FINITE AUTOMATA

19 February 2024 02:57 AM

FINITE AUTOMATA

DEFINITION 1.5

A ***finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set called the ***states***,
2. Σ is a finite set called the ***alphabet***,
3. $\delta\colon Q \times \Sigma \longrightarrow Q$ is the ***transition function***,[1]
4. $q_0 \in Q$ is the ***start state***, and
5. $F \subseteq Q$ is the ***set of accept states***.[2]

$\delta(x, 1) = y.$

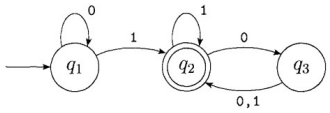


FIGURE 1.6
The finite automaton M_1

[1] Refer back to page 7 if you are uncertain about the meaning of $\delta\colon Q \times \Sigma \longrightarrow Q$.
[2] Accept states sometimes are called ***final states***.

If A is the set of all strings that machine M accepts, we say that A is the ***language of machine M*** and write $L(M) = A$. We say that ***M recognizes A*** or that ***M accepts A***. Because the term *accept* has different meanings when we refer to machines accepting strings and machines accepting languages, we prefer the term *recognize* for languages in order to avoid confusion.

A machine may accept several strings, but it always recognizes only one language. If the machine accepts no strings, it still recognizes one language—namely, the empty language $\emptyset$.

Imp point

finished reading. Note that, because the start state is also an accept state, M_3 accepts the empty string ε. As soon as a machine begins reading the empty

it reads, modulo 3. Every time it receives the $\langle \text{RESET} \rangle$ symbol it resets the count to 0. It accepts if the sum is 0, modulo 3, or in other words, if the sum is a

$\delta_i(q_j, \langle \text{RESET} \rangle) = q_0.$

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \cdots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M ***accepts*** w if a sequence of states $r_0, r_1, \ldots, r_n$ in Q exists with three conditions:

1. $r_0 = q_0$,
2. $\delta(r_i, w_{i+1}) = r_{i+1}$, for $i = 0, \ldots, n-1$, and
3. $r_n \in F$.

Condition 1 says that the machine starts in the start state. Condition 2 says that the machine goes from state to state according to the transition function. Condition 3 says that the machine accepts its input if it ends up in an accept state. We say that M ***recognizes language*** A if $A = \{w | M \text{ accepts } w\}$.

Def^n of finite automaton

Imp

DEFINITION 1.16

A language is called a ***regular language*** if some finite automaton recognizes it.

REGULAR OPERATIONS

manipulating them. We define three operations on languages, called the ***regular operations***, and use them to study properties of the regular languages.

DEFINITION 1.23

Let A and B be languages. We define the regular operations ***union***, ***concatenation***, and ***star*** as follows.

- **Union**: $A \cup B = \{x | x \in A \text{ or } x \in B\}$.
- **Concatenation**: $A \circ B = \{xy | x \in A \text{ and } y \in B\}$.
- **Star**: $A^* = \{x_1 x_2 \ldots x_k | k \geq 0 \text{ and each } x_i \in A\}$.

Imp $\varepsilon \in A^*$ irrespective of A

"any number" includes 0 as a possibility, the empty string ε is always a member of A^*, no matter what A is.

an object still in the collection. We show that the collection of regular languages is closed under all three of the regular operations. In Section 1.3 we show that

M_1 on the input and then simulates M_2 on the input. But we must be careful here! Once the symbols of the input have been read and used to simulate M_1, we can't "rewind the input tape" to try the simulation on M_2. We need another approach.

Let M_1 recognize A_1, where $M_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$, and
M_2 recognize A_2, where $M_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$.

Construct M to recognize $A_1 \cup A_2$, where $M = (Q, \Sigma, \delta, q_0, F)$.

1. $Q = \{(r_1, r_2) | \ r_1 \in Q_1 \text{ and } r_2 \in Q_2\}$.
   This set is the ***Cartesian product*** of sets Q_1 and Q_2 and is written $Q_1 \times Q_2$. It is the set of all pairs of states, the first from Q_1 and the second from Q_2.
2. Σ, the alphabet, is the same as in M_1 and M_2. In this theorem and in all subsequent similar theorems, we assume for simplicity that both M_1 and M_2 have the same input alphabet Σ. The theorem remains true if they have different alphabets, Σ_1 and Σ_2. We would then modify the proof to let $\Sigma = \Sigma_1 \cup \Sigma_2$.
3. δ, the transition function, is defined as follows. For each $(r_1, r_2) \in Q$ and each $a \in \Sigma$, let
   $$\delta((r_1, r_2), a) = (\delta_1(r_1, a), \delta_2(r_2, a)).$$
   Hence δ gets a state of M (which actually is a pair of states from M_1 and M_2), together with an input symbol, and returns M's next state.
4. q_0 is the pair (q_1, q_2).
5. F is the set of pairs in which either member is an accept state of M_1 or M_2. We can write it as
   $$F = \{(r_1, r_2) | \ r_1 \in F_1 \text{ or } r_2 \in F_2\}.$$
   This expression is the same as $F = (F_1 \times Q_2) \cup (Q_1 \times F_2)$. (Note that it is *not* the same as $F = F_1 \times F_2$. What would that give us instead?[3])

formal definition of AUB

NON DETERMINISM To prove AoB is closed we need non determinism

several choices may exist for the next state at any point.

Nondeterminism is a generalization of determinism, so every deterministic finite automaton is automatically a nondeterministic finite automaton. As Fig-

Diff ① exiting arrows

Diff ② labels have alpha. or ∈ in NFA

Diff ③ multiple copies of machine created in NFA

The difference between a deterministic finite automaton, abbreviated DFA, and a nondeterministic finite automaton, abbreviated NFA, is immediately apparent. First, every state of a DFA always has exactly one exiting transition arrow for each symbol in the alphabet. The nondeterministic automaton shown in Figure 1.27 violates that rule. State q_1 has one exiting arrow for 0, but it has two for 1; q_2 has one arrow for 0, but it has none for 1. In an NFA a state may have zero, one, or many exiting arrows for each alphabet symbol.

Second, in a DFA, labels on the transition arrows are symbols from the alphabet. This NFA has an arrow with the label ε. In general, an NFA may have arrows labeled with members of the alphabet or ε. Zero, one, or many arrows may exit from each state with the label ε.

How does an NFA compute? Suppose that we are running an NFA on an input string and come to a state with multiple ways to proceed. For example, say that we are in state q_1 in NFA N_1 and that the next input symbol is a 1. After reading that symbol, the machine splits into multiple copies of itself and follows *all* the possibilities in parallel. Each copy of the machine takes one of the possible ways to proceed and continues as before. If there are subsequent choices, the machine splits again. If the next input symbol doesn't appear on any of the arrows exiting the state occupied by a copy of the machine, that copy of the machine dies, along with the branch of the computation associated with it. Finally, if *any one* of these copies of the machine is in an accept state at the end of the input, the NFA accepts the input string.

If a state with an ε symbol on an exiting arrow is encountered, something similar happens. Without reading any input, the machine splits into multiple copies, one following each of the exiting ε-labeled arrows and one staying at the current state. Then the machine proceeds nondeterministically as before.

Nondeterminism may be viewed as a kind of parallel computation wherein multiple independent "processes" or "threads" can be running concurrently

what happens on encountering ∈ by NFA

strings that contain either 101 or 11 as a substring.

Nondeterministic finite automata are useful in several respects. As we will show, every NFA can be converted into an equivalent DFA, and constructing NFAs is sometimes easier than directly constructing DFAs. An NFA may be much

the collection of all subsets of Q. Here $\mathcal{P}(Q)$ is called the ***power set*** of Q. For any alphabet Σ we write Σ_ε to be $\Sigma \cup \{\varepsilon\}$. Now we can write the formal description of the type of the transition function in an NFA as $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$.

DEFINITION 1.37

A ***nondeterministic finite automaton*** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

1. Q is a finite set of states,
2. Σ is a finite alphabet,
3. $\delta: Q \times \Sigma_\varepsilon \longrightarrow \mathcal{P}(Q)$ is the transition function,
4. $q_0 \in Q$ is the start state, and
5. $F \subseteq Q$ is the set of accept states.

NFA

only Defn

only diff is the δ ↑ from DFA

The formal definition of computation for an NFA is similar to that for a DFA. Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and w a string over the alphabet Σ. Then we say that N ***accepts*** w if we can write w as $w = y_1 y_2 \cdots y_m$, where each y_i is a member of Σ_ε and a sequence of states $r_0, r_1, \ldots, r_m$ exists in Q with three conditions:

1. $r_0 = q_0$,
2. $r_{i+1} \in \delta(r_i, y_{i+1})$, for $i = 0, \ldots, m-1$, and
3. $r_m \in F$.

Deterministic and nondeterministic finite automata recognize the same class of languages. Such equivalence is both surprising and useful. It is surprising because NFAs appear to have more power than DFAs, so we might expect that NFAs recognize more languages. It is useful because describing an NFA for a given language sometimes is much easier than describing a DFA for that language.

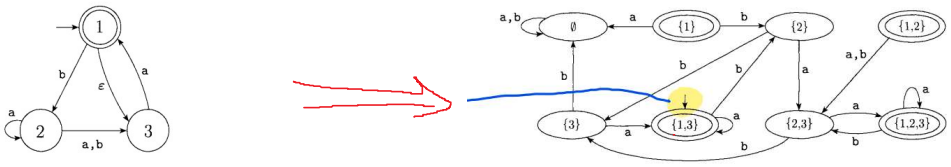
Say that two machines are ***equivalent*** if they recognize the same language.

THEOREM 1.39

Every nondeterministic finite automaton has an equivalent deterministic finite automaton.

COROLLARY 1.40

A language is regular if and only if some nondeterministic finite automaton recognizes it.



REGULAR EXPRESSIONS

Similarly, we can use the regular operations to build up expressions describing languages, which are called ***regular expressions***. An example is:

The value of the arithmetic expression is the number 32. The value of a regular expression is a language. In this case the value is the language consisting of all

$(2 + 3) \times 4$. In regular expressions, the star operation is done first, followed by concatenation, and finally union, unless parentheses are used to change the usual order.

generally, if Σ is any alphabet, the regular expression Σ describes the language consisting of all strings of length 1 over this alphabet, and Σ^* describes the language consisting of all strings over that alphabet. Similarly Σ^*1 is the language

For convenience, we let R^+ be shorthand for RR^*. In other words, whereas R^* has all strings that are 0 or more concatenations of strings from R, the language R^+ has all strings that that are 1 or more concatenations of strings from R. So $R^* \cup \varepsilon = R^*$. In addition, we let R^k be shorthand for the concatenation of k R's with each other.

When we want to distinguish between a regular expression R and the language that it describes, we write $L(R)$ to be the language of R.

FORMAL DEFINITION OF A REGULAR EXPRESSION

DEFINITION 1.52

Say that R is a ***regular expression*** if R is

1. a for some a in the alphabet Σ,
2. ε,
3. $\emptyset$,
4. $(R_1 \cup R_2)$, where R_1 and R_2 are regular expressions,
5. $(R_1 \circ R_2)$, where R_1 and R_2 are regular expressions, or
6. (R_1^*), where R_1 is a regular expression.

In items 1 and 2, the regular expressions a and ε represent the languages $\{a\}$ and $\{\varepsilon\}$, respectively. In item 3, the regular expression $\emptyset$ represents the empty language. In items 4, 5, and 6, the expressions represent the languages obtained by taking the union or concatenation of the languages R_1 and R_2, or the star of the language R_1, respectively.

Don't confuse the regular expressions ε and $\emptyset$. The expression ε represents the language containing a single string—namely, the empty string—whereas $\emptyset$ represents the language that doesn't contain any strings.

4. $1^*(01^*)^* = \{w \mid$ every 0 in w is followed by at least one $1\}$.
5. $(\Sigma\Sigma)^* = \{w \mid w$ is a string of even length$\}$.[5]
6. $(\Sigma\Sigma\Sigma)^* = \{w \mid$ the length of w is a multiple of three$\}$.

11. $1^*\emptyset = \emptyset$.
Concatenating the empty set to any set yields the empty set.
12. $\emptyset^* = \{\varepsilon\}$.
The star operation puts together any number of strings from the language to get a string in the result. If the language is empty, the star operation can put together 0 strings, giving only the empty string.

If we let R be any regular expression, we have the following identities. They are good tests of whether you understand the definition.

$R \cup \emptyset = R$.
Adding the empty language to any other language will not change it.

$R \circ \varepsilon = R$.
Joining the empty string to any string will not change it.

However, exchanging $\emptyset$ and ε in the preceding identities may cause the equalities to fail.

$R \cup \varepsilon$ may not equal R.
For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \cup \varepsilon) = \{0, \varepsilon\}$.

$R \circ \emptyset$ may not equal R.
For example, if $R = 0$, then $L(R) = \{0\}$ but $L(R \circ \emptyset) = \emptyset$.

superficially appear to be rather different. However, any regular expression can be converted into a finite automaton that recognizes the language it describes, and vice versa. Recall that a regular language is one that is recognized by some finite automaton.

THEOREM 1.54

A language is regular if and only if some regular expression describes it.

for proving whether a language is regular we have to prove 6 points

GNFA

We break this procedure into two parts, using a new type of finite automaton called a ***generalized nondeterministic finite automaton***, GNFA. First we show how to convert DFAs into GNFAs, and then GNFAs into regular expressions.

Generalized nondeterministic finite automata are simply nondeterministic finite automata wherein the transition arrows may have any regular expressions as labels, instead of only members of the alphabet or ε. The GNFA reads blocks of symbols from the input, not necessarily just one symbol at a time as in an ordinary NFA. The GNFA moves along a transition arrow connecting two states by reading a block of symbols from the input, which themselves constitute a string described by the regular expression on that arrow. A GNFA is nondeterministic and so may have several different ways to process the same input string. It accepts its input if its processing can cause the GNFA to be in an accept state at the end of the input. The following figure presents an example of a GNFA.

prop of GNFA

For convenience we require that GNFAs always have a special form that meets the following conditions.

- The start state has transition arrows going to every other state but no arrows coming in from any other state.
- There is only a single accept state, and it has arrows coming in from every other state but no arrows going to any other state. Furthermore, the accept state is not the same as the start state.
- Except for the start and accept states, one arrow goes from every state to every other state and also from each state to itself.

- It reads blocks of symbols from the input not necessarily a single symbols
- Transition arrows may have regular exp as labels instead of alphabets or ϵ

We can easily convert a DFA into a GNFA in the special form. We simply add a new start state with an ε arrow to the old start state and a new accept state with ε arrows from the old accept states. If any arrows have multiple labels (or if there are multiple arrows going between the same two states in the same direction), we replace each with a single arrow whose label is the union of the previous labels. Finally, we add arrows labeled $\emptyset$ between states that had no arrows. This last step won't change the language recognized because a transition labeled with $\emptyset$ can never be used. From here on we assume that all GNFAs are in the special form.

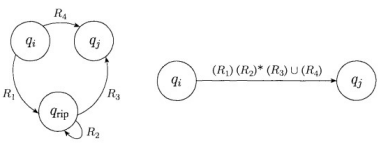
method to convert NFA to GNFA

Now we show how to convert a GNFA into a regular expression. Say that the GNFA has k states. Then, because a GNFA must have a start and an accept state and they must be different from each other, we know that $k \geq 2$. If $k > 2$, we construct an equivalent GNFA with $k - 1$ states. This step can be repeated on the new GNFA until it is reduced to two states. If $k = 2$, the GNFA has a single arrow that goes from the start state to the accept state. The label of this arrow is the equivalent regular expression. For example, the stages in converting a DFA with three states to an equivalent regular expression are shown in the following figure.

Convert GNFA to regular expression

The crucial step is in constructing an equivalent GNFA with one fewer state when $k > 2$. We do so by selecting a state, ripping it out of the machine, and repairing the remainder so that the same language is still recognized. Any state will do, provided that it is not the start or accept state. We are guaranteed that such a state will exist because $k > 2$. Let's call the removed state q_{rip}.

After removing q_{rip} we repair the machine by altering the regular expressions that label each of the remaining arrows. The new labels compensate for the absence of q_{rip} by adding back the lost computations. The new label going from a state q_i to a state q_j is a regular expression that describes all strings that would take the machine from q_i to q_j either directly or via q_{rip}. We illustrate this approach in Figure 1.63.



before after

imp

DEFINITION 1.64

A ***generalized nondeterministic finite automaton*** is a 5-tuple, $(Q, \Sigma, \delta, q_{\text{start}}, q_{\text{accept}})$, where

1. Q is the finite set of states,
2. Σ is the input alphabet,
3. $\delta: (Q - \{q_{\text{accept}}\}) \times (Q - \{q_{\text{start}}\}) \longrightarrow \mathcal{R}$ is the transition function,
4. q_{start} is the start state, and
5. q_{accept} is the accept state.

The symbol $\mathcal{R}$ is the collection of all regular expressions over the alphabet Σ,

formal definition

A GNFA accepts a string w in Σ^* if $w = w_1 w_2 \cdots w_k$, where each w_i is in Σ^* and a sequence of states $q_0, q_1, \ldots, q_k$ exists such that

1. $q_0 = q_{\text{start}}$ is the start state,
2. $q_k = q_{\text{accept}}$ is the accept state, and
3. for each i, we have $w_i \in L(R_i)$, where $R_i = \delta(q_{i-1}, q_i)$; in other words, R_i is the expression on the arrow from q_{i-1} to q_i.

Def of CONVERT(G)

CONVERT(G):

1. Let k be the number of states of G.
2. If $k = 2$, then G must consist of a start state, an accept state, and a single arrow connecting them and labeled with a regular expression R. Return the expression R.
3. If $k > 2$, we select any state $q_{\text{rip}} \in Q$ different from q_{start} and q_{accept} and let G' be the GNFA $(Q', \Sigma, \delta', q_{\text{start}}, q_{\text{accept}})$, where
$$Q' = Q - \{q_{\text{rip}}\},$$
and for any $q_i \in Q' - \{q_{\text{accept}}\}$ and any $q_j \in Q' - \{q_{\text{start}}\}$ let
$$\delta'(q_i, q_j) = (R_1)(R_2)^*(R_3) \cup (R_4),$$
for $R_1 = \delta(q_i, q_{\text{rip}})$, $R_2 = \delta(q_{\text{rip}}, q_{\text{rip}})$, $R_3 = \delta(q_{\text{rip}}, q_j)$, and $R_4 = \delta(q_i, q_j)$.
4. Compute CONVERT(G') and return this value.

For any GNFA G, CONVERT(G) is equivalent to G.

NON REGULAR LANGUAGES

PUMPING LEMMA

not have this property, we are guaranteed that it is not regular. The property states that all strings in the language can be "pumped" if they are at least as long as a certain special value, called the ***pumping length***. That means each

THEOREM 1.70

Pumping lemma If A is a regular language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into three pieces, $s = xyz$, satisfying the following conditions:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$, and
3. $|xy| \leq p$.

An alternative method of proving that C is nonregular follows from our knowledge that B is nonregular. If C were regular, $C \cap 0^*1^*$ also would be regular. The reasons are that the language 0^*1^* is regular and that the class of regular languages is closed under intersection, which we proved in footnote 3 (page 46). But $C \cap 0^*1^*$ equals B, and we know that B is nonregular from Example 1.73.

Comes in handy when trying to prove other languages from languages we know

EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} \mid n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z. These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular.

1.14 **a.** Show that, if M is a DFA that recognizes language B, swapping the accept and nonaccept states in M yields a new DFA that recognizes the complement of B. Conclude that the class of regular languages is closed under complement.

b. Show by giving an example that, if M is an NFA that recognizes language C, swapping the accept and nonaccept states in M doesn't necessarily yield a new NFA that recognizes the complement of C. Is the class of languages recognized by NFAs closed under complement? Explain your answer.

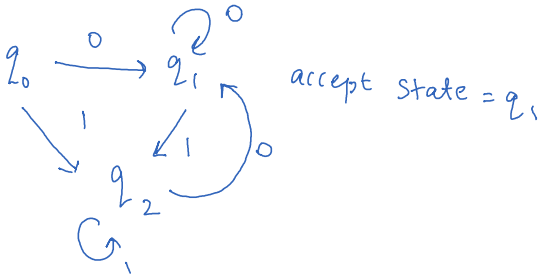
A string w in the new DFA reaches its accept state then it means that it wouldn't have reached in M. therefore it belongs to the complement lagn

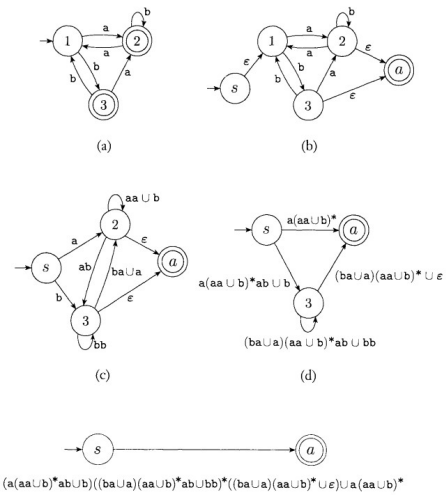
2. Multiple paths:

Unlike DFAs with a single defined path for each string, NFAs can have multiple paths from the start state to any end state, including accept and non-accept states. This creates uncertainty about which path a string will take depending on the non-deterministic choices made.

3. Swapping states:

When we swap accept and non-accept states in an NFA, we change the ending points for valid paths. However, due to non-determinism, some strings might still reach the original accept states through other paths, even though they shouldn't be accepted after the swap.





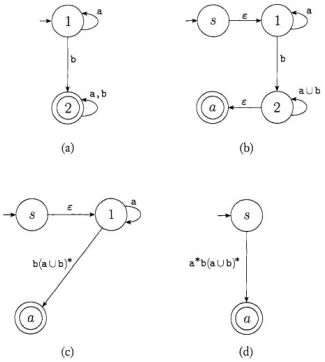


FIGURE **1.67**
Converting a two-state DFA to an equivalent regular expression

Conv to gnfa

CONTEXT FREE LANGUAGES

19 February 2024 06:02 PM

The collection of languages associated with context-free grammars are called the ***context-free languages***. They include all the regular languages and many additional languages. In this chapter, we give a formal definition of context-free

$$A \to 0A1$$
$$A \to B$$
$$B \to \#$$

A grammar consists of a collection of ***substitution rules***, also called ***productions***. Each rule appears as a line in the grammar, comprising a symbol and a string separated by an arrow. The symbol is called a ***variable***. The string consists of variables and other symbols called ***terminals***. The variable symbols often are represented by capital letters. The terminals are analogous to the input alphabet and often are represented by lowercase letters, numbers, or special symbols. One variable is designated as the ***start variable***. It usually occurs on the left-hand side of the topmost rule. For example, grammar G_1 contains three

All strings generated in this way constitute the ***language of the grammar***. We write $L(G_1)$ for the language of grammar G_1. Some experimentation with the grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$. Any language that can be generated by some context-free grammar is called a ***context-free language*** (CFL). For convenience when presenting a context-free grammar, we abbreviate

FORMAL DEFINITION OF A CONTEXT-FREE GRAMMAR

Let's formalize our notion of a context-free grammar (CFG).

DEFINITION 2.2

A ***context-free grammar*** is a 4-tuple (V, Σ, R, S), where

1. V is a finite set called the ***variables***,
2. Σ is a finite set, disjoint from V, called the ***terminals***,
3. R is a finite set of ***rules***, with each rule being a variable and a string of variables and terminals, and
4. $S \in V$ is the start variable.

If u, v, and w are strings of variables and terminals, and $A \to w$ is a rule of the grammar, we say that uAv ***yields*** uwv, written $uAv \Rightarrow uwv$. Say that u ***derives*** v, written $u \stackrel{*}{\Rightarrow} v$, if $u = v$ or if a sequence $u_1, u_2, \ldots, u_k$ exists for $k \geq 0$ and

$$u \Rightarrow u_1 \Rightarrow u_2 \Rightarrow \ldots \Rightarrow u_k \Rightarrow v.$$

The ***language of the grammar*** is $\{w \in \Sigma^* \mid S \stackrel{*}{\Rightarrow} w\}$.

For example, grammar G_1 generates the string 000#111. The sequence of substitutions to obtain a string is called a ***derivation***. A derivation of string 000#111 in grammar G_1 is

$$A \Rightarrow 0A1 \Rightarrow 00A11 \Rightarrow 000A111 \Rightarrow 000B111 \Rightarrow 000\#111$$

You may also represent the same information pictorially with a ***parse tree***. An example of a parse tree is shown in Figure 2.1.

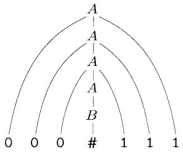


FIGURE 2.1
Parse tree for 000#111 in grammar G_1

grammar G_1 shows us that $L(G_1)$ is $\{0^n\#1^n \mid n \geq 0\}$

DESIGNING CONTEXT-FREE GRAMMARS

Constructing a CFG from CFL (1)

Break the CFL into smaller CFL & take their union.

As with the design of finite automata, discussed in Section 1.1 (page 41), the design of context-free grammars requires creativity. Indeed, context-free grammars are even trickier to construct than finite automata because we are more accustomed to programming a machine for specific tasks than we are to describing languages with grammars. The following techniques are helpful, singly or in combination, when you're faced with the problem of constructing a CFG.

First, many CFLs are the union of simpler CFLs. If you must construct a CFG for a CFL that you can break into simpler pieces, do so and then construct individual grammars for each piece. These individual grammars can be easily merged into a grammar for the original language by combining their rules and then adding the new rule $S \to S_1 \mid S_2 \mid \cdots \mid S_k$, where the variables S_i are the start variables for the individual grammars. Solving several simpler problems is often easier than solving one complicated problem.

For example, to get a grammar for the language $\{0^n1^n \mid n \geq 0\} \cup \{1^n0^n \mid n \geq 0\}$, first construct the grammar

$$S_1 \to 0S_11 \mid \varepsilon$$

for the language $\{0^n1^n \mid n \geq 0\}$ and the grammar

$$S_2 \to 1S_20 \mid \varepsilon$$

for the language $\{1^n0^n \mid n \geq 0\}$ and then add the rule $S \to S_1 \mid S_2$ to give the grammar

$$S \to S_1 \mid S_2$$
$$S_1 \to 0S_11 \mid \varepsilon$$
$$S_2 \to 1S_20 \mid \varepsilon.$$

(2) for a CFL which is regular create a DFA for the lang & then convert DFA → CFG

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \to aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \to \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

(3) if string needs to be remembered then try breaking it down to R → uRv type

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \to uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure

(2) for a CFL which is regular create a DFA for the lang & then convert DFA → CFG

Second, constructing a CFG for a language that happens to be regular is easy if you can first construct a DFA for that language. You can convert any DFA into an equivalent CFG as follows. Make a variable R_i for each state q_i of the DFA. Add the rule $R_i \rightarrow aR_j$ to the CFG if $\delta(q_i, a) = q_j$ is a transition in the DFA. Add the rule $R_i \rightarrow \varepsilon$ if q_i is an accept state of the DFA. Make R_0 the start variable of the grammar, where q_0 is the start state of the machine. Verify on your own that the resulting CFG generates the same language that the DFA recognizes.

(3) if string needs to be remembered then try breaking it down to R → uRv type

Third, certain context-free languages contain strings with two substrings that are "linked" in the sense that a machine for such a language would need to remember an unbounded amount of information about one of the substrings to verify that it corresponds properly to the other substring. This situation occurs in the language $\{0^n1^n \mid n \geq 0\}$ because a machine would need to remember the number of 0s in order to verify that it equals the number of 1s. You can construct a CFG to handle this situation by using a rule of the form $R \rightarrow uRv$, which generates strings wherein the portion containing the u's corresponds to the portion containing the v's.

Finally, in more complex languages, the strings may contain certain structures that appear recursively as part of other (or the same) structures. That situation occurs in the grammar that generates arithmetic expressions in Example 2.4. Any time the symbol a appears, an entire parenthesized expression might appear recursively instead. To achieve this effect, place the variable symbol generating the structure in the location of the rules corresponding to where that structure may recursively appear.

If a grammar generates the same string in several different ways, we say that the string is derived *ambiguously* in that grammar. If a grammar generates some string ambiguously we say that the grammar is *ambiguous*.

Now we formalize the notion of ambiguity. When we say that a grammar generates a string ambiguously, we mean that the string has two different parse trees, not two different derivations. Two derivations may differ merely in the order in which they replace variables yet not in their overall structure. To concentrate on structure we define a type of derivation that replaces variables in a fixed order. A derivation of a string w in a grammar G is a ***leftmost derivation*** if at every step the leftmost remaining variable is the one replaced. The derivation preceding Definition 2.2 (page 102) is a leftmost derivation.

DEFINITION 2.7

A string w is derived ***ambiguously*** in context-free grammar G if it has two or more different leftmost derivations. Grammar G is ***ambiguous*** if it generates some string ambiguously.

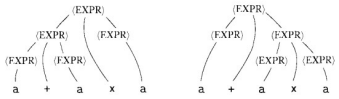


FIGURE 2.6
The two parse trees for the string a+a×a in grammar G_5

G_5 is ambiguous.

learn eg

Sometimes when we have an ambiguous grammar we can find an unambiguous grammar that generates the same language. Some context-free languages, however, can be generated only by ambiguous grammars. Such languages are called ***inherently ambiguous***. Problem 2.29 asks you to prove that the language $\{a^ib^jc^k \mid i = j \text{ or } j = k\}$ is inherently ambiguous.

CHOMSKY NORMAL FORM

DEFINITION 2.8

A context-free grammar is in ***Chomsky normal form*** if every rule is of the form

$$A \rightarrow BC$$
$$A \rightarrow a$$

where a is any terminal and A, B, and C are any variables—except that B and C may not be the start variable. In addition we permit the rule $S \rightarrow \varepsilon$, where S is the start variable.

THEOREM 2.9

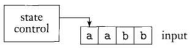
Any context-free language is generated by a context-free grammar in Chomsky normal form.

PUSHDOWN AUTOMATA

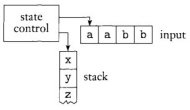
In this section we introduce a new type of computational model called ***pushdown automata***. These automata are like nondeterministic finite automata but have an extra component called a ***stack***. The stack provides additional memory beyond the finite amount available in the control. The stack allows pushdown automata to recognize some nonregular languages.

Pushdown automata are equivalent in power to context-free grammars. This equivalence is useful because it gives us two options for proving that a language is context free. We can give either a context-free grammar generating it or a pushdown automaton recognizing it. Certain languages are more easily described in

The following figure is a schematic representation of a finite automaton. The control represents the states and transition function, the tape contains the input string, and the arrow represents the input head, pointing at the next input symbol to be read.



With the addition of a stack component we obtain a schematic representation of a pushdown automaton, as shown in the following figure.



A pushdown automaton (PDA) can write symbols on the stack and read them back later. Writing a symbol "pushes down" all the other symbols on the stack. At any time the symbol on the top of the stack can be read and removed. The remaining symbols then move back up. Writing a symbol on the stack is often referred to as ***pushing*** the symbol, and removing a symbol is referred to as ***popping*** it. Note that all access to the stack, for both reading and writing, may be done only at the top. In other words a stack is a "last in, first out" storage

Nondeterministic pushdown automata recognize certain languages which no deterministic pushdown automata can recognize, though we will not prove this

Recall that deterministic and nondeterministic finite automata do recognize the same class of languages, so the pushdown automata situation is different. We fo-

FORMAL DEFINITION OF A PUSHDOWN AUTOMATON

input alphabet Σ and a stack alphabet Γ.

$\Sigma_\varepsilon = \Sigma \cup \{\varepsilon\}$ and $\Gamma_\varepsilon = \Gamma \cup \{\varepsilon\}$.

The domain of the transition function is $Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon$. Thus the current state, next input symbol read, and top symbol of the stack determine the next move of a pushdown automaton. Either symbol may be ε, causing the machine to move

$\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$.

DEFINITION 2.13

A ***pushdown automaton*** is a 6-tuple $(Q, \Sigma, \Gamma, \delta, q_0, F)$, where Q, Σ, Γ, and F are all finite sets, and

1. Q is the set of states,
2. Σ is the input alphabet,
3. Γ is the stack alphabet,
4. $\delta: Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \longrightarrow \mathcal{P}(Q \times \Gamma_\varepsilon)$ is the transition function,
5. $q_0 \in Q$ is the start state, and
6. $F \subseteq Q$ is the set of accept states.

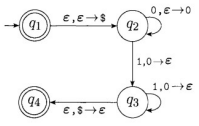
A pushdown automaton $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ computes as follows. It accepts input w if w can be written as $w = w_1 w_2 \cdots w_m$, where each $w_i \in \Sigma_\varepsilon$ and sequences of states $r_0, r_1, \ldots, r_m \in Q$ and strings $s_0, s_1, \ldots, s_m \in \Gamma^*$ exist that satisfy the following three conditions. The strings s_i represent the sequence of stack contents that M has on the accepting branch of the computation.

1. $r_0 = q_0$ and $s_0 = \varepsilon$. This condition signifies that M starts out properly, in the start state and with an empty stack.
2. For $i = 0, \ldots, m-1$, we have $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, where $s_i = at$ and $s_{i+1} = bt$ for some $a, b \in \Gamma_\varepsilon$ and $t \in \Gamma^*$. This condition states that M moves properly according to the state, stack, and next input symbol.
3. $r_m \in F$. This condition states that an accept state occurs at the input end.

going from state to state. We write "$a,b \to c$" to signify that when the machine is reading an a from the input it may replace the symbol b on the top of the stack with a c. Any of a, b, and c may be ε. If a is ε, the machine may make this transition without reading any symbol from the input. If b is ε, the machine may make this transition without reading and popping any symbol from the stack. If c is ε, the machine does not write any symbol on the stack when going along this transition.

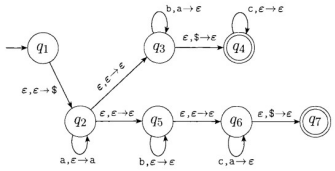
$\varepsilon, b \to c$ replace b with c
$a, \varepsilon \to c$ push c or just transition
$a, b \to \varepsilon$ pop

The formal definition of a PDA contains no explicit mechanism to allow the PDA to test for an empty stack. This PDA is able to get the same effect by initially placing a special symbol \$ on the stack. Then if it ever sees the \$ again, it knows that the stack effectively is empty. Subsequently, when we refer to testing for an



deterministic PDA

FIGURE 2.15
State diagram for the PDA M_1 that recognizes $\{0^n 1^n \mid n \geq 0\}$



Non deterministic PDA

FIGURE 2.17
State diagram for PDA M_2 that recognizes $\{a^i b^j c^k \mid i, j, k \geq 0 \text{ and } i = j \text{ or } i = k\}$

$w^{\mathcal{R}}$ means w written backwards.

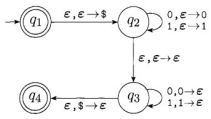


FIGURE 2.19
State diagram for the PDA M_3 that recognizes $\{ww^{\mathcal{R}} \mid w \in \{0,1\}^*\}$

In this section we show that context-free grammars and pushdown automata are equivalent in power. Both are capable of describing the class of context-free languages. We show how to convert any context-free grammar into a pushdown

THEOREM 2.20

A language is context free if and only if some pushdown automaton recognizes it.

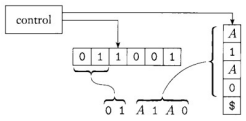
PROOF IDEA Let A be a CFL. From the definition we know that A has a CFG, G, generating it. We show how to convert G into an equivalent PDA, which we call P.

One of the difficulties in testing whether there is a derivation for w is in figuring out which substitutions to make. The PDA's nondeterminism allows it to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any

The following is an informal description of P.

symbol on the stack and that may be a terminal symbol instead of a variable. The way around this problem is to keep only *part* of the intermediate string on the stack: the symbols starting with the first variable in the intermediate string. Any terminal symbols appearing before the first variable are matched immediately with symbols in the input string. The following figure shows the PDA P.



Let q and r be states of the PDA and let a be in Σ_ε and s be in Γ_ε. Say that we want the PDA to go from q to r when it reads a and pops s. Furthermore we want it to push the entire string $u = u_1 \cdots u_l$ on the stack at the same time. We can implement this action by introducing new states $q_1, \ldots, q_{l-1}$ and setting the transition function as follows

$$\begin{aligned}
&\delta(q, a, s) \text{ to contain } (q_1, u_l),\\
&\delta(q_1, \varepsilon, \varepsilon) = \{(q_2, u_{l-1})\},\\
&\delta(q_2, \varepsilon, \varepsilon) = \{(q_3, u_{l-2})\},\\
&\qquad\vdots\\
&\delta(q_{l-1}, \varepsilon, \varepsilon) = \{(r, u_1)\}.
\end{aligned}$$

We use the notation $(r, u) \in \delta(q, a, s)$ to mean that when q is the state of the automaton, a is the next input symbol, and s is the symbol on the top of the stack, the PDA may read the a and pop the s, then push the string u onto the stack and go on to the state r. The following figure shows this implementation.

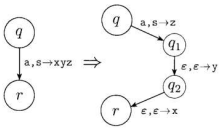


FIGURE 2.23
Implementing the shorthand $(r, xyz) \in \delta(q, a, s)$

LEMMA 2.27

If a pushdown automaton recognizes some language, then it is context free.

Method to convert pda to cfg

PROOF IDEA We have a PDA P, and we want to make a CFG G that generates all the strings that P accepts. In other words, G should generate a string if that string causes the PDA to go from its start state to an accept state.

To achieve this outcome we design a grammar that does somewhat more. For each pair of states p and q in P the grammar will have a variable A_{pq}. This variable generates all the strings that can take P from p with an empty stack to q with an empty stack. Observe that such strings can also take P from p to q, regardless of the stack contents at p, leaving the stack at q in the same condition as it was at p.

First, we simplify our task by modifying P slightly to give it the following three features.

1. It has a single accept state, q_{accept}.
2. It empties its stack before accepting.
3. Each transition either pushes a symbol onto the stack (a *push* move) or pops one off the stack (a *pop* move), but it does not do both at the same time.

Giving P features 1 and 2 is easy. To give it feature 3, we replace each transition that simultaneously pops and pushes with a two transition sequence that goes through a new state, and we replace each transition that neither pops nor pushes with a two transition sequence that pushes then pops an arbitrary stack symbol.

To design G so that A_{pq} generates all strings that take P from p to q, starting and ending with an empty stack, we must understand how P operates on these strings. For any such string x, P's first move on x must be a push, because every move is either a push or a pop and P can't pop an empty stack. Similarly, the last move on x must be a pop, because the stack ends up empty.

Two possibilities occur during P's computation on x. Either the symbol popped at the end is the symbol that was pushed at the beginning, or not. If so, the stack is empty only at the beginning and end of P's computation on x. If not, the initially pushed symbol must get popped at some point before the end of x and thus the stack becomes empty at this point. We simulate the former possibility with the rule $A_{pq} \to aA_{rs}b$, where a is the input read at the first move, b is the input read at the last move, r is the state following p, and s is the state preceding q. We simulate the latter possibility with the rule $A_{pq} \to A_{pr}A_{rq}$, where r is the state when the stack becomes empty.

CLAIM 2.30

If A_{pq} generates x, then x can bring P from p with empty stack to q with empty stack.

CLAIM 2.31

If x can bring P from p with empty stack to q with empty stack, A_{pq} generates x.

COROLLARY 2.32

Every regular language is context free.

to guess the sequence of correct substitutions. At each step of the derivation one of the rules for a particular variable is selected nondeterministically and used to substitute for that variable.

The following is an informal description of P.

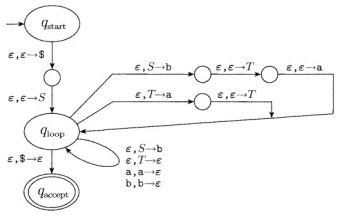
1. Place the marker symbol $ and the start variable on the stack.
2. Repeat the following steps forever.
   a. If the top of stack is a variable symbol A, nondeterministically select one of the rules for A and substitute A by the string on the right-hand side of the rule.
   b. If the top of stack is a terminal symbol a, read the next symbol from the input and compare it to a. If they match, repeat. If they do not match, reject on this branch of the nondeterminism.
   c. If the top of stack is the symbol $, enter the accept state. Doing so accepts the input if it has all been read.

EXAMPLE 2.25

We use the procedure developed in Lemma 2.21 to construct a PDA P_1 from the following CFG G.

$$\begin{aligned}
S &\to \mathtt{a}T\mathtt{b} \mid \mathtt{b}\\
T &\to T\mathtt{a} \mid \varepsilon
\end{aligned}$$

The transition function is shown in the following diagram.



Rules:

The states of P are $Q = \{q_{\text{start}}, q_{\text{loop}}, q_{\text{accept}}\} \cup E$, where E is the set of states we need for implementing the shorthand just described. The start state is q_{start}. The only accept state is q_{accept}.

The transition function is defined as follows. We begin by initializing the stack to contain the symbols $ and S, implementing step 1 in the informal description: $\delta(q_{\text{start}}, \varepsilon, \varepsilon) = \{(q_{\text{loop}}, S\$)\}$. Then we put in transitions for the main loop of step 2.

First, we handle case (a) wherein the top of the stack contains a variable. Let $\delta(q_{\text{loop}}, \varepsilon, A) = \{(q_{\text{loop}}, w) | \text{ where } A \to w \text{ is a rule in } R\}$.

Second, we handle case (b) wherein the top of the stack contains a terminal. Let $\delta(q_{\text{loop}}, a, a) = \{(q_{\text{loop}}, \varepsilon)\}$.

Finally, we handle case (c) wherein the empty stack marker $ is on the top of the stack. Let $\delta(q_{\text{loop}}, \varepsilon, \$) = \{(q_{\text{accept}}, \varepsilon)\}$.

The state diagram is shown in Figure 2.24

THE PUMPING LEMMA FOR CONTEXT-FREE LANGUAGES

THEOREM 2.34

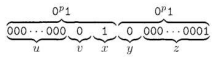
Pumping lemma for context-free languages If A is a context-free language, then there is a number p (the pumping length) where, if s is any string in A of length at least p, then s may be divided into five pieces $s = uvxyz$ satisfying the conditions

1. for each $i \geq 0$, $uv^ixy^iz \in A$,
2. $|vy| > 0$, and
3. $|vxy| \leq p$.

EXAMPLE 2.38

Let $D = \{ww \mid w \in \{0,1\}^*\}$. Use the pumping lemma to show that D is not a CFL. Assume that D is a CFL and obtain a contradiction. Let p be the pumping length given by the pumping lemma.

This time choosing string s is less obvious. One possibility is the string 0^p10^p1. It is a member of D and has length greater than p, so it appears to be a good candidate. But this string *can* be pumped by dividing it as follows, so it is not adequate for our purposes.



Let's try another candidate for s. Intuitively, the string $0^p1^p0^p1^p$ seems to capture more of the "essence" of the language D than the previous candidate did. In fact, we can show that this string does work, as follows.

We show that the string $s = 0^p1^p0^p1^p$ cannot be pumped. This time we use condition 3 of the pumping lemma to restrict the way that s can be divided. It says that we can pump s by dividing $s = uvxyz$, where $|vxy| \leq p$.

First, we show that the substring vxy must straddle the midpoint of s. Otherwise, if the substring occurs only in the first half of s, pumping s up to uv^2xy^2z moves a 1 into the first position of the second half, and so it cannot be of the form ww. Similarly, if vxy occurs in the second half of s, pumping s up to uv^2xy^2z moves a 0 into the last position of the first half, and so it cannot be of the form ww.

But if the substring vxy straddles the midpoint of s, when we try to pump s down to uxz it has the form $0^p1^i0^j1^p$, where i and j cannot both be p. This string is not of the form ww. Thus s cannot be pumped, and D is not a CFL.

CHOMSKY NORMAL FORM

- A non-terminal generating a terminal (e.g.; X->x)
- A non-terminal generating two non-terminals (e.g.; X->YZ)
- Start symbol generating ε. (e.g.; S-> ε)

```
S → ASB
A → aAS|a|ε
B → SbS|A|bb
```

Step 1. As start symbol S appears on the RHS, we will create a new production rule S0->S. Therefore, the grammar will become:

```
S0->S
S → ASB
A → aAS|a|ε
B → SbS|A|bb
```

Step 2. As grammar contains null production A-> ε, its removal from the grammar yields:

```
S0->S
S → ASB|SB
A → aAS|aS|a
B → SbS| A|ε|bb
```

Now, it creates null production B→ ε, its removal from the grammar yields:

```
S0->S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS| A|bb
```

Now, it creates unit production B->A, its removal from the grammar yields:

```
S0->S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Also, removal of unit production S0->S from grammar yields:

```
S0-> AS|ASB| SB| S
S → AS|ASB| SB| S
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Also, removal of unit production S->S and S0->S from grammar yields:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → aAS|aS|a
B → SbS|bb|aAS|aS|a
```

Step 3. In production rule A->aAS |aS and B-> SbS|aAS|aS, terminals a and b exist on RHS with non-terminates. Removing them from RHS:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|bb|XAS|XS|a
X →a
Y→b
```

Also, B->bb can't be part of CNF, removing it from grammar yields:

```
S0-> AS|ASB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
```

Step 4: In production rule S0->ASB, RHS has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|ASB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
```

Similarly, S->ASB has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → XAS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
```

Similarly, A->XAS has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → SYS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
```

Similarly, B->SYS has more than two symbols, removing it from grammar yields:

```
S0 -> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → TS|VV|XAS|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
T → SY
```

Similarly, B->XAX has more than two symbols, removing it from grammar yields:

```
S0-> AS|PB| SB
S → AS|QB| SB
A → RS|XS|a
B → TS|VV|US|XS|a
X → a
Y → b
V → b
P → AS
Q → AS
R → XA
T → SY
U → XA
```

resulting grammar G'. This grammar is supposed to generate A^*.

2.16 Show that the class of context-free languages is closed under the regular operations, union, concatenation, and star.

Check-in 4.1

Which of these are valid CFGs?

C_1: $B \to 0B1 \mid \varepsilon$, $B1 \to 1B$, $0B \to 0B$

C_2: $S \to 0S \mid S1$, $R \to RR$

a) C_1 only
b) C_2 only
c) Both C_1 and C_2
d) Neither

MIT OCW

$L(M) = \emptyset$

Terminals not allowed in lhs

Check-in 4.2

How many reasonable distinct meanings does the following English sentence have?

The boy saw the girl with the mirror.

(a) 1
(b) 2
(c) 3 or more

G_2

$E \to E+T \mid T$
$T \to T\times F \mid F$
$F \to (E) \mid a$

G_3

$E \to E+E \mid E\times E \mid (E) \mid a$

Both G_2 and G_3 recognize the same language, i.e., $L(G_2) = L(G_3)$.
However G_2 is an unambiguous CFG and G_3 is ambiguous.

Check-in 4.3

Is every Regular Language also a Context Free Language?

(a) Yes
(b) No
(c) Not sure

Check-in 5.1

Let $A_1 = \{0^k 1^k 2^l \mid k, l \geq 0\}$ (equal #s of 0s and 1s)
Let $A_2 = \{0^l 1^k 2^k \mid k, l \geq 0\}$ (equal #s of 1s and 2s)
Observe that PDAs can recognize A_1 and A_2. What can we now conclude?

a) The class of CFLs is not closed under intersection.
b) The Pumping Lemma shows that $A_1 \cup A_2$ is not a CFL .
c) The class of CFLs is closed under complement.

Check-in 5.2

How do we get the effect of "crossing off" with a Turing machine?

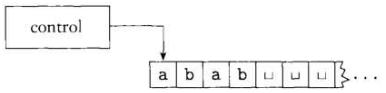
a) We add that feature to the model.
b) We use a tape alphabet $\Gamma = \{a, b, c, \tilde{a}, \tilde{b}, \tilde{c}, \sqcup\}$.
c) All Turing machines come with an eraser.

TURING MACHINES

20 February 2024 11:35 PM

The Turing machine model uses an infinite tape as its unlimited memory. It has a tape head that can read and write symbols and move around on the tape.

Initially the tape contains only the input string and is blank everywhere else. If the machine needs to store information, it may write this information on the tape. To read the information that it has written, the machine can move its head back over it. The machine continues computing until it decides to produce an output. The outputs *accept* and *reject* are obtained by entering designated accepting and rejecting states. If it doesn't enter an accepting or a rejecting state, it will go on forever, never halting.



, δ takes the form: $Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$

machine is in a certain state q and the head is over a tape square containing a symbol a, and if $\delta(q, a) = (r, b, \mathrm{L})$, the machine writes the symbol b replacing the a, and goes to state r. The third component is either L or R and indicates whether the head moves to the left or right after writing. In this case the L indicates a move to the left.

DEFINITION 3.3

A ***Turing machine*** is a 7-tuple, $(Q, \Sigma, \Gamma, \delta, q_0, q_{\text{accept}}, q_{\text{reject}})$, where Q, Σ, Γ are all finite sets and

1. Q is the set of states,
2. Σ is the input alphabet not containing the ***blank symbol*** $\sqcup$,
3. Γ is the tape alphabet, where $\sqcup \in \Gamma$ and $\Sigma \subseteq \Gamma$,
4. $\delta\colon Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$ is the transition function,
5. $q_0 \in Q$ is the start state,
6. $q_{\text{accept}} \in Q$ is the accept state, and
7. $q_{\text{reject}} \in Q$ is the reject state, where $q_{\text{reject}} \neq q_{\text{accept}}$.

by the transition function. If M ever tries to move its head to the left off the left-hand end of the tape, the head stays in the same place for that move, even though the transition function indicates L. The computation continues until it M goes on forever.

As a Turing machine computes, changes occur in the current state, the current tape contents, and the current head location. A setting of these three items is called a ***configuration*** of the Turing machine. Configurations often are represented in a special way. For a state q and two strings u and v over the tape alphabet Γ we write $u\,q\,v$ for the configuration where the current state is q, the current tape contents is uv, and the current head location is the first symbol of v. The tape contains only blanks following the last symbol of v. For example, $\mathtt{1011}q_7\mathtt{01111}$ represents the configuration when the tape is `101101111`, the current state is q_7, and the head is currently on the second `0`. The following figure depicts a Turing machine with that configuration.

chine computes. Say that configuration C_1 ***yields*** configuration C_2 if the Turing machine can legally go from C_1 to C_2 in a single step. We define this notion formally as follows.

Suppose that we have a, b, and c in Γ, as well as u and v in Γ^* and states q_i and q_j. In that case $ua\,q_i\,bv$ and $u\,q_j\,acv$ are two configurations. Say that

$$ua\,q_i\,bv \quad \text{yields} \quad u\,q_j\,acv$$

if in the transition function $\delta(q_i, b) = (q_j, c, \mathrm{L})$. That handles the case where the Turing machine moves leftward. For a rightward move, say that

$$ua\,q_i\,bv \quad \text{yields} \quad uac\,q_j\,v$$

if $\delta(q_i, b) = (q_j, c, \mathrm{R})$.

The ***start configuration*** of M on input w is the configuration $q_0\,w$, which indicates that the machine is in the start state q_0 with its head at the leftmost position on the tape. In an ***accepting configuration*** the state of the configuration is q_{accept}. In a ***rejecting configuration*** the state of the configuration is q_{reject}. Accepting and rejecting configurations are ***halting configurations*** and do not yield further configurations. Because the machine is defined to halt when in the states q_{accept} and q_{reject}, we equivalently could have defined the transition function to have the more complicated form $\delta\colon Q' \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\}$, where Q' is Q without q_{accept} and q_{reject}. A Turing machine M ***accepts*** input w if a sequence of configurations $C_1, C_2, \ldots, C_k$ exists, where

1. C_1 is the start configuration of M on input w,
2. each C_i yields C_{i+1}, and
3. C_k is an accepting configuration.

The collection of strings that M accepts is ***the language of M***, or ***the language recognized by M***, denoted $L(M)$.

DEFINITION 3.5

Call a language ***Turing-recognizable*** if some Turing machine recognizes it.[1]

A Turing machine M can fail to accept an input by entering the q_{reject} state and rejecting, or by looping. Sometimes distinguishing a machine that is looping from one that is merely taking a long time is difficult. For this reason we prefer Turing machines that halt on all inputs; such machines never loop. These machines are called ***deciders*** because they always make a decision to accept or reject. A decider that recognizes some language also is said to ***decide*** that language.

DEFINITION 3.6

Call a language ***Turing-decidable*** or simply ***decidable*** if some Turing machine decides it.[2]

For describing a turing machine, one can just use a higher definition instead of defining the whole 7-tuple.
Describing the high level lang is in a way describing the 7-tuple

HIGH LEVEL DESCRIPTION

EXAMPLE 3.7

Here we describe a Turing machine (TM) M_2 that decides $A = \{0^{2^n} \mid n \geq 0\}$, the language consisting of all strings of 0s whose length is a power of 2.

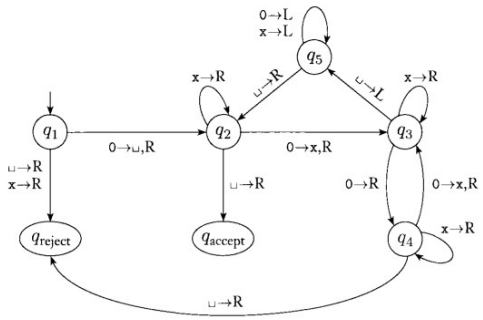
achine

M_2 = "On input string w:

1. Sweep left to right across the tape, crossing off every other 0.
2. If in stage 1 the tape contained a single 0, *accept*.
3. If in stage 1 the tape contained more than a single 0 and the number of 0s was odd, *reject*.
4. Return the head to the left-hand end of the tape.
5. Go to stage 1."

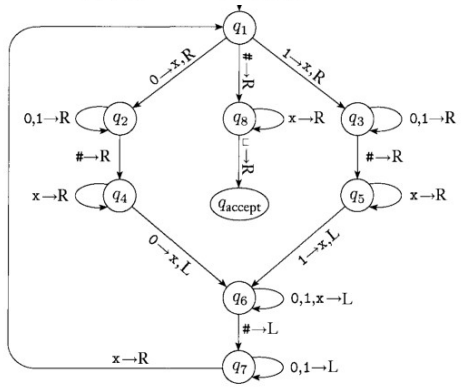
FORMAL DESCRIPTION

- $Q = \{q_1, q_2, q_3, q_4, q_5, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0\}$, and
- $\Gamma = \{0, \text{x}, \sqcup\}$.
- We describe δ with a state diagram (see Figure 3.8).
- The start, accept, and reject states are q_1, q_{accept}, and q_{reject}.



guage $B = \{w\#w \mid w \in \{0,1\}^*\}$.

- $Q = \{q_1, \ldots, q_{14}, q_{\text{accept}}, q_{\text{reject}}\}$,
- $\Sigma = \{0,1,\#\}$, and $\Gamma = \{0,1,\#,\text{x},\sqcup\}$.



In this state diagram, the label $0 \to \sqcup, R$ appears on the transition from q_1 to q_2. This label signifies that, when in state q_1 with the head reading 0, the machine goes to state q_2, writes $\sqcup$, and moves the head to the right. In other words, $\delta(q_1, 0) = (q_2, \sqcup, R)$. For clarity we use the shorthand $0 \to R$ in the transition from q_3 to q_4, to mean that the machine moves to the right when reading 0 in state q_3 but doesn't alter the tape, so $\delta(q_3, 0) = (q_4, 0, R)$.

EXAMPLE 3.11

Here, a TM M_3 is doing some elementary arithmetic. It decides the language $C = \{\mathtt{a}^i\mathtt{b}^j\mathtt{c}^k \mid i \times j = k \text{ and } i, j, k \geq 1\}$.

M_3 = "On input string w:

1. Scan the input from left to right to determine whether it is a member of $\mathtt{a}^*\mathtt{b}^*\mathtt{c}^*$ and *reject* if it isn't.
2. Return the head to the left-hand end of the tape.
3. Cross off an a and scan to the right until a b occurs. Shuttle between the b's and the c's, crossing off one of each until all b's are gone. If all c's have been crossed off and some b's remain, *reject*.
4. Restore the crossed off b's and repeat stage 3 if there is another a to cross off. If all a's have been crossed off, determine whether all c's also have been crossed off. If yes, *accept*; otherwise, *reject*."

→ can make such checks

EXAMPLE 3.12

Here, a TM M_4 is solving what is called the *element distinctness problem*. It is given a list of strings over $\{0,1\}$ separated by #s and its job is to accept if all the strings are different. The language is

$$E = \{\#x_1\#x_2\#\cdots\#x_l \mid \text{each } x_i \in \{0,1\}^* \text{ and } x_i \neq x_j \text{ for each } i \neq j\}.$$

Machine M_4 works by comparing x_1 with x_2 through x_l, then by comparing x_2 with x_3 through x_l, and so on. An informal description of the TM M_4 deciding this language follows.

M_4 = "On input w:

1. Place a mark on top of the leftmost tape symbol. If that symbol was a blank, *accept*. If that symbol was a #, continue with the next stage. Otherwise, *reject*.
2. Scan right to the next # and place a second mark on top of it. If no # is encountered before a blank symbol, only x_1 was present, so *accept*.
3. By zig-zagging, compare the two strings to the right of the marked #s. If they are equal, *reject*.
4. Move the rightmost of the two marks to the next # symbol to the right. If no # symbol is encountered before a blank symbol, move the leftmost mark to the next # to its right and the rightmost mark to the # after that. This time, if no # is available for the rightmost mark, all the strings have been compared, so *accept*.
5. Go to Stage 3."

3.2

VARIANTS OF TURING MACHINES

invariance to certain changes in the definition ***robustness***. Both finite automata and pushdown automata are somewhat robust models, but Turing machines have an astonishing degree of robustness.

The transition function would then have the form $\delta: Q \times \Gamma \longrightarrow Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}$. Might this feature allow Turing machines to recognize additional languages, thus adding to the power of the model? Of course not, because we can convert any TM with the "stay put" feature to one that does not have it. We do so by replacing each stay put transition with two transitions, one that moves to the right and the second back to the left.

This small example contains the key to showing the equivalence of [illegible] variants. To show that two models are equivalent we simply need to show that we can simulate one by the other.

MULTITAPE TURING MACHINES

A ***multitape Turing machine*** is like an ordinary Turing machine with several tapes. Each tape has its own head for reading and writing. Initially the input

$\delta: Q \times \Gamma^k \longrightarrow Q \times \Gamma^k \times \{\mathrm{L}, \mathrm{R}, \mathrm{S}\}^k$,

$\delta(q_i, a_1, \ldots, a_k) = (q_j, b_1, \ldots, b_k, \mathrm{L}, \mathrm{R}, \ldots, \mathrm{L})$

means that, if the machine is in state q_i and heads 1 through k are reading symbols a_1 through a_k, the machine goes to state q_j, writes symbols b_1 through b_k, and directs each head to move left or right, or to stay put, as specified.

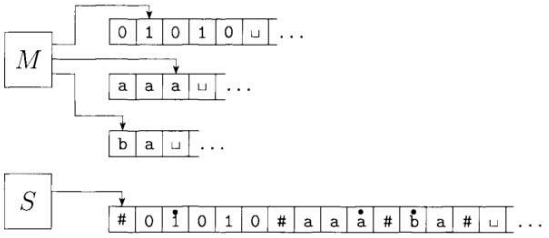
Multitape Turing machines appear to be more powerful than ordinary Turing machines, but we can show that they are equivalent in power. Recall that two machines are equivalent if they recognize the same language.

THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

PROOF We show how to convert a multitape TM M to an equivalent single-tape TM S. The key idea is to show how to simulate M with S.

Say that M has k tapes. Then S simulates the effect of k tapes by storing their information on its single tape. It uses the new symbol # as a delimiter to separate the contents of the different tapes. In addition to the contents of these tapes, S must keep track of the locations of the heads. It does so by writing a tape symbol with a dot above it to mark the place where the head on that tape would be. Think of these as "virtual" tapes and heads. As before, the "dotted" tape symbols are simply new symbols that have been added to the tape alphabet. The following figure illustrates how one tape can be used to represent three tapes.



THEOREM 3.13

Every multitape Turing machine has an equivalent single-tape Turing machine.

COROLLARY 3.15

A language is Turing-recognizable if and only if some multitape Turing machine recognizes it.

S = "On input $w = w_1 \cdots w_n$:

1. First S puts its tape into the format that represents all k tapes of M. The formatted tape contains

   $$\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#$$

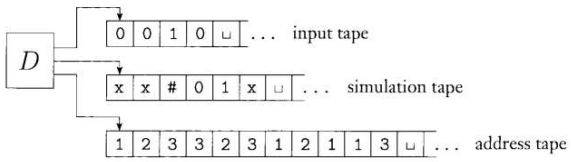
2. To simulate a single move, S scans its tape from the first #, which marks the left-hand end, to the $(k+1)$st #, which marks the right-hand end, in order to determine the symbols under the virtual heads. Then S makes a second pass to update the tapes according to the way that M's transition function dictates.
3. If at any point S moves one of the virtual heads to the right onto a #, this action signifies that M has moved the corresponding head onto the previously unread blank portion of that tape. So S writes a blank symbol on this tape cell and shifts the tape contents, from this cell until the rightmost #, one unit to the right. Then it continues the simulation as before."

NONDETERMINISTIC TURING MACHINES

$\delta: Q \times \Gamma \longrightarrow \mathcal{P}(Q \times \Gamma \times \{\mathrm{L}, \mathrm{R}\})$.

idea is to have D explore the tree by using depth-first search. The depth-first search strategy goes all the way down one branch before backing up to explore other branches. If D were to explore the tree in this manner, D could go forever down one infinite branch and miss an accepting configuration on some other branch. Hence we design D to explore the tree by using breadth first search instead. This strategy explores all branches to the same depth before going on to explore any branch to the next depth. This method guarantees that D will visit every node in the tree until it encounters an accepting configuration.

PROOF The simulating deterministic TM D has three tapes. By Theorem 3.13 this arrangement is equivalent to having a single tape. The machine D uses its three tapes in a particular way, as illustrated in the following figure. Tape 1 always contains the input string and is never altered. Tape 2 maintains a copy of N's tape on some branch of its nondeterministic computation. Tape 3 keeps track of D's location in N's nondeterministic computation tree.



THEOREM 3.16

Every nondeterministic Turing machine has an equivalent deterministic Turing machine.

COROLLARY 3.18

A language is Turing-recognizable if and only if some nondeterministic Turing machine recognizes it.

COROLLARY 3.19

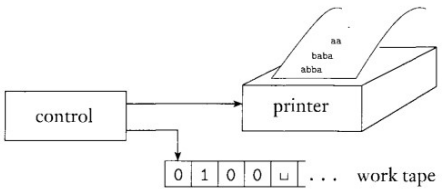
A language is decidable if and only if some nondeterministic Turing machine decides it.

Let's first consider the data representation on tape 3. Every node in the tree can have at most b children, where b is the size of the largest set of possible choices given by N's transition function. To every node in the tree we assign an address that is a string over the alphabet $\Sigma_b = \{1, 2, \ldots, b\}$. We assign the address 231 to the node we arrive at by starting at the root, going to its 2nd child, going to that node's 3rd child, and finally going to that node's 1st child. Each symbol in the string tells us which choice to make next when simulating a step in one branch in N's nondeterministic computation. Sometimes a symbol may not correspond to any choice if too few choices are available for a configuration. In that case the address is invalid and doesn't correspond to any node. Tape 3 contains a string over Σ_b. It represents the branch of N's computation from the root to the node addressed by that string, unless the address is invalid. The empty string is the address of the root of the tree. Now we are ready to describe D.

1. Initially tape 1 contains the input w, and tapes 2 and 3 are empty.
2. Copy tape 1 to tape 2.
3. Use tape 2 to simulate N with input w on one branch of its nondeterministic computation. Before each step of N consult the next symbol on tape 3 to determine which choice to make among those allowed by N's transition function. If no more symbols remain on tape 3 or if this nondeterministic choice is invalid, abort this branch by going to stage 4. Also go to stage 4 if a rejecting configuration is encountered. If an accepting configuration is encountered, *accept* the input.
4. Replace the string on tape 3 with the lexicographically next string. Simulate the next branch of N's computation by going to stage 2.

ENUMERATORS

As we mentioned earlier, some people use the term *recursively enumerable language* for Turing-recognizable language. That term originates from a type of Turing machine variant called an ***enumerator***. Loosely defined, an enumerator is a Turing machine with an attached printer. The Turing machine can use that printer as an output device to print strings. Every time the Turing machine wants to add a string to the list, it sends the string to the printer. Exercise 3.4 asks you to give a formal definition of an enumerator. The following figure depicts a schematic of this model.



An enumerator E starts with a blank input tape. If the enumerator doesn't halt, it may print an infinite list of strings. The language enumerated by E is the collection of all the strings that it eventually prints out. Moreover, E may generate the strings of the language in any order, possibly with repetitions.

THEOREM 3.21

A language is Turing-recognizable if and only if some enumerator enumerates it.

PROOF First we show that if we have an enumerator E that enumerates a language A, a TM M recognizes A. The TM M works in the following way.

M = "On input w:

1. Run E. Every time that E outputs a string, compare it with w.
2. If w ever appears in the output of E, *accept*."

Clearly, M accepts those strings that appear on E's list.

Now we do the other direction. If TM M recognizes a language A, we can construct the following enumerator E for A. Say that $s_1, s_2, s_3, \ldots$ is a list of all possible strings in Σ^*.

E = "Ignore the input.

1. Repeat the following for $i = 1, 2, 3, \ldots$
2. Run M for i steps on each input, $s_1, s_2, \ldots, s_i$.
3. If any computations accept, print out the corresponding s_j."

If M accepts a particular string s, eventually it will appear on the list generated by E. In fact, it will appear on the list infinitely many times because M runs from the beginning on each string for each repetition of step 1. This procedure gives the effect of running M in parallel on all possible input strings.

3.3 THE DEFINITION OF ALGORITHM

Informally speaking, an *algorithm* is a collection of simple instructions for carrying out some task. Commonplace in everyday life, algorithms sometimes are

The definition came in the 1936 papers of Alonzo Church and Alan Turing. Church used a notational system called the λ-calculus to define algorithms. Turing did it with his "machines." These two definitions were shown to be equivalent. This connection between the informal notion of algorithm and the precise definition has come to be called the ***Church–Turing thesis***.

such algorithms? Students commonly ask this question, especially when preparing solutions to exercises and problems. Let's entertain three possibilities. The first is the *formal description* that spells out in full the Turing machine's states, transition function, and so on. It is the lowest, most detailed, level of description. The second is a higher level of description, called the *implementation description*, in which we use English prose to describe the way that the Turing machine moves its head and the way that it stores data on its tape. At this level we do not give details of states or transition function. Third is the *high-level description*, wherein we use English prose to describe an algorithm, ignoring the implementation details. At this level we do not need to mention how the machine manages its tape or head.

We now set up a format and notation for describing Turing machines. The input to a Turing machine is always a string. If we want to provide an object other than a string as input, we must first represent that object as a string. Strings can easily represent polynomials, graphs, grammars, automata, and any combination of those objects. A Turing machine may be programmed to decode the representation so that it can be interpreted in the way we intend. Our notation for the encoding of an object O into its representation as a string is $\langle O \rangle$. If we have several objects $O_1, O_2, \ldots, O_k$, we denote their encoding into a single string $\langle O_1, O_2, \ldots, O_k \rangle$. The encoding itself can be done in many reasonable ways. It doesn't matter which one we pick because a Turing machine can always translate one such encoding into another.

In our format, we describe Turing machine algorithms with an indented segment of text within quotes. We break the algorithm into stages, each usually involving many individual steps of the Turing machine's computation. We indicate the block structure of the algorithm with further indentation. The first line of the algorithm describes the input to the machine. If the input description is simply w, the input is taken to be a string. If the input description is the encoding of an object as in $\langle A \rangle$, the Turing machine first implicitly tests whether the input properly encodes an object of the desired form and rejects it if it doesn't.

Let A be the language consisting of all strings representing undirected graphs that are connected. Recall that a graph is ***connected*** if every node can be reached from every other node by traveling along the edges of the graph. We write

$$A = \{\langle G \rangle \mid G \text{ is a connected undirected graph}\}.$$

The following is a high-level description of a TM M that decides A.

M = "On input $\langle G \rangle$, the encoding of a graph G:

1. Select the first node of G and mark it.
2. Repeat the following stage until no new nodes are marked:
3. For each node in G, mark it if it is attached by an edge to a node that is already marked.
4. Scan all the nodes of G to determine whether they all are marked. If they are, *accept*; otherwise, *reject*."

$$D_1 = \{p \mid p \text{ is a polynomial over } x \text{ with an integral root}\}.$$

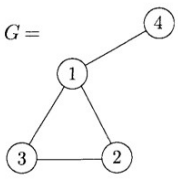
Here is a TM M_1 that recognizes D_1:

M_1 = "The input is a polynomial p over the variable x.

1. Evaluate p with x set successively to the values 0, 1, -1, 2, -2, 3, -3, ... If at any point the polynomial evaluates to 0, *accept*."

Both M_1 and $\bar{M}$ are recognizers but not deciders.

First, we must understand how $\langle G \rangle$ encodes the graph G as a string. Consider an encoding that is a list of the nodes of G followed by a list of the edges of G.

$G =$ 

$\langle G \rangle =$

`(1,2,3,4)((1,2),(2,3),(3,1),(1,4))`

input is the proper encoding of some graph. To do so, M scans the tape to be sure that there are two lists and that they are in the proper form. The first list should be a list of distinct decimal numbers, and the second should be a list of pairs of decimal numbers. Then M checks several things. First, the node list should contain no repetitions, and second, every node appearing on the edge list should also appear on the node list. For the first, we can use the procedure given

3.15 Show that the collection of decidable languages is closed under the operation of

[A]**a.** union.
b. concatenation.
c. star.
d. complementation.
e. intersection.

3.16 Show that the collection of Turing-recognizable languages is closed under the operation of

[A]**a.** union.
b. concatenation.
c. star.
d. intersection.

DECIDABILITY

21 February 2024 02:12 PM

expressed as a language, A_{DFA}. This language contains the encodings of all DFAs together with strings that the DFAs accept. Let

$$A_{\text{DFA}} = \{\langle B, w\rangle \mid B \text{ is a DFA that accepts input string } w\}.$$

THEOREM 4.1

A_{DFA} is a decidable language.

PROOF IDEA We simply need to present a TM M that decides A_{DFA}.

M = "On input $\langle B, w\rangle$, where B is a DFA and w is a string:

1. Simulate B on input w.
2. If the simulation ends in an accept state, *accept*. If it ends in a nonaccepting state, *reject*."

PROOF We mention just a few implementation details of this proof. For those of you familiar with writing programs in any standard programming language, imagine how you would write a program to carry out the simulation.

First, let's examine the input $\langle B, w\rangle$. It is a representation of a DFA B together with a string w. One reasonable representation of B is simply a list of its five components, Q, Σ, δ, q_0, and F. When M receives its input, M first determines whether it properly represents a DFA B and a string w. If not, M rejects.

Then M carries out the simulation directly. It keeps track of B's current state and B's current position in the input w by writing this information down on its tape. Initially, B's current state is q_0 and B's current input position is the leftmost symbol of w. The states and position are updated according to the specified transition function δ. When M finishes processing the last symbol of w, M accepts the input if B is in an accepting state; M rejects the input if B is in a nonaccepting state.

We can prove a similar theorem for nondeterministic finite automata. Let

$$A_{\text{NFA}} = \{\langle B, w\rangle \mid B \text{ is an NFA that accepts input string } w\}.$$

THEOREM 4.2

A_{NFA} is a decidable language.

PROOF We present a TM N that decides A_{NFA}. We could design N to operate like M, simulating an NFA instead of a DFA. Instead, we'll do it differently to illustrate a new idea: have N use M as a subroutine. Because M is designed to work with DFAs, N first converts the NFA it receives as input to a DFA before passing it to M.

N = "On input $\langle B, w\rangle$ where B is an NFA, and w is a string:

1. Convert NFA B to an equivalent DFA C, using the procedure for this conversion given in Theorem 1.39.
2. Run TM M from Theorem 4.1 on input $\langle C, w\rangle$.
3. If M accepts, *accept*; otherwise, *reject*."

Similarly, we can determine whether a regular expression generates a given string. Let $A_{\text{REX}} = \{\langle R, w\rangle \mid R \text{ is a regular expression that generates string } w\}$.

THEOREM 4.3

A_{REX} is a decidable language.

PROOF The following TM P decides A_{REX}.

P = "On input $\langle R, w\rangle$ where R is a regular expression and w is a string:

1. Convert regular expression R to an equivalent NFA A by using the procedure for this conversion given in Theorem 1.54.
2. Run TM N on input $\langle A, w\rangle$.
3. If N accepts, *accept*; if N rejects, *reject*."

Theorems 4.1, 4.2, and 4.3 illustrate that, for decidability purposes, presenting the Turing machine with a DFA, NFA, or regular expression are all equivalent because the machine is able to convert one form of encoding to another.

$E_{\text{DFA}} = \{\langle A\rangle \mid A \text{ is a DFA and } L(A) = \emptyset\}$.

THEOREM 4.4

E_{DFA} is a decidable language.

T = "On input $\langle A\rangle$ where A is a DFA:

1. Mark the start state of A.
2. Repeat until no new states get marked:
3. Mark any state that has a transition coming into it from any state that is already marked.
4. If no accept state is marked, *accept*; otherwise, *reject*."

$EQ_{\text{DFA}} = \{\langle A, B\rangle \mid A \text{ and } B \text{ are DFAs and } L(A) = L(B)\}$.

THEOREM 4.5

EQ_{DFA} is a decidable language.

PROOF To prove this theorem we use Theorem 4.4. We construct a new DFA C from A and B, where C accepts only those strings that are accepted by either A or B but not by both. Thus, if A and B recognize the same language, C will accept nothing. The language of C is

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right).$$

This expression is sometimes called the ***symmetric difference*** of $L(A)$ and $L(B)$ and is illustrated in the following figure. Here $\overline{L(A)}$ is the complement of $L(A)$. The symmetric difference is useful here because $L(C) = \emptyset$ iff $L(A) = L(B)$. We can construct C from A and B with the constructions for proving the class of regular languages closed under complementation, union, and intersection. These constructions are algorithms that can be carried out by Turing machines. Once we have constructed C we can use Theorem 4.4 to test whether $L(C)$ is empty. If it is empty, $L(A)$ and $L(B)$ must be equal.

F = "On input $\langle A, B\rangle$, where A and B are DFAs:

1. Construct DFA C as described.
2. Run TM T from Theorem 4.4 on input $\langle C\rangle$.
3. If T accepts, *accept*. If T rejects, *reject*."

$A_{\text{CFG}} = \{\langle G, w\rangle \mid G \text{ is a CFG that generates string } w\}$.

THEOREM 4.7

A_{CFG} is a decidable language.

PROOF IDEA For CFG G and string w we want to determine whether G generates w. One idea is to use G to go through all derivations to determine whether any is a derivation of w. This idea doesn't work, as infinitely many derivations may have to be tried. If G does not generate w, this algorithm would never halt. This idea gives a Turing machine that is a recognizer, but not a decider, for A_{CFG}.

To make this Turing machine into a decider we need to ensure that the algorithm tries only finitely many derivations. In Problem 2.26 (page 130) we showed that, if G were in Chomsky normal form, any derivation of w has $2n - 1$ steps, where n is the length of w. In that case checking only derivations with $2n - 1$ steps to determine whether G generates w would be sufficient. Only finitely many such derivations exist. We can convert G to Chomsky normal form by using the procedure given in Section 2.1.

PROOF The TM S for A_{CFG} follows.

S = "On input $\langle G, w\rangle$, where G is a CFG and w is a string:

1. Convert G to an equivalent grammar in Chomsky normal form.
2. List all derivations with $2n - 1$ steps, where n is the length of w, except if $n = 0$, then instead list all derivations with 1 step.
3. If any of these derivations generate w, *accept*; if not, *reject*."

$E_{\text{CFG}} = \{\langle G\rangle \mid G \text{ is a CFG and } L(G) = \emptyset\}$.

THEOREM 4.8

E_{CFG} is a decidable language.

In order to determine whether the language of a grammar is empty, we need to test whether the start variable can generate a string of terminals. The algorithm does so by solving a more general problem. It determines *for each variable* whether that variable is capable of generating a string of terminals. When the algorithm has determined that a variable can generate some string of terminals, the algorithm keeps track of this information by placing a mark on that variable.

First, the algorithm marks all the terminal symbols in the grammar. Then, it scans all the rules of the grammar. If it ever finds a rule that permits some variable to be replaced by some string of symbols all of which are already marked, the algorithm knows that this variable can be marked, too. The algorithm continues in this way until it cannot mark any additional variables. The TM R implements this algorithm.

PROOF

R = "On input $\langle G\rangle$, where G is a CFG:

1. Mark all terminal symbols in G.
2. Repeat until no new variables get marked:
3. Mark any variable A where G has a rule $A \to U_1U_2\cdots U_k$ and each symbol $U_1, \ldots, U_k$ has already been marked.
4. If the start variable is not marked, *accept*; otherwise, *reject*."

$EQ_{\text{CFG}} = \{\langle G, H\rangle \mid G \text{ and } H \text{ are CFGs and } L(G) = L(H)\}$.

a similar strategy to prove that EQ_{CFG} is decidable. But something is wrong with this idea! The class of context-free languages is *not* closed under complementation or intersection, as you proved in Exercise 2.2. In fact, EQ_{CFG} is not decidable. The technique for proving so is presented in Chapter 5.

Now we show that every context-free language is decidable by a Turing ma-

THEOREM 4.9

Every context-free language is decidable.

PROOF IDEA Let A be a CFL. Our objective is to show that A is decidable.

PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

In the context of the image you sent, the statement "CFL is decidable" means that

Every context-free language is decidable.

In the context of the image you sent, the statement "CFL is decidable" means that **every context-free language (CFL) has a corresponding Turing machine that can definitively decide whether any given string belongs to that language.** This Turing machine must always halt and output either "accept" or "reject", even for inputs that are not actually in the language.

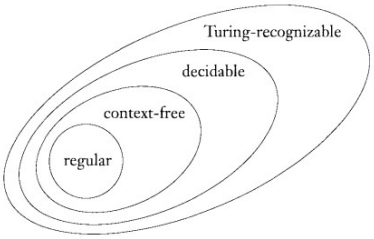
PROOF Let G be a CFG for A and design a TM M_G that decides A. We build a copy of G into M_G. It works as follows.

M_G = "On input w:

1. Run TM S on input $\langle G, w\rangle$
2. If this machine accepts, *accept*; if it rejects, *reject*."

TM S is chomsky normal form

classes of languages that we have described so far: regular, context free, decidable, and Turing-recognizable. The following figure depicts this relationship.



HALTING PROBLEM

$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}$.

THEOREM 4.11

A_{TM} is undecidable.

Before we get to the proof, let's first observe that A_{TM} is Turing-recognizable. Thus this theorem shows that recognizers *are* more powerful than deciders. Requiring a TM to halt on all inputs restricts the kinds of languages that it can recognize. The following Turing machine U recognizes A_{TM}.

U = "On input $\langle M, w\rangle$, where M is a TM and w is a string:

1. Simulate M on input w.
2. If M ever enters its accept state, *accept*; if M ever enters its reject state, *reject*."

Note that this machine loops on input $\langle M, w\rangle$ if M loops on w, which is why this machine does not decide A_{TM}. If the algorithm had some way to determine that M was not halting on w, it could *reject*. Hence A_{TM} is sometimes called the ***halting problem***. As we demonstrate, an algorithm has no way to make this determination.

The Turing machine U is interesting in its own right. It is an example of the *universal Turing machine* first proposed by Turing. This machine is called universal because it is capable of simulating any other Turing machine from the description of that machine. The universal Turing machine played an important early role in stimulating the development of stored-program computers.

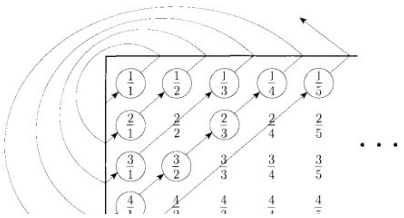
DIAGONALISATION

DEFINITION 4.12

Assume that we have sets A and B and a function f from A to B. Say that f is ***one-to-one*** if it never maps two different elements to the same place—that is, if $f(a) \neq f(b)$ whenever $a \neq b$. Say that f is ***onto*** if it hits every element of B—that is, if for every $b \in B$ there is an $a \in A$ such that $f(a) = b$. Say that A and B are the ***same size*** if there is a one-to-one, onto function $f: A \longrightarrow B$. A function that is both one-to-one and onto is called a ***correspondence***. In a correspondence every element of A maps to a unique element of B and each element of B has a unique element of A mapping to it. A correspondence is simply a way of pairing the elements of A with the elements of B.

DEFINITION 4.14

A set A is ***countable*** if either it is finite or it has the same size as $\mathcal{N}$.



THEOREM 4.17

$\mathcal{R}$ is uncountable.

We construct the desired x by giving its decimal representation. It is a number between 0 and 1, so all its significant digits are fractional digits following the decimal point. Our objective is to ensure that $x \neq f(n)$ for any n. To ensure that $x \neq f(1)$ we let the first digit of x be anything different from the first fractional digit 1 of $f(1) = 3.14159\ldots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$ we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.55555\ldots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12345\ldots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f, we obtain all the digits of x, as

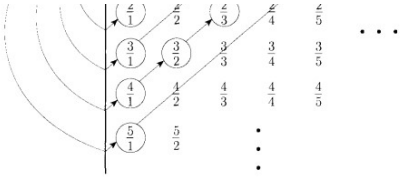


FIGURE 4.16
A correspondence of $\mathcal{N}$ and $\mathcal{Q}$

digit 1 of $f(1) = 3.14159\ldots$. Arbitrarily, we let it be 4. To ensure that $x \neq f(2)$ we let the second digit of x be anything different from the second fractional digit 5 of $f(2) = 55.555555\ldots$. Arbitrarily, we let it be 6. The third fractional digit of $f(3) = 0.12345\ldots$ is 3, so we let x be anything different—say, 4. Continuing in this way down the diagonal of the table for f, we obtain all the digits of x, as shown in the following table. We know that x is not $f(n)$ for any n because it differs from $f(n)$ in the nth fractional digit. (A slight problem arises because certain numbers, such as $0.1999\ldots$ and $0.2000\ldots$, are equal even though their decimal representations are different. We avoid this problem by never selecting the digits 0 or 9 when we construct x.)

COROLLARY 4.18

Some languages are not Turing-recognizable.

PROOF To show that the set of all Turing machines is countable we first observe that the set of all strings Σ^* is countable, for any alphabet Σ. With only finitely many strings of each length, we may form a list of Σ^* by writing down all strings of length 0, length 1, length 2, and so on.

The set of all Turing machines is countable because each Turing machine M has an encoding into a string $\langle M \rangle$. If we simply omit those strings that are not legal encodings of Turing machines, we can obtain a list of all Turing machines.

To show that the set of all languages is uncountable we first observe that the set of all infinite binary sequences is uncountable. An *infinite binary sequence* is an unending sequence of 0s and 1s. Let $\mathcal{B}$ be the set of all infinite binary sequences. We can show that $\mathcal{B}$ is uncountable by using a proof by diagonalization similar to the one we used in Theorem 4.17 to show that $\mathcal{R}$ is uncountable.

Let $\mathcal{L}$ be the set of all languages over alphabet Σ. We show that $\mathcal{L}$ is uncountable by giving a correspondence with $\mathcal{B}$, thus showing that the two sets are the same size. Let $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Each language $A \in \mathcal{L}$ has a unique sequence in $\mathcal{B}$. The ith bit of that sequence is a 1 if $s_i \in A$ and is a 0 if $s_i \notin A$, which is called the ***characteristic sequence*** of A. For example, if A were the language of all strings starting with a 0 over the alphabet $\{0,1\}$, its characteristic sequence χ_A would be

$$\begin{array}{lccccccccccc}
\Sigma^* = \{ & \varepsilon & , & 0 & , & 1 & , & 00 & , & 01 & , & 10 & , & 11 & , & 000 & , & 001 & , & \cdots & \} ; \\
A = \{ & & & 0 & , & & & 00 & , & 01 & , & & & & & 000 & , & 001 & , & \cdots & \} ; \\
\chi_A = & 0 & & 1 & & 0 & & 1 & & 1 & & 0 & & 0 & & 1 & & 1 & & \cdots & .
\end{array}$$

The function $f: \mathcal{L} \longrightarrow \mathcal{B}$, where $f(A)$ equals the characteristic sequence of A, is one-to-one and onto and hence a correspondence. Therefore, as $\mathcal{B}$ is uncountable, $\mathcal{L}$ is uncountable as well.

The preceding theorem has an important application to the theory of computation. It shows that some languages are not decidable or even Turing-recognizable, for the reason that there are uncountably many languages yet only countably many Turing machines. Because each Turing machine can recognize a single language and there are more languages than Turing machines, some languages are not recognized by any Turing machine. Such languages are not Turing-recognizable, as we state in the following corollary.

THE HALTING PROBLEM IS UNDECIDABLE

Now we are ready to prove Theorem 4.11, the undecidability of the language

$$A_{\text{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ accepts } w\}.$$

undecidable
Recognizable

PROOF We assume that A_{TM} is decidable and obtain a contradiction. Suppose that H is a decider for A_{TM}. On input $\langle M, w\rangle$, where M is a TM and w is a string, H halts and accepts if M accepts w. Furthermore, H halts and rejects if M fails to accept w. In other words, we assume that H is a TM, where

$$H(\langle M, w\rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ accepts } w \\ \textit{reject} & \text{if } M \text{ does not accept } w. \end{cases}$$

Now we construct a new Turing machine D with H as a subroutine. This new TM calls H to determine what M does when the input to M is its own description $\langle M \rangle$. Once D has determined this information, it does the opposite. That is, it rejects if M accepts and accepts if M does not accept. The following is a description of D.

D = "On input $\langle M \rangle$, where M is a TM:

1. Run H on input $\langle M, \langle M \rangle\rangle$.
2. Output the opposite of what H outputs; that is, if H accepts, *reject* and if H rejects, *accept*."

Don't be confused by the idea of running a machine on its own description! That is similar to running a program with itself as input, something that does occasionally occur in practice. For example, a compiler is a program that translates other programs. A compiler for the language Pascal may itself be written in Pascal, so running that program on itself would make sense. In summary,

$$D(\langle M \rangle) = \begin{cases} \textit{accept} & \text{if } M \text{ does not accept } \langle M \rangle \\ \textit{reject} & \text{if } M \text{ accepts } \langle M \rangle. \end{cases}$$

What happens when we run D with its own description $\langle D \rangle$ as input? In that case we get

$$D(\langle D \rangle) = \begin{cases} \textit{accept} & \text{if } D \text{ does not accept } \langle D \rangle \\ \textit{reject} & \text{if } D \text{ accepts } \langle D \rangle. \end{cases}$$

No matter what D does, it is forced to do the opposite, which is obviously a contradiction. Thus neither TM D nor TM H can exist.

Now since H decides for every machine
But D cannot be decided. ∴ contra^n
H & D both cannot exists.

↓ Now since H (the decider) cannot exist, therefore not every input <M,w> can be decided. Therefore lang. is undecidable.

if it REJECTS then it means D accepts <D> which recursively means it rejects. This goes on & on so it is undecisive

4 on so it is undecisive

A TURING-UNRECOGNIZABLE LANGUAGE

a language and its complement are Turing-recognizable, the language is decidable. Hence, for any undecidable language, either it or its complement is not Turing-recognizable. Recall that the complement of a language is the language consisting of all strings that are not in the language. We say that a language is ***co-Turing-recognizable*** if it is the complement of a Turing-recognizable language.

PROOF We have two directions to prove. First, if A is decidable, we can easily see that both A and its complement $\overline{A}$ are Turing-recognizable. Any decidable language is Turing-recognizable, and the complement of a decidable language also is decidable.

For the other direction, if both A and $\overline{A}$ are Turing-recognizable, we let M_1 be the recognizer for A and M_2 be the recognizer for $\overline{A}$. The following Turing machine M is a decider for A.

M = "On input w:

1. Run both M_1 and M_2 on input w in parallel.
2. If M_1 accepts, *accept*; if M_2 accepts, *reject*."

Running the two machines in parallel means that M has two tapes, one for simulating M_1 and the other for simulating M_2. In this case M takes turns simulating one step of each machine, which continues until one of them accepts.

Now we show that M decides A. Every string w is either in A or $\overline{A}$. Therefore either M_1 or M_2 must accept w. Because M halts whenever M_1 or M_2 accepts, M always halts and so it is a decider. Furthermore, it accepts all strings in A and rejects all strings not in A. So M is a decider for A, and thus A is decidable.

THEOREM 4.22

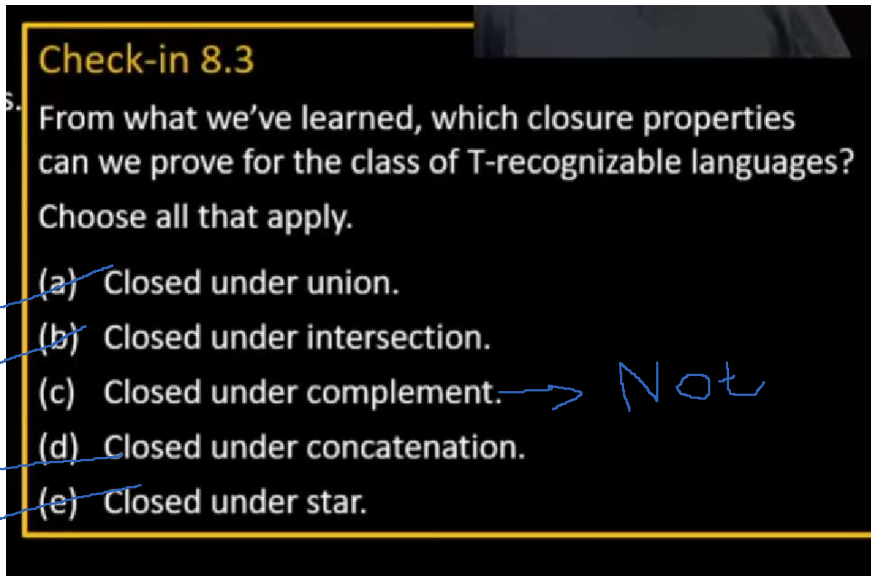
A language is decidable iff it is Turing-recognizable and co-Turing-recognizable.

COROLLARY 4.23

$\overline{A_{\text{TM}}}$ is not Turing-recognizable.

PROOF We know that A_{TM} is Turing-recognizable. If $\overline{A_{\text{TM}}}$ also were Turing-recognizable, A_{TM} would be decidable. Theorem 4.11 tells us that A_{TM} is not decidable, so $\overline{A_{\text{TM}}}$ must not be Turing-recognizable.

on a Turing machine and gave one example of a problem, A_{TM}, that is computationally unsolvable. In this chapter we examine several additional unsolvable



REDUCIBILITY

21 February 2024 04:24 PM

A ***reduction*** is a way of converting one problem to another problem in such a way that a solution to the second problem can be used to solve the first problem.

Reducibility plays an important role in classifying problems by decidability and later in complexity theory as well. When A is reducible to B, solving A cannot be harder than solving B because a solution to B gives a solution to A. In terms of computability theory, if A is reducible to B and B is decidable, A also is decidable. Equivalently, if A is undecidable and reducible to B, B is undecidable.

finding whether a Turing machine accepts a given input. Let's consider a related problem, $HALT_{\mathsf{TM}}$, the problem of determining whether a Turing machine halts (by accepting or rejecting) on a given input.[1] We use the undecidability of A_{TM} to prove the undecidability of $HALT_{\mathsf{TM}}$ by reducing A_{TM} to $HALT_{\mathsf{TM}}$. Let

$$HALT_{\mathsf{TM}} = \{\langle M, w\rangle \mid M \text{ is a TM and } M \text{ halts on input } w\}.$$

THEOREM 5.1

$HALT_{\mathsf{TM}}$ is undecidable.

PROOF Let's assume for the purposes of obtaining a contradiction that TM R decides $HALT_{\mathsf{TM}}$. We construct TM S to decide A_{TM}, with S operating as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Run TM R on input $\langle M, w\rangle$.
2. If R rejects, *reject*.
3. If R accepts, simulate M on w until it halts.
4. If M has accepted, *accept*; if M has rejected, *reject*."

Clearly, if R decides $HALT_{\mathsf{TM}}$, then S decides A_{TM}. Because A_{TM} is undecidable, $HALT_{\mathsf{TM}}$ also must be undecidable.

of the reducibility method for proving undecidability. Let

$$E_{\mathsf{TM}} = \{\langle M\rangle \mid M \text{ is a TM and } L(M) = \emptyset\}.$$

THEOREM 5.2

E_{TM} is undecidable.

PROOF Let's write the modified machine described in the proof idea using our standard notation. We call it M_1.

M_1 = "On input x:

1. If $x \neq w$, *reject*.
2. If $x = w$, run M on input w and *accept* if M does."

This machine has the string w as part of its description. It conducts the test of whether $x = w$ in the obvious way, by scanning the input and comparing it character by character with w to determine whether they are the same.

Putting all this together, we assume that TM R decides E_{TM} and construct TM S that decides A_{TM} as follows.

S = "On input $\langle M, w\rangle$, an encoding of a TM M and a string w:

1. Use the description of M and w to construct the TM M_1 just described.
2. Run R on input $\langle M_1\rangle$.
3. If R accepts, *reject*; if R rejects, *accept*."

Note that S must actually be able to compute a description of M_1 from a description of M and w. It is able to do so because it needs only add extra states to M that perform the $x = w$ test.

If R were a decider for E_{TM}, S would be a decider for A_{TM}. A decider for A_{TM} cannot exist, so we know that E_{TM} must be undecidable.

whether the Turing machine recognizes a regular language. Let

$REGULAR_{\mathsf{TM}} = \{\langle M \rangle \mid M \text{ is a TM and } L(M) \text{ is a regular language}\}$.

THEOREM 5.3

$REGULAR_{\mathsf{TM}}$ is undecidable.

PROOF IDEA As usual for undecidability theorems, this proof is by reduction from A_{TM}. We assume that $REGULAR_{\mathsf{TM}}$ is decidable by a TM R and use this assumption to construct a TM S that decides A_{TM}. Less obvious now is how to use R's ability to assist S in its task. Nonetheless we can do so.

The idea is for S to take its input $\langle M, w \rangle$ and modify M so that the resulting TM recognizes a regular language if and only if M accepts w. We call the modified machine M_2. We design M_2 to recognize the nonregular language $\{0^n 1^n \mid n \geq 0\}$ if M does not accept w, and to recognize the regular language Σ^* if M accepts w. We must specify how S can construct such an M_2 from M and w. Here, M_2 works by automatically accepting all strings in $\{0^n 1^n \mid n \geq 0\}$. In addition, if M accepts w, M_2 accepts all other strings.

PROOF We let R be a TM that decides $REGULAR_{\mathsf{TM}}$ and construct TM S to decide A_{TM}. Then S works in the following manner.

S = "On input $\langle M, w \rangle$, where M is a TM and w is a string:

1. Construct the following TM M_2.
   M_2 = "On input x:
   1. If x has the form $0^n 1^n$, *accept*.
   2. If x does not have this form, run M on input w and *accept* if M accepts w."
2. Run R on input $\langle M_2 \rangle$.
3. If R accepts, *accept*; if R rejects, *reject*."

Similarly, the problems of testing whether the language of a Turing machine is a context-free language, a decidable language, or even a finite language, can be shown to be undecidable with similar proofs. In fact, a general result, called

proof by reduction from E_{TM}. Let

$EQ_{\mathsf{TM}} = \{\langle M_1, M_2 \rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$.

THEOREM 5.4

EQ_{TM} is undecidable.

PROOF IDEA Show that, if EQ_{TM} were decidable, E_{TM} also would be decidable, by giving a reduction from E_{TM} to EQ_{TM}. The idea is simple. E_{TM} is the problem of determining whether the language of a TM is empty. EQ_{TM} is the problem of determining whether the languages of two TMs are the same. If one of these languages happens to be $\emptyset$, we end up with the problem of determining whether the language of the other machine is empty—that is, the E_{TM} problem. So in a sense, the E_{TM} problem is a special case of the EQ_{TM} problem wherein one of the machines is fixed to recognize the empty language. This idea makes giving the reduction easy.

PROOF We let TM R decide EQ_{TM} and construct TM S to decide E_{TM} as follows.

S = "On input $\langle M \rangle$, where M is a TM:

1. Run R on input $\langle M, M_1 \rangle$, where M_1 is a TM that rejects all inputs.
2. If R accepts, *accept*; if R rejects, *reject*."

If R decides EQ_{TM}, S decides E_{TM}. But E_{TM} is undecidable by Theorem 5.2, so EQ_{TM} also must be undecidable.

MAPPING REDUCIBLITY

Our chosen is a simple type of reducibility called ***mapping reducibility***.

Roughly speaking, being able to reduce problem A to problem B by using a mapping reducibility means that a computable function exists that converts instances of problem A to instances of problem B. If we have such a conversion function, called a *reduction*, we can solve A with a solver for B. The reason is

DEFINITION 5.17

A function $f: \Sigma^* \longrightarrow \Sigma^*$ is a ***computable function*** if some Turing machine M, on every input w, halts with just $f(w)$ on its tape.

EXAMPLE 5.18

All usual arithmetic operations on integers are computable functions. For example, we can make a machine that takes input $\langle m, n \rangle$ and returns $m + n$, the sum of m and n. We don't give any details here, leaving them as exercises.

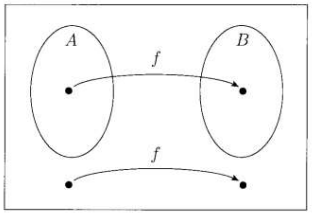
DEFINITION 5.20

Language A is ***mapping reducible*** to language B, written $A \leq_m B$, if there is a computable function $f: \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$w \in A \iff f(w) \in B.$$

The function f is called the ***reduction*** of A to B.

The following figure illustrates mapping reducibility.



THEOREM 5.22

If $A \leq_m B$ and B is decidable, then A is decidable.

PROOF We let M be the decider for B and f be the reduction from A to B. We describe a decider N for A as follows.

N = "On input w:

1. Compute $f(w)$.
2. Run M on input $f(w)$ and output whatever M outputs."

Clearly, if $w \in A$, then $f(w) \in B$ because f is a reduction from A to B. Thus M accepts $f(w)$ whenever $w \in A$. Therefore N works as desired.

COROLLARY 5.23

If $A \leq_m B$ and A is undecidable, then B is undecidable.

THEOREM 5.28

If $A \leq_m B$ and B is Turing-recognizable, then A is Turing-recognizable.

COROLLARY 5.29

If $A \leq_m B$ and A is not Turing-recognizable, then B is not Turing-recognizable.

THEOREM 5.30

EQ_{TM} is neither Turing-recognizable nor co-Turing-recognizable.

PROOF First we show that EQ_{TM} is not Turing-recognizable. We do so by showing that A_{TM} is reducible to $\overline{EQ_{\mathsf{TM}}}$. The reducing function f works as follows.

F = "On input $\langle M, w\rangle$ where M is a TM and w a string:

1. Construct the following two machines M_1 and M_2.
   M_1 = "On any input:
   1. *Reject*."
   M_2 = "On any input:
   1. Run M on w. If it accepts, *accept*."
2. Output $\langle M_1, M_2\rangle$."

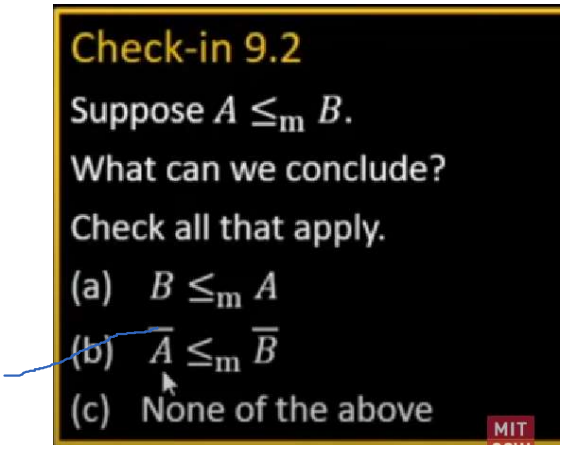
Here, M_1 accepts nothing. If M accepts w, M_2 accepts everything, and so the two machines are not equivalent. Conversely, if M doesn't accept w, M_2 accepts nothing, and they are equivalent. Thus f reduces A_{TM} to $\overline{EQ_{\mathsf{TM}}}$, as desired.

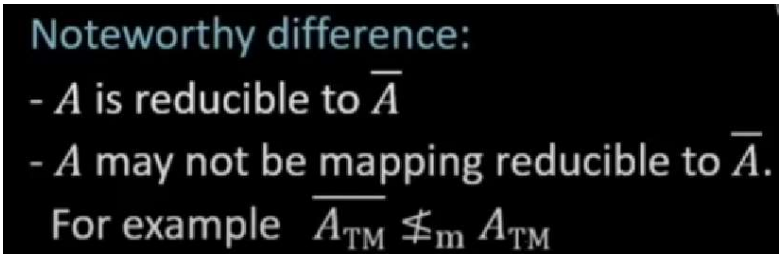
To show that $\overline{EQ_{\mathsf{TM}}}$ is not Turing-recognizable we give a reduction from A_{TM} to the complement of $\overline{EQ_{\mathsf{TM}}}$—namely, EQ_{TM}. Hence we show that $A_{\mathsf{TM}} \leq_{\mathrm{m}} EQ_{\mathsf{TM}}$. The following TM G computes the reducing function g.

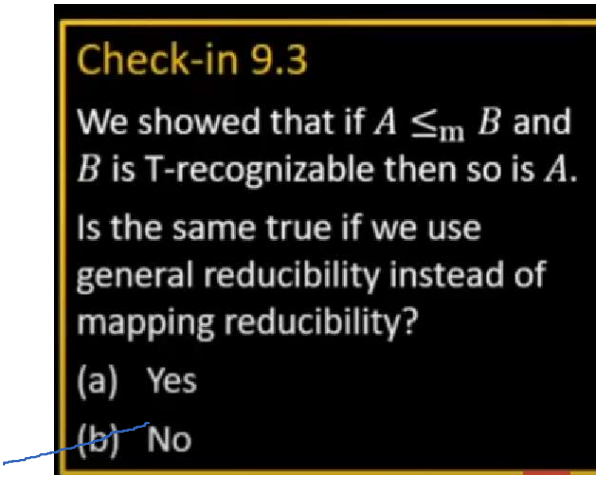
G = "The input is $\langle M, w\rangle$ where M is a TM and w a string:

1. Construct the following two machines M_1 and M_2.
   M_1 = "On any input:
   1. *Accept*."
   M_2 = "On any input:
   1. Run M on w.
   2. If it accepts, *accept*."
2. Output $\langle M_1, M_2\rangle$."

The only difference between f and g is in machine M_1. In f, machine M_1 always rejects, whereas in g it always accepts. In both f and g, M accepts w iff M_2 always accepts. In g, M accepts w iff M_1 and M_2 are equivalent. That is why g is a reduction from A_{TM} to EQ_{TM}.







To prove B is undecidable:
- Show undecidable A is reducible to B. (often A is A_{TM})
- Template: Assume TM R decides B.
  Construct TM S deciding A. Contradiction.

To prove B is T-unrecognizable:
- Show T-unrecognizable A is mapping reducible to B. (often A is $\overline{A_{\text{TM}}}$)
- Template: give reduction function f.

9. Reducibility

E_{TM} is T-unrecognizable

Recall $E_{\text{TM}} = \{\langle M \rangle | \ M \text{ is a TM and } L(M) = \emptyset \}$

Theorem: E_{TM} is T-unrecognizable

Proof: Show $\overline{A_{\text{TM}}} \leq_{\text{m}} E_{\text{TM}}$

Reduction function: $f(\langle M, w \rangle) = \langle M_w \rangle$

Explanation: $\langle M, w \rangle \in \overline{A_{\text{TM}}}$ iff $\langle M_w \rangle \in E_{\text{TM}}$

M rejects w iff $L(\langle M_w \rangle) = \emptyset$

Recall TM M_w = "On input x
1. If $x \neq w$, *reject*.
2. else run M on w
3. *Accept* if M accepts."

EQ_{TM} and EQ_{TM} are T-unrecognizable

$EQ_{\text{TM}} = \{\langle M_1, M_2\rangle \mid M_1 \text{ and } M_2 \text{ are TMs and } L(M_1) = L(M_2)\}$

Theorem: Both EQ_{TM} and $\overline{EQ_{\text{TM}}}$ are T-unrecognizable

Proof: (1) $\overline{A_{\text{TM}}} \leq_{\text{m}} EQ_{\text{TM}}$

(2) $\overline{A_{\text{TM}}} \leq_{\text{m}} \overline{EQ_{\text{TM}}}$

For any w let $T_w =$ "On input x

1. Ignore x.
2. Simulate M on w."

(1) Here we give f which maps $\overline{A_{\text{TM}}}$ problems (of the form $\langle M, w\rangle$) to EQ_{TM} problems (of the form $\langle T_1, T_2\rangle$).

$f(\langle M, w\rangle) = \langle T_w, T_{\text{reject}}\rangle$ T_{reject} is a TM that always rejects.

(2) Similarly $f(\langle M, w\rangle) = \langle T_w, T_{\text{accept}}\rangle$ T_{accept} always accepts.

RECURSION THEOREM

21 February 2024 07:23 PM

SELF-REFERENCE

Let's begin by making a Turing machine that ignores its input and prints out a copy of its own description. We call this machine *SELF*. To help describe *SELF*, we need the following lemma.

LEMMA 6.1

There is a computable function $q: \Sigma^* \longrightarrow \Sigma^*$, where if w is any string, $q(w)$ is the description of a Turing machine P_w that prints out w and then halts.

PROOF Once we understand the statement of this lemma, the proof is easy. Obviously, we can take any string w and construct from it a Turing machine that has w built into a table so that the machine can simply output w when started. The following TM Q computes $q(w)$.

Q = "On input string w:

1. Construct the following Turing machine P_w.
   P_w = "On any input:
   1. Erase input.
   2. Write w on the tape.
   3. Halt."
2. Output $\langle P_w \rangle$."

The Turing machine *SELF* is in two parts, A and B. We think of A and B as being two separate procedures that go together to make up *SELF*. We want *SELF* to print out $\langle SELF \rangle = \langle AB \rangle$.

Part A runs first and upon completion passes control to B. The job of A is to print out a description of B, and conversely the job of B is to print out a description of A. The result is the desired description of *SELF*. The jobs are

[illegible] but they are carried out differently. We show how to get part A first.

For A we use the machine $P_{\langle B \rangle}$, described by $q(\langle B \rangle)$, which is the result of applying the function q to $\langle B \rangle$. Thus part A is a Turing machine that prints out $\langle B \rangle$. Our description of A depends on having a description of B. So we can't complete the description of A until we construct B.

defined in terms of B. That would be a *circular* definition of an object in terms of itself, a logical transgression. Instead, we define B so that it prints A by using a different strategy: B *computes* A from the output that A produces.

$A = P_{\langle B \rangle}$, and

B = "On input $\langle M \rangle$, where M is a portion of a TM:

1. Compute $q(\langle M \rangle)$.
2. Combine the result with $\langle M \rangle$ to make a complete TM.
3. Print the description of this TM and halt."

If we now run *SELF* we observe the following behavior.

1. First A runs. It prints $\langle B \rangle$ on the tape.
2. B starts. It looks at the tape and finds its input, $\langle B \rangle$.
3. B calculates $q(\langle B \rangle) = \langle A \rangle$ and combines that with $\langle B \rangle$ into a TM description, $\langle SELF \rangle$.
4. B prints this description and halts.

THEOREM 6.3

Recursion theorem Let T be a Turing machine that computes a function $t: \Sigma^* \times \Sigma^* \longrightarrow \Sigma^*$. There is a Turing machine R that computes a function $r: \Sigma^* \longrightarrow \Sigma^*$, where for every w,

$$r(w) = t(\langle R \rangle, w).$$

Theorem: For any TM T there is a TM R where for all w R on input w operates in the same way as T on input $\langle w, R \rangle$.

SELF = "On any input:

1. Obtain, via the recursion theorem, own description $\langle SELF \rangle$.
2. Print $\langle SELF \rangle$."

PROOF The proof is similar to the construction of *SELF*. We construct a TM R in three parts, A, B, and T, where T is given by the statement of the theorem; a schematic diagram is presented in the following figure.

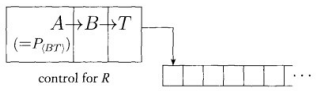
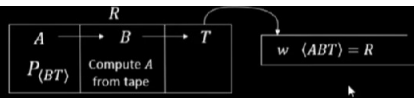


FIGURE 6.4
Schematic of R

Here, A is the Turing machine $P_{\langle BT \rangle}$ described by $q(\langle BT \rangle)$. To preserve the input w, we redesign q so that $P_{\langle BT \rangle}$ writes its output following any string preexisting on the tape. After A runs, the tape contains $w\langle BT \rangle$.

Again, B is a procedure that examines its tape and applies q to its contents. The result is $\langle A \rangle$. Then B combines A, B, and T into a single machine and obtains its description $\langle ABT \rangle = \langle R \rangle$. Finally, it encodes that description together with w, places the resulting string $\langle R, w \rangle$ on the tape, and passes control to T.

Proof of Theorem: R has three parts: A, B, and T.
T is given
$A = P_{\langle BT \rangle}$
B = "1. Compute q(tape contents after w) to get A.
2. Combine with BT to get $ABT = R$.
3. Pass control to T on input $\langle w, R \rangle$."



THEOREM 6.5

A_{TM} is undecidable.

PROOF We assume that Turing machine H decides A_{TM}, for the purposes of obtaining a contradiction. We construct the following machine B.

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B \rangle$.
2. Run H on input $\langle B, w \rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and

THEOREM 6.5

A_{TM} is undecidable.

PROOF We assume that Turing machine H decides A_{TM}, for the purposes of obtaining a contradiction. We construct the following machine B.

B = "On input w:

1. Obtain, via the recursion theorem, own description $\langle B \rangle$.
2. Run H on input $\langle B, w \rangle$.
3. Do the opposite of what H says. That is, *accept* if H rejects and *reject* if H accepts."

Running B on input w does the opposite of what H declares it does. Therefore H cannot be deciding A_{TM}. Done!

$A = P_{\langle BT \rangle}$

B = "1. Compute q(tape contents after w) to get A.
2. Combine with BT to get $ABT = R$.
3. Pass control to T on input $\langle w, R \rangle$."

$P_{\langle BT \rangle}$	Compute A from tape	

DEFINITION 6.6

If M is a Turing machine, then we say that the *length* of the description $\langle M \rangle$ of M is the number of symbols in the string describing M. Say that M is ***minimal*** if there is no Turing machine equivalent to M that has a shorter description. Let

$$MIN_{\text{TM}} = \{\langle M \rangle \mid M \text{ is a minimal TM}\}.$$

THEOREM 6.7

MIN_{TM} is not Turing-recognizable.

PROOF Assume that some TM E enumerates MIN_{TM} and obtain a contradiction. We construct the following TM C.

C = "On input w:

1. Obtain, via the recursion theorem, own description $\langle C \rangle$.
2. Run the enumerator E until a machine D appears with a longer description than that of C.
3. Simulate D on input w."

Because MIN_{TM} is infinite, E's list must contain a TM with a longer description than C's description. Therefore step 2 of C eventually terminates with some TM D that is longer than C. Then C simulates D and so is equivalent to it. Because C is shorter than D and is equivalent to it, D cannot be minimal. But D appears on the list that E produces. Thus we have a contradiction.

THEOREM 6.8

Let $t\colon \Sigma^* \longrightarrow \Sigma^*$ be a computable function. Then there is a Turing machine F for which $t(\langle F \rangle)$ describes a Turing machine equivalent to F. Here we'll assume that if a string isn't a proper Turing machine encoding, it describes a Turing machine that always rejects immediately.

Theorem: For any computable function $f\colon \Sigma^* \to \Sigma^*$, there is a TM R such that $L(R) = L(S)$ where $f(\langle R \rangle) = \langle S \rangle$.

PROOF Let F be the following Turing machine.

F = "On input w:

1. Obtain, via the recursion theorem, own description $\langle F \rangle$.
2. Compute $t(\langle F \rangle)$ to obtain the description of a TM G.
3. Simulate G on w."

Clearly, $\langle F \rangle$ and $t(\langle F \rangle) = \langle G \rangle$ describe equivalent Turing machines because F simulates G.

Check-in 11.1

Implementations of the Recursion Theorem have two parts, a Template and an Action. In the TM and English implementations, which is the Action part?

(a) A and the upper phrase
(b) A and the lower phrase
(c) B and the upper phrase
(d) B and the lower phrase.

Write the following twice, the second time in quotes

"Write the following twice, the second time in quotes"

Write the following twice, the second time in quotes
"Write the following twice, the second time in quotes"

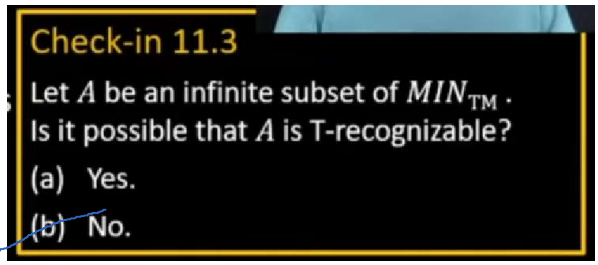
action

Check-in 11.2

Can we use the Recursion Theorem to design a TM T where $L(T) = \{\langle T \rangle\}$?

(a) Yes.
(b) No.

MIT OCW



You can have lang which are not turing recog but have infinite turing recog subsets

NFA TO DFA CONVERSION

If k is the number of states of the NFA, it has 2^k subsets of states. Each subset corresponds to one of the possibilities that the DFA must remember, so the DFA simulating the NFA will have 2^k states. Now we need to figure out which will be the start state and accept states of the DFA, and what will be its transition function. We can discuss this more easily after setting up some formal notation.

PROOF Let $N = (Q, \Sigma, \delta, q_0, F)$ be the NFA recognizing some language A. We construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ recognizing A. Before doing the full construction, let's first consider the easier case wherein N has no ε arrows. Later we take the ε arrows into account.

1. $Q' = \mathcal{P}(Q)$.
   Every state of M is a set of states of N. Recall that $\mathcal{P}(Q)$ is the set of subsets of Q.
2. For $R \in Q'$ and $a \in \Sigma$ let $\delta'(R, a) = \{q \in Q \mid q \in \delta(r, a) \text{ for some } r \in R\}$.
   If R is a state of M, it is also a set of states of N. When M reads a symbol a in state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. Another way to write this expression is
   $$\delta'(R, a) = \bigcup_{r \in R} \delta(r, a).$$ [4]
3. $q_0' = \{q_0\}$.
   M starts in the state corresponding to the collection containing just the start state of N.
4. $F' = \{R \in Q' \mid R \text{ contains an accept state of } N\}$.
   The machine M accepts if one of the possible states that N could be in at this point is an accept state.

Consideration of ε

Now we need to consider the ε arrows. To do so we set up an extra bit of notation. For any state R of M we define $E(R)$ to be the collection of states that can be reached from R by going only along ε arrows, including the members of R themselves. Formally, for $R \subseteq Q$ let

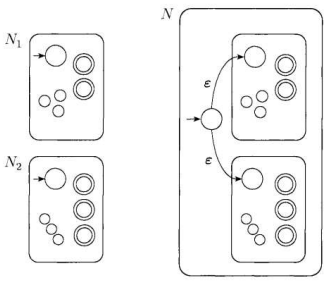
$E(R) = \{q \mid q \text{ can be reached from } R \text{ by traveling along 0 or more } \varepsilon \text{ arrows}\}$.

Then we modify the transition function of M to place additional fingers on all states that can be reached by going along ε arrows after every step. Replacing $\delta(r, a)$ by $E(\delta(r, a))$ achieves this effect. Thus

$$\delta'(R, a) = \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}.$$

Additionally we need to modify the start state of M to move the fingers initially to all possible states that can be reached from the start state of N along the ε arrows. Changing q_0' to be $E(\{q_0\})$ achieves this effect. We have now completed the construction of the DFA M that simulates the NFA N.

The construction of M obviously works correctly. At every step in the computation of M on an input, it clearly enters a state that corresponds to the subset of states that N could be in at that point. Thus our proof is complete.



proof of A∪B using NFA is easy

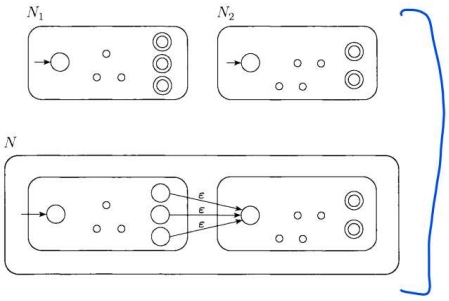
PROOF

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize $A_1 \cup A_2$.

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2, with the addition of a new start state q_0.
2. The state q_0 is the start state of N.
3. The accept states $F = F_1 \cup F_2$.
   The accept states of N are all the accept states of N_1 and N_2. That way N accepts if either N_1 accepts or N_2 accepts.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,
   $$\delta(q, a) = \begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

for A∪B [ ] inside machines [ ] at start state



A∘B using NFA

Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1, and
$N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ recognize A_2.

Construct $N = (Q, \Sigma, \delta, q_1, F_2)$ to recognize $A_1 \circ A_2$.

1. $Q = Q_1 \cup Q_2$.
   The states of N are all the states of N_1 and N_2.
2. The state q_1 is the same as the start state of N_1.
3. The accept states F_2 are the same as the accept states of N_2.
4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

for A∘B

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ and } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

imp to consider all possibilities of $\delta(q,a)$

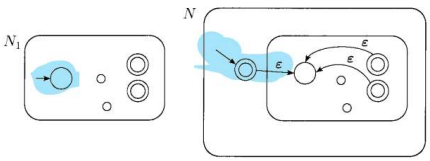


FIGURE 1.50
Construction of N to recognize A^*

PROOF Let $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ recognize A_1.
Construct $N = (Q, \Sigma, \delta, q_0, F)$ to recognize A_1^*.

1. $Q = \{q_0\} \cup Q_1$.
   The states of N are the states of N_1 plus a new start state.
2. The state q_0 is the new start state.
3. $F = \{q_0\} \cup F_1$.
   The accept states are the old accept states plus the new start state.

4. Define δ so that for any $q \in Q$ and any $a \in \Sigma_\varepsilon$,

δ(q,a) for A*

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ and } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ and } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1 \text{ and } a = \varepsilon \\ \{q_1\} & q = q_0 \text{ and } a = \varepsilon \\ \emptyset & q = q_0 \text{ and } a \neq \varepsilon. \end{cases}$$

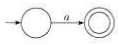
LEMMA 1.55

If a language is described by a regular expression, then it is regular.

PROOF IDEA Say that we have a regular expression R describing some language A. We show how to convert R into an NFA recognizing A. By Corollary 1.40, if an NFA recognizes A then A is regular.

PROOF Let's convert R into an NFA N. We consider the six cases in the formal definition of regular expressions.

1. $R = a$ for some a in Σ. Then $L(R) = \{a\}$, and the following NFA recognizes $L(R)$.



   Note that this machine fits the definition of an NFA but not that of a DFA because it has some states with no exiting arrow for each possible input symbol. Of course, we could have presented an equivalent DFA here but an NFA is all we need for now, and it is easier to describe.
   Formally, $N = (\{q_1, q_2\}, \Sigma, \delta, q_1, \{q_2\})$, where we describe δ by saying that $\delta(q_1, a) = \{q_2\}$ and that $\delta(r, b) = \emptyset$ for $r \neq q_1$ or $b \neq a$.
2. $R = \varepsilon$. Then $L(R) = \{\varepsilon\}$, and the following NFA recognizes $L(R)$.

   Formally, $N = (\{q_1\}, \Sigma, \delta, q_1, \{q_1\})$, where $\delta(r, b) = \emptyset$ for any r and b.
3. $R = \emptyset$. Then $L(R) = \emptyset$, and the following NFA recognizes $L(R)$.

   Formally, $N = (\{q\}, \Sigma, \delta, q, \emptyset)$, where $\delta(r, b) = \emptyset$ for any r and b.
4. $R = R_1 \cup R_2$.
5. $R = R_1 \circ R_2$.
6. $R = R_1^*$.

For the last three cases we use the constructions given in the proofs that the class of regular languages is closed under the regular operations. In other words, we construct the NFA for R from the NFAs for R_1 and R_2 (or just R_1 in case 6) and the appropriate closure construction.