Chapter 6

Formal Language Theory

In this chapter, we introduce formal language theory, the computational theories of languages and grammars. The models are actually inspired by formal logic, enriched with insights from the theory of computation.

We begin with the definition of a language and then proceed to a rough characterization of the basic *Chomsky hierarchy*. We then turn to a more detailed consideration of the types of languages in the hierarchy and automata theory.

6.1 Languages

What is a language? Formally, a language L is defined as as set (possibly infinite) of strings over some finite alphabet.

Definition 7 (Language) *A language L is a possibly infinite set of strings over a finite alphabet Σ.*

We define Σ^* as the set of all possible strings over some alphabet Σ. Thus $L \subseteq \Sigma^*$. The set of all possible languages over some alphabet Σ is the set of all possible subsets of Σ^*, i.e. 2^{Σ^*} or $\wp(\Sigma^*)$. This may seem rather simple, but is actually perfectly adequate for our purposes.

6.2 Grammars

A grammar is a way to characterize a language L, a way to list out which strings of Σ^* are in L and which are not. If L is finite, we could simply list

the strings, but languages by definition need not be finite. In fact, all of the languages we are interested in are infinite. This is, as we showed in chapter 2, also true of human language.

Relating the material of this chapter to that of the preceding two, we can view a grammar as a logical system by which we can prove things. For example, we can view the strings of a language as WFFs. If we can prove some string u with respect to some language L, then we would conclude that u is in L, i.e. $u \in L$.

Another way to view a grammar as a logical system is as a set of formal statements we can use to prove that some particular string u follows from some initial assumption. This, in fact, is precisely how we presented the syntax of sentential logic in chapter 4. For example, we can think of the symbol WFF as the initial assumption or symbol of any derivational tree of a well-formed formula of sentential logic. We then follow the rules for atomic statements (page 47) and WFFs (page 47).

Our notion of grammar will be more specific, of course. The grammar includes a set of rules from which we can derive strings. These rules are effectively statements of logical equivalence of the form: $\psi \rightarrow \omega$, where ψ and ω are strings.[1]

Consider again the WFFs of sentential logic. We know a formula like $(p \wedge q')$ is well-formed because we can progress upward from atomic statements to WFFs showing how each fits the rules cited above. For example, we know that p is an atomic statement and q is an atomic statement. We also know that if q is an atomic statement, then so is q'. We also know that any atomic statement is a WFF. Finally, we know that two WFFs can be assembled together into a WFF with parentheses around the whole thing and a conjunction $\wedge$ in the middle.

We can represent all these steps in the form $\psi \rightarrow \omega$ if we add some additional symbols. Let's adopt W for a WFF and A for an atomic statement. If we know that p and q can be atomic statements, then this is equivalent to $A \rightarrow p$ and $A \rightarrow q$. Likewise, we know that any atomic statement followed by a prime is also an atomic statement: $A \rightarrow A'$. We know that any atomic statement is a WFF: $W \rightarrow A$. Last, we know that any two WFFs can be

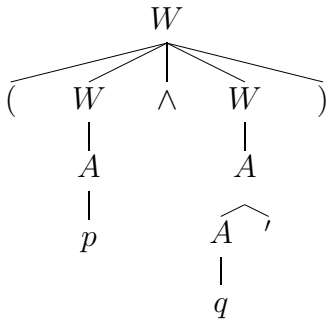
[1]These statements seem to go in only one direction, yet they are not bound by the restriction we saw in first-order logic where a substitution based on logical consequence can only apply to an entire formula. It's probably best to understand these statements as more like biconditionals, rather than conditionals, even though the traditional symbol here is the same as for a logical conditional.

conjoined: $W \rightarrow (W \wedge W)$.

Each of these rules is part of the grammar of the syntax of WFFs. If every part of a formula follows one of the rules of the grammar of the syntax of WFFs, then we say that the formula is indeed a WFF.

Returning to the example $(p \wedge q')$, we can show that every part of the formula follows one of these rules by constructing a tree.

(6.1)



Each branch corresponds to one of the rules we posited. The mother of each branch corresponds to ψ and the daughters to ω. The elements at the very ends of branches are referred to as *terminal elements*, and the elements higher in the tree are all *non-terminal elements*. If all branches correspond to actual rules of the grammar and the top node is a legal starting node, then the string is syntactically well-formed with respect to that grammar.

Formally, we define a grammar as $\{V_T, V_N, S, R\}$, where V_T is the set of terminal elements, V_N is the set of non-terminals, S is a member of V_N, and R is a finite set of rules of the form above. The symbol S is defined as the only legal 'root' non-terminal. As in the preceding example, we use capital letters for non-terminals and lowercase letters for terminals.

Definition 8 (Grammar) *$\{V_T, V_N, S, R\}$, where V_T is the set of terminal elements, V_N is the set of non-terminals, S is a member of V_N, and R is a finite set of rules.*

Looking more closely at R, we will require that the left side of a rule contain at least one non-terminal element and any number of other elements. We define Σ as $V_T \cup V_N$, all of the terminals and non-terminals together. R is a finite set of ordered pairs from $\Sigma^* V_N \Sigma^* \times \Sigma^*$. Thus $\psi \rightarrow \omega$ is equivalent to $\langle \psi, \omega \rangle$.

Definition 9 (Rule) *R is a finite set of ordered pairs from $\Sigma^* V_N \Sigma^* \times \Sigma^*$, where $\Sigma = V_T \cup V_N$.*

We can now consider grammars of different types. The simplest case to consider first, from this perspective, are *context-free* grammars, or *Type 2* grammars. In such a grammar, all rules of R are of the form $A \rightarrow \psi$, where A is a single non-terminal element of V_N and ψ is a string of terminals from V_T and non-terminals from V_N. Such a rule says that a non-terminal A can dominate the string ψ in a tree. These are the traditional *phrase-structure* taught in introductory linguistics courses. The set of languages that can be generated with such a system is fairly restricted and derivations are straightforwardly represented with a syntactic tree. The partial grammar we exemplified above for sentential logic was of this sort.

A somewhat more powerful system can be had if we allow *context-sensitive* rewrite rules, e.g. $A \rightarrow \psi / \alpha \_\_ \beta$ (where ψ cannot be ϵ). Such a rule says that A can dominate ψ in a tree if ψ is preceded by α and followed by β. If we set trees aside, and just concentrate on string equivalences, then this is equivalent to $\alpha A \beta \rightarrow \alpha \psi \beta$. Context-sensitive grammars are also referred to as *Type 1* grammars.

In the other direction from context-free grammars, that is toward *less* powerful grammars, we have the *regular* or *right-linear* or *Type 3* grammars. Such grammars only contain rules of the following form: $A \rightarrow xB$ or $A \rightarrow x$. The non-terminal A can be rewritten as a single terminal element x or a single non-terminal followed by a single terminal.

(6.2)

1	context-sensitive	$A \rightarrow \psi / \alpha \_\_ \beta$
2	context-free	$A \rightarrow \psi$
3	right-linear	$\left\{ \begin{array}{c} A \rightarrow x\ B \\ A \rightarrow x \end{array} \right\}$

We will see that these three types of grammars allow for successively more restrictive languages and can be paired with specific types of abstract models of computers. We will also see that the formal properties of the most restrictive grammar types are quite well understood and that as we move up the hierarchy, the systems become less and less well understood, or, more and more interesting.

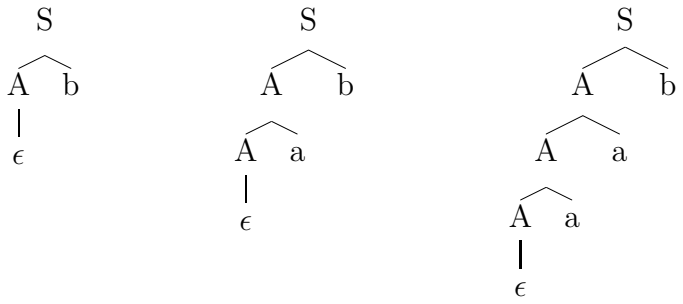
Let's look at a few examples. For all of these, assume the alphabet is $\Sigma = \{a, b, c\}$.

How might we define a grammar for the language that includes all strings composed of one instance of b preceded by any number of instances of a: $\{b, ab, aab, aaab, \ldots\}$? We must first decide what sort of grammar to write among the three types we've discussed. In general, context-free grammars are the easiest and most intuitive to write. In this case, we might have something like this:

(6.3) $S \rightarrow A\ b$
$A \rightarrow \epsilon$
$A \rightarrow A\ a$

This is an instance of a context-free grammar because all rules have a single non-terminal on the left and a string of terminals and non-terminals on the right. This grammar cannot be right-linear because it includes rules where the right side has a non-terminal followed by a terminal. This grammar cannot be context-sensitive because it contains rules where the right side is ϵ. For the strings b, ab, and aab, this produces the following trees.

(6.4)



In terms of our formal characterization of grammars, we have:

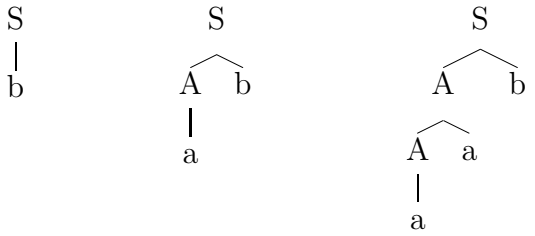
$$
\begin{array}{rcl}
V_T & = & \{a, b\} \\
V_N & = & \{S, A\} \\
S & = & S \\
R & = & \left\{ \begin{array}{l} S \rightarrow A\ b \\ A \rightarrow \epsilon \\ A \rightarrow A\ a \end{array} \right\}
\end{array} \tag{6.5}
$$

Other grammars are possible for this language too. For example:

$$
\begin{array}{l}
S \rightarrow b \\
S \rightarrow A\ b \\
A \rightarrow a \\
A \rightarrow A\ a
\end{array} \tag{6.6}
$$

This grammar is context-free, but also qualifies as context-sensitive. We no longer have ϵ on the right side of any rule and a single non-terminal on the left qualifies as a string of terminals and non-terminals. This grammar produces the following trees for the same three strings.

(6.7)

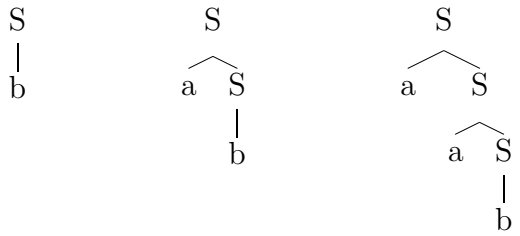


We can also write a grammar that qualifies as right-linear that will characterize this language.

$$
\begin{array}{l}
S \rightarrow b \\
S \rightarrow a\ S
\end{array} \tag{6.8}
$$

This produces trees as follows for our three examples.

(6.9)

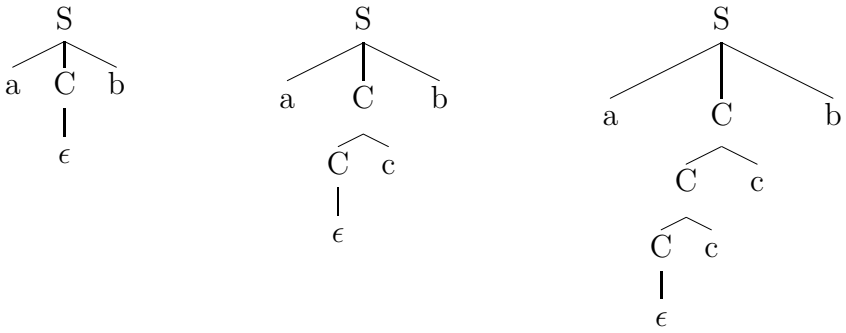


Let's consider a somewhat harder case: a language where strings begin with an a, end with a b, with any number of intervening instances of c, e.g. $\{ab, acb, accb, \ldots\}$. This can be described using all three grammar types. First, a context-free grammar:

(6.10) $S \rightarrow a\ C\ b$
$C \rightarrow C\ c$
$C \rightarrow \epsilon$

This grammar is neither right-linear nor context-sensitive. It produces trees like these:

(6.11)

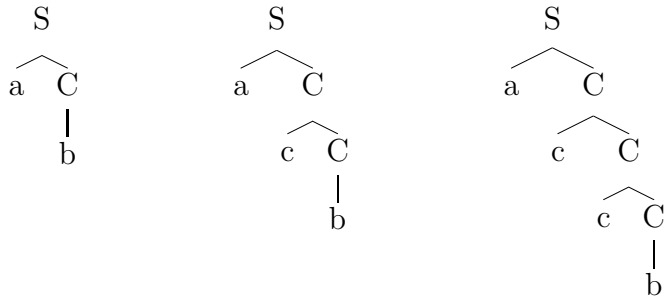


Here is a right-linear grammar that generates the same strings:

(6.12) $S \rightarrow a\ C$
$C \rightarrow c\ C$
$C \rightarrow b$

This produces trees as follows for the same three examples:

(6.13)

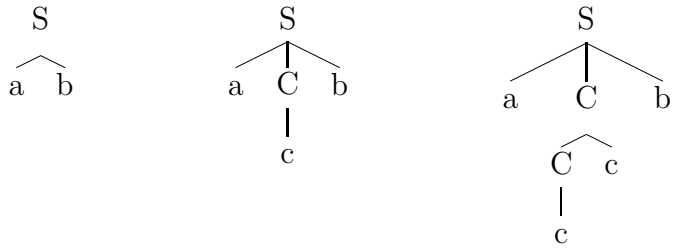


We can also write a grammar that is both context-free and context-sensitive that produces this language.

(6.14) $S \rightarrow a\ b$
$S \rightarrow a\ C\ b$
$C \rightarrow C\ c$
$C \rightarrow c$

This results in the following trees.

(6.15)



We will see that the set of languages that can be described by the three types of grammar are not the same. Right-linear grammars can only accommodate a subset of the languages that can be treated with context-free and context-sensitive grammars. If we set aside the null string ϵ, context-free grammars can only handle a subset of the languages that context-sensitive grammars can treat.

In the following sections, we more closely examine the properties of the sets of languages each grammar formalism can accommodate and the set of abstract machines that correspond to each type.

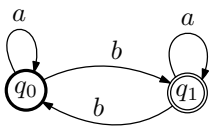
6.3 Finite State Automata

In this section, we treat finite state automata. We consider two types of finite state automata: deterministic and non-deterministic. We define each formally and then show their equivalence.

What is a *finite automaton*? In intuitive terms, it is a very simple model of a computer. The machine reads an input tape which bears a string of symbols. The machine can be in any number of states and, as each symbol is read, the machine switches from state to state based on what symbol is read at each point. If the machine ends up in one of a set of particular states, then the string of symbols is said to be *accepted*. If it ends up in any other state, then the string is not accepted.

What is a finite automaton more formally? Let's start with a *deterministic finite automaton* (DFA). A DFA is a machine composed of a finite set of states linked by arcs labeled with symbols from a finite alphabet. Each time a symbol is read, the machine changes state, the new state *uniquely* determined by the symbol read and the labeled arcs from the current state. For example, imagine we have an automaton with the structure in figure 6.16 below.

(6.16)



There are two states q_0 and q_1. The first state, q_0, is the designated start state and the second state, q_1, is a designated final state. This is indicated with a dark circle for the start state and a double circle for any final state. The alphabet Σ is defined as $\{a, b\}$.

This automaton describes the language where all strings contain an odd number of the symbol b, for it is only with an input string that satisfies that restriction that the automaton will end up in state q_1. For example, let's go through what happens when the machine reads the string *bab*. It starts in state q_0 and reads the first symbol b. It then follows the arc labeled b to

state q_1. It then reads the symbol a and follows the arc from q_1 back to q_1. Finally, it reads the last symbol b and follows the arc back to q_0. Since q_0 is not a designated final state, the string is not accepted.

Consider now a string *abbb*. The machine starts in state q_0 and reads the symbol a. It then follows the arc back to q_0. It reads the first b and follows the arc to q_1. It reads the second b and follows the arc labeled b back to q_0. Finally, it reads the last b and follows the arc from q_0 back to q_1. Since q_1 is a designated final state, the string is accepted.

We can define a DFA more formally as follows:

Definition 10 (DFA) *A deterministic finite automaton (DFA) is a quintuple $\langle K, \Sigma, q_0, F, \delta \rangle$, where K is a finite number of states, Σ is a finite alphabet, $q_0 \in K$ is a single designated start state, and δ is a function from $K \times \Sigma$ to K.*

For example, in the DFA in figure 6.16, K is $\{q_0, q_1\}$, Σ is $\{a, b\}$, q_0 is the designated start state and $F = \{q_1\}$. The function δ has the following domain and range:

(6.17)

domain	range
q_0, a	q_0
q_0, b	q_1
q_1, a	q_1
q_1, b	q_0

Thus, the function δ can be represented either graphically as arcs, as in (6.16), or textually as a table, as in (6.17).[2]

The *situation* of a finite automaton is a triple: (x, q, y), where x is the portion of the input string that the machine has already "consumed", q is the current state, and y is the part of the string on the tape yet to be read. We can think of the progress of the tape as a sequence of situations licensed by δ. Consider what happens when we feed *abab* to the DFA in figure 6.16. We start with $(\epsilon, q_0, abab)$ and then go to (a, q_0, bab), then to (ab, q_1, ab), etc.

[2]Some treatments distinguish deterministic automata from complete automata. A deterministic automaton has no more than one arc from any state labeled with any particular symbol. A complete automaton has at least one arc from every state for every symbol.

The steps of the derivation are encoded with the turnstile symbol $\vdash$. The entire derivation is given below:

(6.18) $(\epsilon, q_0, abab) \vdash (a, q_0, bab) \vdash (ab, q_1, ab) \vdash (aba, q_1, b) \vdash (abab, q_0, \epsilon)$

Since the DFA does not end up in a state of F ($q_0 \notin F$), this string is not accepted.

Let's define the turnstile more formally as follows:

Definition 11 (produces in one move) *Assume a DFA M, where $M = \langle K, \Sigma, \delta, q_0, F\rangle$. A situation (x, q, y) produces situation (x', q', y') in one move iff: 1) there is a symbol $\sigma \in \Sigma$ such that $y = \sigma y'$ and $x' = x\sigma$ (i.e., the machine reads one symbol), and 2) $\delta(q, \sigma) = q'$ (i.e., the appropriate state change occurs on reading σ).*

We can use this to define a notion "produces in zero or more steps": $\vdash^*$. We say that $S_1 \vdash^* S_n$ if there is a sequence of situations $S_1 \vdash S_2 \vdash \ldots \vdash S_{n-1} \vdash S_n$. Thus the derivation in (6.18) is equivalent to the following.

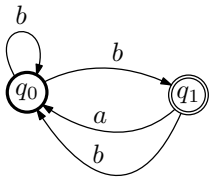
(6.19) $(\epsilon, q_0, abab) \vdash^* (abab, q_0, \epsilon)$

Let's now consider *non-deterministic finite automata* (NFAs). These are just like DFAs except i) arcs can be labeled with the null string ϵ, and ii) there can be multiple arcs with the same label from the same state; thus δ is a relation, not a function, in a NFA.

Definition 12 (NFA) *A non-deterministic finite automaton M is a quintuple $\langle K, \Sigma, \Delta, q_0, F\rangle$, where K, Σ, q_0, and F are as for a DFA, and Δ, the transition relation, is a finite subset of $K \times (\Sigma \cup \epsilon) \times K$.*

Let's look at an example. The NFA in (6.20) generates the language where any instance of the symbol a must have at least one b on either side of it; the string must begin with at least one instance of b.

(6.20)



Here, q_0 is the designated start state and q_1 is in F. We can see that there are *two* arcs from q_0 on b, but none on a; this is thus necessarily non-deterministic.

The transition relation Δ can be represented in tabular form as well. Here, we list all the mappings for every combination of $K \times \Sigma^*$.

(6.21)

domain	range
q_0, a	$\emptyset$
q_0, b	$\{q_0, q_1\}$
q_1, a	$\{q_0\}$
q_1, b	$\{q_0\}$

Given that there are multiple paths through an NFA for any particular string, how do we assess whether a string is accepted by the automaton? To see if some string is accepted by a NFA, we see if there is at least *one* path through the automaton that terminates in a state of F.

Consider the automaton above and the string *bab*. There are several paths that work.

(6.22) a. $(\epsilon, q_0, bab) \vdash (b, q_0, ab) \vdash ?$

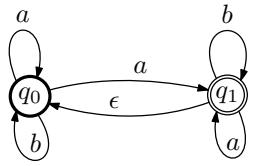
b. $(\epsilon, q_0, bab) \vdash (b, q_1, ab) \vdash (ba, q_0, b) \vdash (bab, q_0, \epsilon)$

c. $(\epsilon, q_0, bab) \vdash (b, q_1, ab) \vdash (ba, q_0, b) \vdash (bab, q_1, \epsilon)$

There are three paths possible. The first, (6.22a), doesn't terminate. The second terminates, but only in a non-final state. The third, (6.22c), terminates in a final state. Hence, since there is at least one path that terminates in a final state, the string is accepted.

It's a little trickier when the NFA contains arcs labeled with ϵ. For example:

(6.23)



Here we have the usual sort of non-determinism with two arcs labeled with a from q_0. We also have an arc labeled ϵ from q_1 to q_0. This latter sort of arc can be followed at any time without consuming a symbol. Let's consider how a string like *aba* might be parsed by this machine. The following chart shows all possible paths.

(6.24)
a. $(\epsilon, q_0, aba) \vdash (a, q_0, ba) \vdash (ab, q_0, a) \vdash (aba, q_1, \epsilon)$
b. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (ab, q_1, a) \vdash (aba, q_1, \epsilon)$
c. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (a, q_0, ba) \vdash (ab, q_0, a) \vdash (aba, q_0, \epsilon)$
d. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (ab, q_1, a) \vdash (ab, q_0, a) \vdash (aba, q_0, \epsilon)$
e. $(\epsilon, q_0, aba) \vdash (a, q_0, ba) \vdash (ab, q_0, a) \vdash (aba, q_1, \epsilon) \vdash (aba, q_0, \epsilon)$
f. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (ab, q_1, a) \vdash (aba, q_1, \epsilon) \vdash (aba, q_0, \epsilon)$
g. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (a, q_0, ba) \vdash (ab, q_0, a) \vdash (aba, q_1, \epsilon)$
h. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (a, q_0, ba) \vdash (ab, q_0, a) \vdash$
$(aba, q_1, \epsilon) \vdash (aba, q_0, \epsilon)$
i. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (ab, q_1, a) \vdash (ab, q_0, a) \vdash (aba, q_1, \epsilon)$
j. $(\epsilon, q_0, aba) \vdash (a, q_1, ba) \vdash (ab, q_1, a) \vdash (ab, q_0, a) \vdash$
$(aba, q_1, \epsilon) \vdash (aba, q_0, \epsilon)$

The ϵ-arc can be followed whenever the machine is in state q_1. It is indicated in the chart above by a move from q_1 to q_0 without a symbol being read. Note that it results in an explosion in the number of possible paths. In this case, since at least one string ends up in the designated final state q_1, the string is accepted.

Notice that it's a potentially very scary proposition to determine if some NFA generates some string x. Given that there are ϵ-arcs, which can be followed at any time without reading a symbol, there can be an *infinite* number of paths for any finite string.[3] Fortunately, this is not a problem, because DFAs and NFAs generate the same class of languages.

Theorem 1 *DFAs and NFAs produce the same languages.*

Let's show this. DFAs are obviously a subcase of NFAs; hence any language generated by a DFA is trivially generated by an NFA.

[3]This can arise if we have cycles involving ϵ.

Proving this in the other direction is a little trickier. What we will do is show how a DFA can be constructed from any NFA (Hopcroft and Ullman, 1979). Recall that the arcs of an NFA can be represented as a map from $K \times (\Sigma \cup \epsilon)$ to all possible subsets of K. What we do to construct the DFA is to use these sets of states as literal labels for new states.

For example, in the NFA in (6.20), call it M, we have Δ as in (6.21). The possible sets of states are: $\emptyset$, $\{q_0\}$, $\{q_1\}$, and $\{q_0, q_1\}$.[4] The new DFA M' will then have state labels: $[\emptyset]$, $[q_0]$, $[q_1]$, and $[q_0, q_1]$, replacing the curly braces that denote sets with square braces which we will use to denote state labels. For the new DFA, we define δ' as follows:

$$\delta'([q_1, q_2, \ldots, q_n], a) = [p_1, p_2, \ldots, p_n]$$

if and only if, in the original NFA:

$$\Delta(\{q_1, q_2, \ldots, q_n\}, a) = \{p_1, p_2, \ldots, p_n\}$$

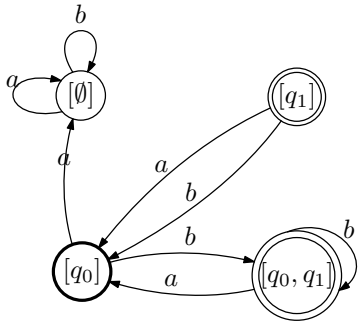
The latter means that we apply Δ to every state in the first list of states and union together the resulting states.

Applying this to the NFA in (6.20), we get this chart for the new DFA.

$$\begin{aligned}
\delta([\emptyset], a) &= [\emptyset] \\
\delta([\emptyset], b) &= [\emptyset] \\
\delta([q_0], a) &= [\emptyset] \\
\delta([q_0], b) &= [q_0, q_1] \\
\delta([q_1], a) &= [q_0] \\
\delta([q_1], b) &= [q_0] \\
\delta([q_0, q_1], a) &= [q_0] \\
\delta([q_0, q_1], b) &= [q_0, q_1]
\end{aligned} \tag{6.25}$$

The initial start state was q_0, so the new start state is $[q_0]$. Any set containing a possible final state from the initial automaton is a final state in the new automaton: $[q_1]$ and $[q_0, q_1]$. The new automaton is given below.

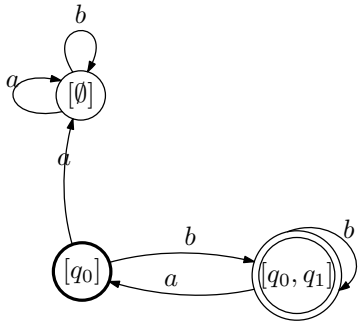
[4] Recall that there will be 2^K of these.



(6.26)

This automaton accepts exactly the same language as the previous one. If we can always construct a DFA from an NFA that accepts exactly the same language, it follows that there is no language accepted by an NFA that cannot be accepted by a DFA. □

Notice two things about the resulting DFA in (6.26). First, there is a state that cannot be reached: $[q_1]$. Such states can safely be pruned. The following automaton is equivalent to (6.26).



(6.27)

Second, notice that the derived DFA can, in principle, be massively bigger than the original NFA. In the worst case, if the original NFA has n states, the new automaton can have as many as 2^n states.[5]

In the following, since NFAs and DFAs are equivalent, I will refer to the general class as *Finite State Automata* (FSAs).

[5]There are algorithms for minimizing the number of states in a DFA, but they are beyond the scope of this introduction. See Hopcroft and Ullman (1979). Even minimized, it is generally true that an NFA will be smaller than its equivalent DFA.

6.4 Regular Languages

We now consider the class of regular languages. We'll show that these are precisely those that can be accepted by an FSA *and* which can be generated by a right-linear grammar.

The *regular* languages are defined as follows.

Definition 13 (Regular Language) *Given a finite alphabet Σ:*

1. *$\emptyset$ is a regular language.*
2. *For any string $x \in \Sigma^*$, $\{x\}$ is a regular language.*
3. *If A and B are regular languages, then so is $A \cup B$.*
4. *If A and B are regular languages, then so is AB.*
5. *If A is a regular language, then so is A^*.*
6. *Nothing else is a regular language.*

Consider each of these operations in turn. First, we have that any string of symbols from the alphabet can be a specification of a language. Thus, if the alphabet is $\Sigma = \{a, b, c\}$, then the regular language L can be $\{a\}$.[6]

If $L_1 = \{a\}$ and $L_2 = \{b\}$, then we can define the regular language which is the union of L_1 and L_2: $L_3 = L_1 \cup L_2$, i.e. $L_3 = \{a, b\}$. In string terms, this is usually written $L_3 = (a|b)$.

We can also concatenate two regular languages, e.g. $L_3 = L_1 L_2$, e.g. $L_3 = \{ab\}$.

Finally, we have *Kleene star*, which allows us to repeat some regular language zero or more times. Thus, if L_1 is a regular language, then $L_2 = L_1^*$ is a regular language, e.g. $L_2 = \{a, aa, aaa, \ldots\}$. In string terms: $L_2 = a^*$.

These operations can, of course, be applied recursively in any order.[7] For example, $a(b^*|c)a^*$ refers to the language where all strings are composed of a single instance of a followed by any number of instances of b or a single c, followed in turn by any number of instances of a.

We can go in the other direction as well. For example, how might we describe the language where all strings contain an even number of instances of a plus any number of the other symbols: $((b|c)^*a(b|c)^*a(b|c)^*)^*$.

[6] Notice that it would work just as well to start form any symbol $a \in \Sigma$ here, since we have recursive concatenation below.

[7] In complex examples, we can use parentheses to indicate scope.

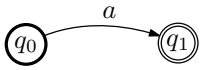
6.4.1 Automata and Regular Languages

It turns out that the set of languages that can be accepted by a finite state automaton is exactly the regular languages.

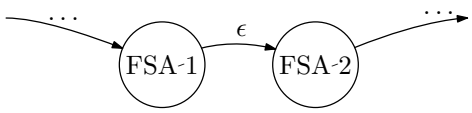
Theorem 2 *A set of strings is a finite automaton language if and only if it is a regular language.*

We won't prove this rigorously, but we can see the logic of the proof fairly straightforwardly. There are really only four things that we can do with a finite automaton, and each of these four correspond to one of the basic clauses of the definition of a regular language.

First, we have that a single symbol is a legal regular language because we can have a finite automaton with a start state, a single arc, and a final state.

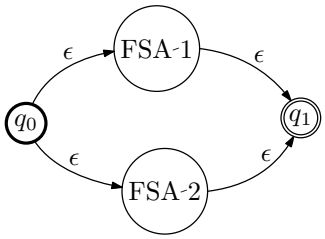
(6.28) 

Second, we have concatenation of two regular languages by taking two automata and connecting them with an arc labeled with ϵ.

(6.29) 

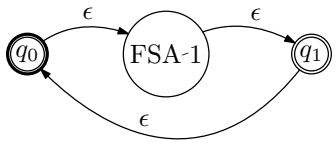
We connect all final states of the first automaton with the start state of the second automaton with ϵ-arcs. The final states of the first automaton are made non-final. The start state of the second automaton is made a non-start state.

Union is straightforward as well. We simply create a new start state and then create arcs from that state to the former start states of the two automata labeled with ϵ. We create a new final state as well, with ϵ-arcs from the former final states of the two automata.

(6.30) 

Finally, we can get Kleene star by creating a new start state (which is also a final state), a new final state, and an ϵ-loop between them.

(6.31)



If we can construct an automaton for every step in the construction of a regular language, it should follow that any regular language can be accepted by some automaton.[8]

6.4.2 Right-linear Grammars and Automata

Another equivalence that is of use is that between the regular languages and right-linear grammars. Right-linear grammars generate precisely the set of regular languages.

We can show this by pairing the rules of a right-linear grammar with the arcs of an automaton. First, for every rewrite rule of the form $A \rightarrow x\ B$, we have an arc from state A to state B labeled x. For every rewrite rule of the form $A \rightarrow x$, we have an arc from state A to the designated final state, call it F.

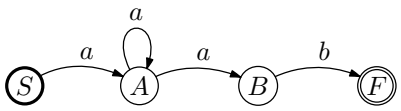
Consider this very simple example of a right-linear grammar.

(6.32)
a. $S \rightarrow a\ A$
b. $A \rightarrow a\ A$
c. $A \rightarrow a\ B$
d. $B \rightarrow b$

This generates the language where all strings are composed of two or more instances of a, followed by exactly one b.

If we follow the construction of the FSA above, we get this:

(6.33)



[8] A rigorous proof would require that we go through this in the other direction as well, from automaton to regular language.

This FSA accepts the same language generated by the right-linear grammar in (6.32).

Notice now that if FSAs and right-linear grammars generate the same set of languages and FSAs generate regular languages, then it follows that right-linear grammars generate regular languages. Thus we have a three-way equivalence between regular languages, right-linear grammars, and FSAs.

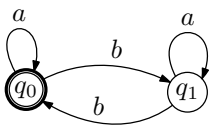
6.4.3 Closure Properties

Let's now turn to closure properties of the regular languages. By the definition of regular language, it follows that they are closed under the properties that define them: concatenation, union, Kleene star. They are also closed under complement. The complement of some regular language L defined over the alphabet Σ is $L' = \Sigma^* - L$.

It's rather easy to show this using DFAs. In particular, to construct the complement of some language L, we create the DFA that generates that language and then swap the final and non-final states.

Let's consider the DFA in (6.16) on page 102 above. This generates the language $a^*ba^*(ba^*ba^*)^*$, where every legal string contains an odd number of instances of the symbol b, and any number of instances of the symbol a. We now reverse the final and non-final states so that q_0 is both the start state and the final state.

(6.34)



This now generates the complement language: $a^*(ba^*ba^*)^*$. Every legal string has an even number of instances of b (including zero), and any number of instances of a.

With complement so defined, and DeMorgan's Law (the set-theoretic version), it follows that the regular languages are closed under intersection as well. Recall the following equivalences from chapter 3.

$$(X \cup Y)' = X' \cap Y' \tag{6.35}$$
$$(X \cap Y)' = X' \cup Y'$$

Therefore, since the regular languages are closed under union and under complement, it follows that they are closed under intersection. Thus if we want to intersect the languages L_1 and L_2, we union their complements, i.e. $L_1 \cap L_2 = (L_1' \cup L_2')'$.

6.5 Context-free Languages

In this section, we treat the *context-free* languages, generated with rules of the form $A \rightarrow \psi$, where A is a non-terminal and ψ is a string of terminals and non-terminals.

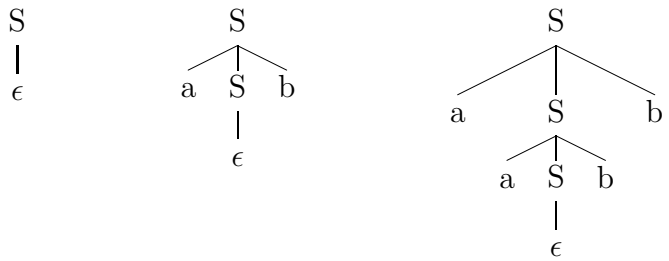
From the definition of right-linear grammars and context-free grammars, it follows that any language that can be described in terms of a right-linear grammar can be described in terms of a context-free grammar. This is true trivially since any right-linear grammar is definitionally also a context-free grammar.

What about in the other direction though? There are languages that can be described in terms of context-free grammars that cannot be described in terms of right-linear grammars. Consider, for example, the language $a^n b^n$: $\{\epsilon, ab, aabb, aaabbb, \ldots\}$. It can be generated by a context-free grammar, but not by a right-linear grammar. Here is a simple context-free grammar for this language:

(6.36) $S \rightarrow a\ S\ b$
$S \rightarrow \epsilon$

Here are some sample trees produced by this grammar.

(6.37)



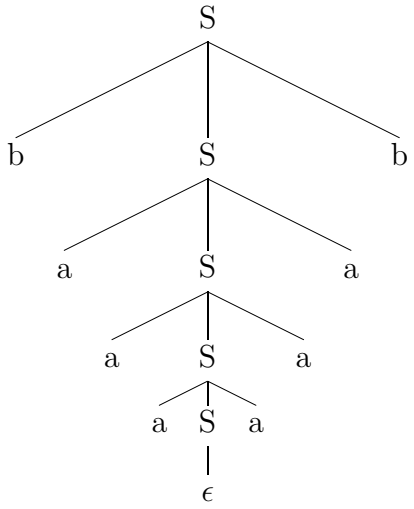
Another language type that cannot be treated with a right-linear grammar is xx^R where a string x is followed by its mirror-image x^R, including

strings like *aa*, *bb*, *abba*, *baaaaaab*, etc. This can be treated with a context-free grammar like this:

(6.38) $S \rightarrow a\ S\ a$
$S \rightarrow b\ S\ b$
$S \rightarrow \epsilon$

This produces trees like this one:

(6.39)



The problem is that both sorts of language require that we keep track of a potentially infinite amount of information over the string. Context-free grammars do this by allowing the edges of the right side of rules to depend on each other (with other non-terminals in between). This sort of dependency is, of course, not possible with a right-linear grammar.

6.5.1 Pushdown Automata

Context-free grammars are also equivalent to a particular simple computational model, e.g. a *non-deterministic pushdown automaton* (PDA). A PDA is just like a FSA, except it includes a *stack*, a memory store that can be utilized as each symbol is read from the tape.

The stack is restricted, however. In particular, symbols can be added to or read off of the top of the stack, but not to or from lower down in the stack.

For example, If the symbols a, b, and c are put on the stack in that order, they can only be retrieved from the stack in the opposite order: c, b, and then a. This is the intended sense of the term *pushdown*.[9]

Thus, at each step of the PDA, we need to know what state we are in, what symbol is on the tape, and what symbol is on top of the stack. We can then move to a different state, reading the next symbol on the tape, adding or removing the topmost symbol of the stack, or leaving it intact. A string is accepted by a PDA if the following hold:

1. the whole input has been read;
2. the stack is empty;
3. the PDA is in a final state.

A non-deterministic pushdown automaton can be defined more formally as follows:

Definition 14 *A non-deterministic PDA is a sextuple* $\langle K, \Sigma, \Gamma, s, F, \Delta \rangle$, *where* K *is a finite set of states,* Σ *is a finite set (the input alphabet),* Γ *is a finite set (the stack alphabet),* $s \in K$ *is the initial state,* $F \subseteq K$ *is the set of final states, and* Δ, *the set of transitions is a finite subset of* $K \times (\Sigma \cup \epsilon) \times (\Gamma \cup \epsilon) \times K \times (\Gamma \cup \epsilon)$.

Let's consider an example. Here is a PDA for $a^n b^n$.

(6.40)

States:	K	$=$	$\{q_0, q_1\}$
Input alphabet:	Σ	$=$	$\{a, b\}$
Stack alphabet:	Γ	$=$	$\{c\}$
Initial state:	s	$=$	q_0
Final states:	F	$=$	$\{q_0, q_1\}$
Transitions:	Δ	$=$	$\left\{ \begin{array}{l} (q_0, a, \epsilon) \rightarrow (q_0, c) \\ (q_0, b, c) \rightarrow (q_1, \epsilon) \\ (q_1, b, c) \rightarrow (q_1, \epsilon) \end{array} \right\}$

[9] A stack is also referred to as "last in first out" (LIFO) memory.

The PDA puts the symbol c on the stack every time it reads the symbol a on the tape. As soon as it reads the symbol b, it removes the topmost c from the stack and moves to state q_1, where it removes an c from the stack for every b that it reads on the tape. If the same number of instances of a and b are read, then the stack will be empty when there are no more symbols on the tape.

To see this more clearly, let us define a *situation* for a PDA as follows.

Definition 15 (Situation of a PDA) *A situation of a PDA is a quadruple* (x, q, y, z)*, where* $q \in K$*,* $x, y \in \Sigma^*$*, and* $z \in \Gamma^*$*.*

This is just like the situation of an FSA, except that it includes a specification of the state of the stack in z.

Consider now the sequence of situations which shows the operation of the previous PDA on the string *aaabbb*.

$$(6.41) \quad \begin{aligned}&(\epsilon, q_0, aaabbb, \epsilon) \vdash (a, q_0, aabbb, c) \vdash (aa, q_0, abbb, cc) \vdash \\ &(aaa, q_0, bbb, ccc) \vdash (aaab, q_1, bb, cc) \vdash (aaabb, q_1, b, c) \vdash \\ &(aaabbb, q_1, \epsilon, \epsilon)\end{aligned}$$

Notice that this PDA is deterministic in the sense that there is no more than one arc from any state on the same symbol.[10] This PDA still qualifies as non-deterministic under Definition 14, since deterministic automata are always a subset of non-deterministic automata.

The context-free languages cannot all be treated with deterministic PDAs, however. Consider the language xx^R, where a string is followed by its mirror image, e.g. *aa*, *abba*, *bbaabb*, etc. We've already seen that this is trivially generated using context-free rules. Here is a non-deterministic PDA that generates the same language.

[10]Notice that the PDA is not *complete*, as there is no arc on a from state q_1.

$$
\begin{array}{lrcl}
\text{States:} & K & = & \{q_0, q_1\} \\
\text{Input alphabet:} & \Sigma & = & \{a, b\} \\
\text{Stack alphabet:} & \Gamma & = & \{A, B\} \\
\text{Initial state:} & s & = & q_0 \\
\text{Final states:} & F & = & \{q_0, q_1\} \\
\text{Transitions:} & \Delta & = & \left\{ \begin{array}{l} (q_0, a, \epsilon) \rightarrow (q_0, A) \\ (q_0, b, \epsilon) \rightarrow (q_0, B) \\ (q_0, a, A) \rightarrow (q_1, \epsilon) \\ (q_0, b, B) \rightarrow (q_1, \epsilon) \\ (q_1, a, A) \rightarrow (q_1, \epsilon) \\ (q_1, b, B) \rightarrow (q_1, \epsilon) \end{array} \right\}
\end{array} \tag{6.42}
$$

Here is the sequence of situations for *abba* that results in the string being accepted.

$$
\begin{array}{l}
(\epsilon, q_0, abba, \epsilon) \vdash (a, q_0, bba, A) \vdash (ab, q_0, ba, BA) \vdash \\
(abb, q_1, a, A) \vdash (abba, q_1, \epsilon, \epsilon)
\end{array} \tag{6.43}
$$

Notice that at any point where two identical symbols occur in a row, the PDA can guess wrong and presume the reversal has occurred or that it has not. In the case of *abba*, the second b does signal the beginning of the reversal, but in *abbaabba*, the second b does not signal the beginning of the reversal. With a string of all identical symbols, like *aaaaaa*, there are many ways to go wrong.

This PDA is necessarily non-deterministic. There is no way to know, locally, when the reversal begins. It then follows that the set of languages that are accepted by a deterministic PDA are not equivalent to the set of languages accepted by a non-deterministic PDA. For example, any kind of PDA can accept $a^n b^n$, but only a non-deterministic PDA will accept xx^R.

We've said that non-determistic PDAs accept the set of languages generated by context-free grammars.

Theorem 3 *Context-free grammars generate the same kinds of languages as non-deterministic pushdown automata.*

This is actually rather complex to show, but we will show how to construct a non-deterministic PDA from a context-free grammar. Given a CFG $G = \langle V_N, V_T, S, R \rangle$, we construct a non-deterministic PDA as follows.

1. $K = \{q_0, q_1\}$
2. $s = q_0$
3. $F = \{q_1\}$
4. $\Sigma = V_T$
5. $\Gamma = \{V_N \cup V_T\}$

There are only two states, one being the start state and the other the sole final state. The input alphabet is identical to the set of terminal elements allowed by the CFG and the stack alphabet is identical to the set of terminal plus non-terminal elements.

The transition relation Δ is constructed as follows:

1. $(q_0, \epsilon, \epsilon) \rightarrow (q_1, S)$ is in Δ.
2. For each rule of the CFG of the form $A \rightarrow \psi$, Δ includes a transition $(q_1, \epsilon, A) \rightarrow (q_1, \psi)$.
3. For each symbol $a \in V_T$, there is a transition $(q_1, a, a) \rightarrow (q_1, \epsilon)$.

Let's consider how this works with a simple context-free grammar:

(6.44)
$$\begin{aligned} S &\rightarrow NP\ VP \\ VP &\rightarrow V\ NP \\ NP &\rightarrow N \\ N &\rightarrow \text{John} \\ N &\rightarrow \text{Mary} \\ V &\rightarrow \text{loves} \end{aligned}$$

We have K, s, and F as above. For Σ and Γ, we have:

(6.45) $\Sigma = \{\text{John}, \text{loves}, \text{Mary}\}$
$\Gamma = \{S, NP, VP, V, N, \text{John}, \text{loves}, \text{Mary}\}$

The transitions of Δ are as follows:

(6.46)
$$\begin{aligned}
(q_0, \epsilon, \epsilon) &\rightarrow (q_1, S) \\
(q_1, \epsilon, S) &\rightarrow (q_1, NP\ VP) \\
(q_1, \epsilon, NP) &\rightarrow (q_1, N) \\
(q_1, \epsilon, VP) &\rightarrow (q_1, V\ NP) \\
(q_1, \epsilon, N) &\rightarrow (q_1, \text{John}) \\
(q_1, \epsilon, N) &\rightarrow (q_1, \text{Mary}) \\
(q_1, \epsilon, V) &\rightarrow (q_1, \text{loves}) \\
(q_1, \text{John}, \text{John}) &\rightarrow (q_1, \epsilon) \\
(q_1, \text{Mary}, \text{Mary}) &\rightarrow (q_1, \epsilon) \\
(q_1, \text{loves}, \text{loves}) &\rightarrow (q_1, \epsilon)
\end{aligned}$$

Let's now look at how this PDA treats an input sentence like *Mary loves John.*

(6.47)
$$\begin{aligned}
&(\epsilon, q_0, \text{Mary loves John}, \epsilon) \vdash \\
&(\epsilon, q_1, \text{Mary loves John}, S) \vdash \\
&(\epsilon, q_1, \text{Mary loves John}, NP\ VP) \vdash \\
&(\epsilon, q_1, \text{Mary loves John}, N\ VP) \vdash \\
&(\epsilon, q_1, \text{Mary loves John}, \text{Mary}\ VP) \vdash \\
&(\text{Mary}, q_1, \text{loves John}, VP) \vdash \\
&(\text{Mary}, q_1, \text{loves John}, V\ NP) \vdash \\
&(\text{Mary}, q_1, \text{loves John}, \text{loves}\ NP) \vdash \\
&(\text{Mary loves}, q_1, \text{John}, NP) \vdash \\
&(\text{Mary loves}, q_1, \text{John}, N) \vdash \\
&(\text{Mary loves}, q_1, \text{John}, \text{John}) \vdash \\
&(\text{Mary loves John}, q_1, \epsilon, \epsilon)
\end{aligned}$$

This is not a proof that CFGs and PDAs are equivalent, but it shows the basic logic of that proof.

6.5.2 Closure Properties

The context-free languages are closed under a number of operations including union, concatenation, and Kleene star. They are *not* closed under complementation, nor are they closed under intersection.[11]

The demonstration that they are not generally closed under intersection is easy to see. One can show that $a^n b^n c^n$ is beyond the limits of context-free grammar. Now we know that $a^n b^n$ is context-free. We can complicate that just a little and still stay within the limits of context-free grammar: $a^n b^n c^m$, where though the first two symbols must be paired, there can be any number of instances of the third. If we try to intersect $a^n b^n c^m$ and $a^m b^n c^n$, which is also of course describable with a context-free grammar, we get $a^n b^n c^n$, which is not context-free.

6.5.3 Natural Language Syntax

The syntax of English cannot be regular. Consider these examples:

(6.48) The cat died.
The cat [the dog chased] died.
The cat [the dog [the rat bit] chased] died.
⋮

This is referred to as *center embedding*. Center-embedded sentences generally require a match between the number of subjects and the number of verbs. These elements do not occur adjacent to each other (except for the most embedded pair). This, then, is equivalent to $a^n b^n$ and beyond the range of the regular languages.[12]

Most speakers of English find these sentences rather marginal once they get above two or three clauses. It is thus a little disturbing that the best example of how natural language syntax cannot be regular is of this sort.

[11] Though they are, as you might expect, closed under intersection with a regular language.

[12] This argument is due to Partee et al. (1990).

Notice that if the grammar does not allow center-embedding beyond some specific number of clauses n, then the grammar can easily be regular. For example, imagine the upper bound on the $a^n b^n$ pattern is $n \leq 3$; this constitutes a finite set of sentences and can just be listed.

Is natural language syntax context-free or context-sensitive? Shieber (1985) argues that natural language syntax must be context-sensitive based on data from Swiss German. Examples like the following are grammatical. (Assume the sentence begins with the phrase *Jan säit das* 'John said that'.)

(6.49) mer d'chind em Hans es huus lönd hälfe aastriiche.
we children Hans house let help paint
'...we let the children help Hans paint the house.'

This is equivalent to the language xx, e.g. *aa*, *abab*, *abaaba*, etc., which is known not to be context-free.[13]

If the Swiss German pattern is correct, then it means that any formal account of natural language syntax requires more than a PDA and that a formalism based on context-free grammar is inadequate. Notice, however, that, once again, it is essential that the center-embedding be unbounded.

6.5.4 Other Properties of Context-free Languages

Notice that since context-free languages are accepted by *non-deterministic* PDAs, determining whether some particular string is accepted by such a PDA is a more complex operation than for the regular languages.

Recall that, for a regular language, we can always construct a DFA. Thus, for the regular languages, the number of steps we need to consider to determine the acceptability of some string s is equivalent to the length of that string.

On the other hand, if we are interested in whether some string s is accepted by a non-deterministic PDA, we must keep considering paths through the automaton until we find one that terminates with the appropriate properties: end of string, empty stack, in a final state. This may be quite a few paths to consider. Recall the context-free language xx^R and the non-deterministic PDA we described to treat it. For any string of length n, we

[13]Shieber actually goes further and shows that examples of this sort are not simply an accidental subset of the set of Swiss German sentences, but I leave this aside here.

must entertain the hypothesis that the reversal begins at any point between 1 and n. This entails that we must consider lots of paths for a long string.[14]

What this means, in concrete terms, is that if some phenomenon can be treated in finite-state terms or in context-free terms, and efficiency is a concern, go with the finite-state treatment.

6.6 Other Machines

There are other machines far more powerful than PDAs. For example, there are *Turing machines* (TMs). These are like FSAs except i) the reading head can move in either direction, and ii) it can write to as well as read from the tape. These properties allow TMs to use unused portions of the input tape as an unbounded memory store without the access limitations of a stack. Formally, a TM is defined as follows.

Definition 16 (Turing Machine) *A TM is a quadruple* (K, Σ, s, δ)*, where* K *is a finite set of states,* Σ *is a finite alphabet including #,* $s \in K$ *is the start state, and* δ *is a partial function from* $K \times \Sigma$ *to* $K \times (\Sigma \cup \{L, R\})$*.*

Here # is used to mark initially unused portions of the input tape. The logic of δ is that it maps from particular state/symbol combinations to a new state, simultaneously (potentially) writing a symbol to the tape and moving left or right.

TMs can describe far more complex languages than are thought to be appropriate for human language. For example $a^n b^{n!}$ can be treated with a TM. Likewise, a^n, where n is prime, can be handled with a TM. Hence, while there is a lot of work in computer science on the properties of TMs, there has not been a lot of work in grammatical theory using them.

Another machine type that we have not treated so far is the *finite state transducer* (FST). The basic idea is that we start with an FSA, but label the arcs with *pairs* of symbols. The machine can be thought of as reading two tapes in parallel. If the pairs of symbols match what is on the two tapes—and the machine finishes in a designated final state—then the pair of strings is

[14] It might be thought that we must consider an infinite number of paths, but this is not necessarily so. Any non-deterministic PDA with an infinite number of paths for some finite string can be converted into a non-deterministic PDA with only a finite number of paths for some finite string. See Hopcroft and Ullman (1979).

accepted. Another way to think of these, however, is that the machine reads one tape and spits out some symbol every time it transits an arc (perhaps writing those latter symbols to a new blank tape).

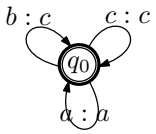
Formally, we can define an FST as follows:

Definition 17 (FST) *An FST is a sextuple* $(K, \Sigma, \Gamma, s, F, \Delta)$ *where* K *is a finite set of states,* Σ *is the finite input alphabet,* Γ *is the finite output alphabet,* $s \in K$ *is the start state,* $F \subseteq K$ *is the set of final states and* Δ *is a relation from* $K \times (\Sigma \cup \epsilon)$ *to* $K \times (\Gamma \cup \epsilon)$*.*

The relation Δ moves from state to state pairing symbols of Σ with symbols of Γ. The instances of ϵ in Δ allow it to insert or remove symbols, thus matching strings of different lengths.

For example, here is an FST that operates with the alphabet $\Sigma = \{a, b, c\}$, where anytime a b is confronted on one tape, the machine spits out c.

(6.50)



Such an FST would convert *abbcbcaa* into *acccccaa*.

The interest of such machines is twofold. First, like FSAs they are quite restricted in power and very well understood. Second, many domains of language and linguistics are modeled with input–output pairings and a transducer provides a tempting model for such a system. For example, in phonology, if we posit rules that nasalize vowels before nasal consonants, we might model that with a transducer that pairs oral vowels with nasal vowels just in case the following segment is a nasal consonant.

Let's look a little more closely at the kinds of relationships transducers allow (Kaplan and Kay, 1994). First we need a notion of n-way concatenation. This generalizes the usual notion of concatenation to transducers.

Definition 18 (n-way concatenation) *If* X *is an ordered tuple of strings* $\langle x_1, x_2, \ldots, x_n \rangle$ *and* Y *is an ordered tuple of strings* $\langle y_1, y_2, \ldots, y_n \rangle$ *then the n-way concatenation of* X *and* Y*,* $X \cdot Y$ *is defined as* $\langle x_1 y_1, x_2 y_2, \ldots x_n y_n \rangle$

We also need a way to talk about alphabets that include ϵ. We define $\Sigma^\epsilon = \{\Sigma \cup \epsilon\}$. With these in place, we can define the notion of a *regular relation*.

Definition 19 (Regular Relation) *We define the regular relations as follows:*

1. *The empty set and all a in $\Sigma^\epsilon \times \ldots \times \Sigma^\epsilon$ are n-way regular relations.*
2. *If R_1, R_2, and R are all regular relations, then so are:*

   $R_1 \cdot R_2 = \{xy \mid x \in R_1, y \in R_2\}$ *(n-way concatenation)*
   $R_1 \cup R_2$ *(union)*
   $R^* = \bigcup_{i=0}^{\infty} R^i$ *(n-way Kleene closure)*
3. *There are no other regular relations.*

It should be apparent that the regular relations are closed under the usual regular operations.[15] The regular relations are closed under a number of other operations too, e.g. the ones above, but also reversal, inverse, and composition.

They are *not* closed under intersection and complementation, however. For example, imagine we have

$$R_1 = \{\langle a^n, b^n c^* \rangle \mid n \geq 0\}$$

and

$$R_2 = \{\langle a^n, b^* c^n \rangle \mid n \geq 0\}$$

The intersection is clearly not regular. Each of these languages is, however, regular. We can see R_1 as $a : b^*\ \epsilon : c^*$ and R_2 as $\epsilon : b^*\ a : c^*$. It follows, of course, that the regular relations are not closed under complementation (by DeMorgan's Law).[16]

6.7 Summary

The chapter began with a presentation of the formal definition of language and grammar.

[15] Note that the regular relations are equivalent to the "rational relations" of algebra for the mathematically inclined.

[16] Same-length regular relations *are* closed under intersection.

We went on to consider different kinds of grammars, covering right-linear grammars, context-free grammars, and context-sensitive grammars. These increase in complexity as we go along, such that certain kinds of languages can only be described by a grammar sufficiently high up in the hierarchy. For example, $a^n b^n$ and ww^R require at least a context-free description, while ww and $a^n b^n c^n$ require a context-sensitive description.

We next turned to finite state automata. We gave a formal characterization and showed how these work. We showed how deterministic and non-deterministic FSAs are equivalent. We defined the regular languages and showed how FSAs and right-linear grammars accept and generate them. Finally, we showed how the regular languages are closed under their defining properties, but also complement and intersection.

We next considered the context-free languages showing how they can be accommodated by non-deterministic pushdown automata. We showed how deterministic pushdown automata were not sufficient to accommodate all the context-free languages.

We considered the implications of formal language theory for natural language syntax, citing several arguments that natural language must be context-free and may even be context-sensitive.

Last, we briefly reviewed two additional abstract machine types: Turing machines and finite state transducers.

6.8 Exercises

1. Write a right-linear grammar that generates the language where strings must have exactly one instance of a and any number of instances of the other symbols.

2. Write a right-linear grammar where strings must contain an a, a b, and a c in just that order, plus any number of other symbols.

3. Write a context-free grammar where legal strings are composed of some number of instances of a, followed by a c, followed by exactly the same number of instances of b as there were of a, followed by another c.

4. Write a context-sensitive grammar where legal strings are composed of some number of instances of a, followed by exactly the same number of

instances of b as there were of a, followed by exactly the same number of instances of c.

5. Write a DFA that generates the language where strings must have exactly one instance of a and any number of instances of the other symbols.

6. Write a DFA where strings must contain an a, a b, and a c in just that order, plus any number of other symbols.

7. Write a DFA where anywhere a occurs, it must be immediately followed by a b, and any number of instances of c my occur around those bits.

8. Describe this language in words: $b(a^*|c^*)c$

9. Describe this language in words: $b(a|c)^*c$

10. Describe this language in words: $(a|b|c)^*a(a|b|c)^*a(a|b|c)^*$

11. Formalize this language as a regular language: all strings contain precisely three symbols.

12. Formalize this language as a regular language: all strings contain more instances of a than of b, in any order, with no instances of c.

13. Explain why ww^R cannot be regular.

14. Explain why ww cannot be context-free.

15. Assuming the alphabet $\{a, b, e, f\}$, write a transducer which replaces any instance of a that precedes an f with an e. Otherwise, strings are unchanged.