a non-deterministic top-down parser for a given CFG. To understand how a parsing PDA must work, we consider a leftmost derivation in the familiar expression grammar.

Example 43. Consider the grammar for arithmetic expressions we introduced earlier. It is reproduced below for convenience. G = ({E, T, F}, {n,v, +, *, (,)}, P, E), where

E = { 1: E → E + T,
2: E → T,
3: T → T∗ F,
4: T → F,
5: F → n,
6: F → v,
7: F → (E),
}

Suppose the input to our parser is the expression, n*(v+n*v). Since G is unambiguous this expression has only one leftmost derivation, π = 2345712463456. We describe the behavior of the PDA in general, and then step through its moves using this derivation to guide the computation.

PDA Behavior.

Step 1: Initialize the stack with the start symbol (E in this case). The start symbol will serve as the bottom of stack marker.

Step 2: Ignoring the input, check the top symbol of the stack.
Case (a) The top of stack is a nonterminal, "X"**:** non-deterministically decide which X-rule to use as the next step of the derivation. After selecting a rule, replace X in the stack with the rightpart of that rule. If the stack is non-empty, repeat step 2. Otherwise, halt (input may or may not be empty.)

Case(b) Top of stack is a terminal, "a"**:** read the next input. If the input matches "a", then pop the stack and repeat step 2. Otherwise, halt (without popping "a" from the stack.)

We illustrate this parsing algorithm by showing the sequence of configurations the parser would assume in an accepting computation for the input, n*(v+n*v). Assume "q_0" is the one and only state of this PDA. π = 2345712463456.

$(q_0, n*(v+n*v), E)$
$\overset{2}{\Rightarrow}_M (q_0, n*(v+n*v), T)$
$\overset{3}{\Rightarrow}_M (q_0, n*(v+n*v), T*F)$
$\overset{4}{\Rightarrow}_M (q_0, n*(v+n*v), F*F)$
$\overset{5}{\Rightarrow}_M (q_0, n*(v+n*v), n*F)$ $\overset{read}{\Rightarrow}_M (q_0, *(v+n*v), *F)$ $\overset{read}{\Rightarrow}_M (q_0, (v+n*v), F)$
$\overset{7}{\Rightarrow}_M (q_0, (v+n*v), (E))$ $\overset{read}{\Rightarrow}_M (q_0, v+n*v), E))$
$\overset{1}{\Rightarrow}_M (q_0, v+n*v), E+T))$
$\overset{2}{\Rightarrow}_M (q_0, v+n*v), T+T))$
$\overset{4}{\Rightarrow}_M (q_0, v+n*v), F+T))$
$\overset{6}{\Rightarrow}_M (q_0, v+n*v), v+T))$ $\overset{read}{\Rightarrow}_M (q_0, +n*v), +T))$ $\overset{read}{\Rightarrow}_M (q_0, n*v), T))$
$\overset{3}{\Rightarrow}_M (q_0, n*v), T*F))$
$\overset{4}{\Rightarrow}_M (q_0, n*v), F*F))$
$\overset{5}{\Rightarrow}_M (q_0, n*v), n*F))$ $\overset{read}{\Rightarrow}_M (q_0, *v), *F))$ $\overset{read}{\Rightarrow}_M (q_0, v), F))$
$\overset{6}{\Rightarrow}_M (q_0, v), v))$ $\overset{read}{\Rightarrow}_M (q_0,),))$ $\overset{read}{\Rightarrow}_M (q_0, \lambda, \lambda)$ **accept!**

The above example can be generalized for any Context-free grammar and so we state the following,

Theorem 19. Let G = (N, Σ, P, S) denote any reduced Context-free grammar. Then there is a PDA, M = ({q_0}, Σ, V_G, δ, q_0, Φ), such that $L_{stk}(M) = L(G)$. The converse is also true. If M is any PDA accepting by empty stack, then there is a Context-free grammar, G, such that
$L(G) = L_{stk}(M)$.

Proof. We have only to define the transition relation d in terms of the productions of G. For every a ∈Σ, define δ(q_0, a, a) = { (q_0, λ) }; if a terminal in on top of the stack that matches the next input, the stack is popped. For every X ∈ N, and X → β ∈ P, add (q_0, β) to δ(q_0,Λ, X). What we wish to show is that $(q_0, x, S) \overset{+}{\Rightarrow}_M (q0, \lambda, \lambda)$ if and only if $S \overset{*}{\Rightarrow}_G x$, where π is a leftmost derivation in G. To this end we prove a slightly stronger statement:

IH(n): For any Z ∈ N, and x,y ∈ Σ*, $(q_0, xy , Z) \overset{n}{\Rightarrow}_M (q_0, y, \theta)$ if and only if $Z \overset{n}{\Rightarrow}_G x\theta$, where μ is a sequence of moves of M and π is a leftmost derivation in G related in the following fashion.
(a) $\pi \in P^+$ and $|\pi| = n > 0$.
(b) $\mu \in (\Sigma \cup P)^+$ and $|\mu| = n + |x|$; μ should be interpreted in the following way.
If $(q_0, az, a\gamma) \Rightarrow_M (q_0, z, \gamma)$, then we identify this as an "a-move".
If $(q_0, z, X\gamma) \Rightarrow_M (q_0, z, \beta\gamma)$, where r = X→β, then this is identified as an "r-move".
(c) h(μ) = π, where h is the homomorphism h(a) = λ, for each a ∈Σ, and h(r) = r, for each r ∈P.
(d) h'(μ) = x, where h' is the homomorphism h'(r) = λ, for all r ∈P, and h'(a) = a, for all a ∈Σ.

This can be established by induction on n in a straightforward manner by applying the definition of M and leftmost derivations in G. Details are left to the interested reader.
From IH it then follows that $(q_0, x, S) \overset{n}{\Rightarrow}_M (q_0, \lambda, \lambda)$ if and only if $S \overset{n}{\Rightarrow}_G x$. This establishes that $L(G) = L_{stk}(M)$.

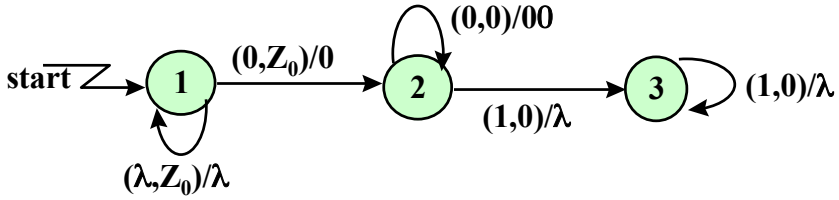
We now consider the converse. Let M be PDA that accepts by empty stack. To construct a grammar from M it is necessary to define nonterminals to be *triples* of the form [qZq'], where q and q' denote states of M and Z denotes a possible stack symbol. The grammar rules are defined as follows:

(a) Let S be the start symbol of G, where S is distinct from any symbol in Γ (we assume Σ ⊆ Γ). Then define the following productions for S, { S → [q_0Z_0q'] | q_0 is the initial state of M, Z_0 is the initial stack symbol of M, and q' ∈ Q_M is arbitrary}.
(b) For (q', λ) ∈ δ(q, σ ,Z), where σ ∈ (Σ ∪{Λ}), add the production, [qZq'] → σ, to G.

(c) For $(q', Y_1Y_2\ldots Y_n) \in \delta(q, \sigma, Z)$, where $n \geq 1$ and $\sigma \in (\Sigma \cup \{\Lambda\})$, add the set
$\{ [qZp_n] \rightarrow \sigma [q'Y_1p_1][p_1Y_2p_2] \ldots [p_{n-1}Y_np_n] \mid p_1, p_2, \ldots, p_n \in Q \}$ to G.

It can be shown that $[qZq'] \underset{G}{\Rightarrow}^* x \in \Sigma^*$, where π is a leftmost derivation, if and only if $(q, x, Z) \underset{M}{\Rightarrow}^* (q', \lambda, \lambda)$. A formal proof of this fact would proceed by induction on the length of $|\pi| = |\mu|$. From this it straightforward to argue that $L(G) = L_{stk}(M)$.

Example 44. We illustrate the construction of a CFG from a PDA with a PDA, M, for which $L_{stk}(M) = \{0^n1^n \mid n \geq 0\}$.



The rules of G produced for this PDA are:
(a) $\{ S \rightarrow [1\mathbf{Z_0}1], S \rightarrow [1\mathbf{Z_0}2], S \rightarrow [1\mathbf{Z_0}3] \} \cup$
(b) $\{ [1\ \mathbf{Z_0}1] \rightarrow \lambda, [2\mathbf{0}3] \rightarrow 1, [3\mathbf{0}3] \rightarrow 1\} \cup$
(c) $\{ [1\mathbf{Z_0}1] \rightarrow 0[2\mathbf{0}1], [1\mathbf{Z_0}2] \rightarrow 0[2\mathbf{0}2], [1\mathbf{Z_0}3] \rightarrow 0[2\mathbf{0}3] \} \cup$
$\{ [2\mathbf{0}1] \rightarrow 0[2\mathbf{0}1][1\mathbf{0}1], [2\mathbf{0}1] \rightarrow 0[2\mathbf{0}2][2\mathbf{0}1], [2\mathbf{0}1] \rightarrow 0[2\mathbf{0}3][3\mathbf{0}1] \} \cup$
$\{ [2\mathbf{0}2] \rightarrow 0[2\mathbf{0}1][1\mathbf{0}2], [2\mathbf{0}2] \rightarrow 0[2\mathbf{0}2][2\mathbf{0}2], [2\mathbf{0}2] \rightarrow 0[2\mathbf{0}3][3\mathbf{0}2] \} \cup$
$\{ [2\mathbf{0}3] \rightarrow 0[2\mathbf{0}1][1\mathbf{0}3], [2\mathbf{0}3] \rightarrow 0[2\mathbf{0}2][2\mathbf{0}3], [2\mathbf{0}3] \rightarrow 0[2\mathbf{0}3][3\mathbf{0}3] \}$

This construction always creates useless productions. So, as a good exercise of your understanding of the reduction algorithm for CFGs, apply it to G to obtained its reduced form.
Compare your result to the one given below. Then test this grammar to convince yourself that it does indeed generate the language $L_{stk}(M) = \{0^n1^n \mid n \geq 0\}$.

Reduced form of G:
$\{ 1: S \rightarrow [1\mathbf{Z_0}1], 2: S \rightarrow [1\mathbf{Z_0}3] \} \cup \{ 3: [1\ \mathbf{Z_0}1] \rightarrow \lambda, 4: [2\mathbf{0}3] \rightarrow 1, 5: [3\mathbf{0}3] \rightarrow 1\} \cup$
$\{ 6: [1\mathbf{Z_0}3] \rightarrow 0[2\mathbf{0}3], 7: [2\mathbf{0}3] \rightarrow 0[2\mathbf{0}3][3\mathbf{0}3] \}$

Observe that $S \overset{13}{\Rightarrow} \lambda$, and $S \overset{2677455}{\Rightarrow} 000111$. You can generalize from these examples.

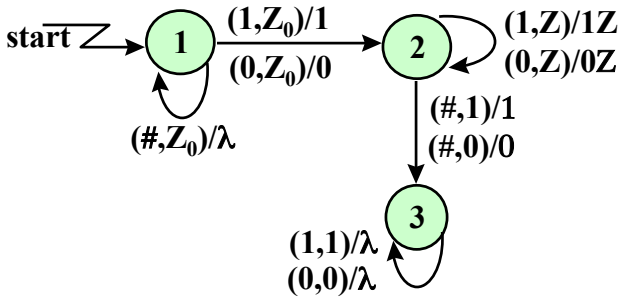
Deterministic PDAs

Now that we have characterized the family of Context-free languages in terms of nondeterministic PDAs, we consider the power of deterministic PDAs. After all, in any practical application, such as designing a compiler, we intend to use only deterministic algorithms. Since a nondeterministic parser will not do, we are natually led to wonder what family of languages can be recognized by DPDAs (Deterministic PDA), and equally important, what class of Context-free grammars corresponds to this new family.

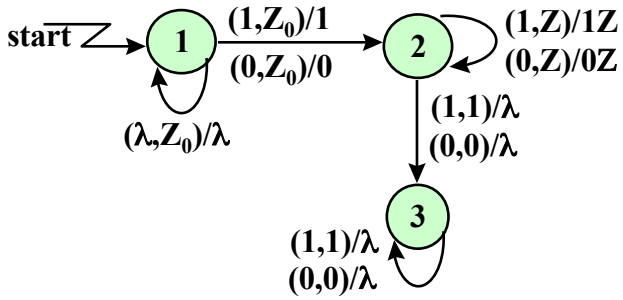
Definition 36. A *Deterministic Pushdown Automaton* (DPDA) is a 7-tuple,
$M = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, A)$, where Q = finite set of states, Σ = input alphabet, Γ = stack alphabet,
$q_0 \in Q$ = the initial state, $Z_0 \in \Gamma$ = bottom of stack marker (or initial stack symbol), and
$\delta: Q \times (\Sigma \cup \{\Lambda\}) \times \Gamma \rightarrow Q \times \Gamma^*$ = the transition function (not necessarily total). Specifically,
if $\delta(q, a, Z)$ is defined for some $a \in \Sigma$ and $Z \in \Gamma$, then $\delta(q, \Lambda, Z) = \Phi$ and $|\delta(q, a, Z)| = 1$. Conversely, if $\delta(q, \Lambda, Z) \neq \Phi$, for some Z, then $\delta(q, a, Z) = \Phi$, for all $a \in \Sigma$, and $|\delta(q, \Lambda, Z)| = 1$.

What this means is that for a given pair in $Q \times \Gamma$, M may either read from its input and change the stack, or ignore the input and change the stack - it can do one or the other, perhaps neither, but it cannot do both. In either case, whatever action is taken, the outcome is unique - that is, the next state and string pushed on the stack to replace Z will always be unique whenever δ is defined. We illustrate the power of a DPDA with some examples.

Example 45. $L_{stk}(M) = \{w\#w^{rev} \mid w \in \{0,1\}^* \}$ The # is needed to so that the DPDA can detect the middle of the input, that is, the point at which it can begin comparing its stack contents symbol-by-symbol with the remaining input.



Compare this with the PDA, M', below which accepts: $L_{stk}(M') = \{ww^{rev} \mid w \in \{0,1\}^* \}$. Note that in state 1, when Z_0 is the stack top, M' can make a transition that reads from the input and can make a transition that ignores the input. Also, in state 2, when a 0 (or 1) is the stack top, M' has two possible transitions - both of which read from the input - M' can push the symbol read, or if it matches the stack symbol, can pop the stack. Both of these behaviors are prohibited for a DPDA. As it turns out, one can prove that $L = \{ww^{rev} \mid w \in \{0,1\}^* \}$ cannot be recognized by any DPDA !

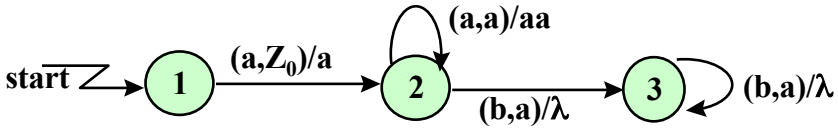


Non-deterministic PDA for $L = \{ww^{rev} \mid w \in \{0,1\}^* \}$

Corollary 36-1. The family of DCFLs (Deterministic Context-free Languages) is properly included in the CFLs. In particular, $L = \{ww^{rev} \mid w \in \{0,1\}^* \}$ cannot be recognized by any DPDA !

Corollary 36-2. $L = \{a^n b^n \mid n > 0 \} = L_{stk}(M)$, for some DPDA, M, but $L' = \{(ab)^n \mid n > 0 \} \neq L_{stk}(M)$, for any DPDA, M. On the other hand, there is a DPDA, M', such that $L' = L_{ste}(M')$.

Proof. The DPDA M shown below accepts L by empty stack.



In general, if L is a language that is not prefix-free, then L cannot be accepted by a DPDA by empty stack. L is prefix-free if for every pair of distinct strings x, y ∈ L neither is a prefix of the other. (Note: if λ ∈ L and L is prefix-free, then L = {λ}!)

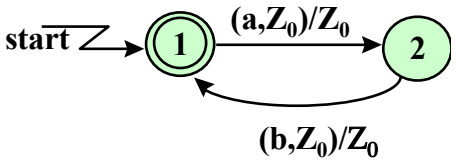
The reason why this is true is simple. Suppose DPDA M accepts L by empty stack, and suppose L is not prefix-free. Then for some x and y in L, y = xu, where $u \neq \lambda$. Consider the behavior of M on input y. After reading x the stack

will be empty (x ∈L) and the DPDA cannot continue. Since M is deterministic, no other behavior is possible. Thus u can never be read and y can never be accepted.

Since λ is a proper prefix of any non-null string, it follows that any language that contains both λ and some non-null string cannot be prefix-free and therefore cannot be accepted by a DPDA by empty stack.

Now consider L'. Clearly ab and abab both belong to L'. Thus L' is not prefix-free and cannot be accepted by any DPDA that accepts by empty stack.

To complete our proof we offer a DPDA, M', for which $L_{ste}(M') = L' = \{(ab)^n \mid n > 0 \}$. First observe that L' is actually Regular. Thus L' can be accepted by a DPDA that does not use its stack at all!



The DPDA, M'

Corollary 36-3. $DCFL_{stk} \subset DCFL_{ste} \subset CFL$. Furthermore, if R denotes the Regular languages, then $R - DCFL_{stk} \neq \Phi$ and $DCFL_{stk} - R \neq \Phi$, but $R \subset DCFL_{ste}$.

Added material from C. E. Hughes

Cocke-Kasami-Younger (CKY) Context Free Language Recognizer

This $O(n^3)$ parsing algorithm was originally designed to work for just Chomsky Normal Form (CNF). Fortunately, the conversion of an arbitrary CFG, G = (N, Σ, S, P), to an equivalent CNF increases the size of the grammar to at most a square of its original size so, while this will add to the constants associated with our n^3.

The key concept behind CKY is that any derivation of a string $w = a_1\ a_2\ a_3\ \ldots\ a_n$, where $k > 1$, from a CNF grammar must start with $S \Rightarrow A\ B$ where $A,B \in N$ and, for some j, $1 < j \leq n$,
$A \Rightarrow^* a_1\ \ldots\ a_{j-1}$
$B \Rightarrow^* a_j\ \ldots\ a_n$

While CKY uses this concept, it uses it in a bottom up manner. Here we compute, for each i, the set of non-terminals that derive a_i directly. In effect, we start by determining the set of non-terminals that derive the substring of w starting at a_i and extending for a total length of just 1 character. We then compute, for each i, $1 \leq i < n$, the set of non-terminals that derive the substring of w starting at a_i and extending for a total length of 2 characters. This can be computed by looking at set of all pairs, (A,B), such that $A \Rightarrow a_i$ and $B \Rightarrow a_{i+1}$. We then add any C such that $C \rightarrow A\ B$ is one of the productions in G. This continues with substrings of length 3 and so on until we end with the only substring of length n, w.

We implement this process by drawing the upper triangle of an n×n matrix, where |w| = n. We then label the columns

a_1 a_2 a_3 … a_n

We label the rows 1 .. n. We refer to the cells as P[i,j], where i is the row and j is the column

We can quickly fill in the first row by defining P[1,j] to be the set of all A in N, such that $A \rightarrow a_i$.

We then fill in successive rows using the following rule

Add C to P[i,j], $i > 1$, $j \leq n-i+1$, whenever there is a $1 \leq k < i$, such that $A \in P[k,j]$, $B \in P[i-k,j+k]$ and $C \rightarrow A\ B$.

Note: $A \in P[k,j]$ means that $A \Rightarrow^* a_j\ \ldots\ a_{j+k-1}$; $B \in P[i-k,j+k]$ means that $B \Rightarrow^* a_{j+k}\ \ldots\ a_{j+i-1}$

The goal is to see what non-terminals can derive w. This will end up in the cell P[n,1].
Thus, $w \in L(G)$ if and only if $S \in P[n,1]$.

Example:

Present the **CKY** recognition matrix for the string **b b a a b a** assuming the grammar specified by the rules

S -> S T | T S | a
T -> B S | b
B -> b

	b	**b**	**a**	**a**	**b**	**a**
1	**T,B**	**T,B**	**S**	**S**	**T,B**	**S**
2		**S,T**		**S**	**S,T**	
3	**S,T**	**S**		**S**		
4	**S,T**	**S**				
5	**S,T**	**S**				
6	**S,T**					

A slight variation:
From Wikipedia

```
Let the input be a string S consisting of n characters: a1 ... an.
Let the grammar contain r nonterminal symbols R1 ... Rr.
This grammar contains the subset Rs which is the set of start symbols.
Let P[n,n,r] be an array of booleans. Initialize all elements of P to false.
For each i = 1 to n
  For each unit production Rj -> ai, set P[i,1,j] = true.
For each i = 2 to n -- Length of span
  For each j = 1 to n-i+1 -- Start of span
    For each k = 1 to i-1 -- Partition of span
      For each production RA -> RB RC
        If P[j,k,B] and P[j+k,i-k,C] then set P[j,i,A] = true
If any of P[1,n,x] is true (x is iterated over the set s, where s are all the
indices for Rs)
  Then S is member of language
  Else S is not member of language
```