Appendix B: The Theory

33 Working with Logical Formulas

Boolean formulas describe circuits. First-order logic formulas encode software specifications and robot plans. We need efficient and correct techniques for manipulating them. In this chapter, we present some fundamental theoretical results that serve as the basis for such techniques. We'll begin with Boolean formulas. Then we'll consider the extension of some of the Boolean ideas to first-order logic.

33.1 Working with Boolean Formulas: Normal Forms, Resolution and OBDDs

In this section we discuss three issues that may arise when working with Boolean (propositional) formulas:

- conversion of an arbitrary Boolean formula into a more restricted form (a normal form),
- boolean resolution, a proof by refutation technique, and
- efficient manipulation of Boolean formulas.

33.1.1 Normal Forms for Boolean Logic

Recall that a normal form for a set of data objects is a restricted syntactic form that simplifies one or more operations on the objects. When we use the term "normal form", we generally require that every object in the original set have some equivalent (with respect to the operations for which the normal form will be used) representation in the restricted form.

In this section we define three important normal forms for Boolean formulas and we prove that any Boolean formula has an equivalent formula in each of those normal forms. By "equivalent" in this case we mean logical equivalence. We begin with two definitions:

A ***literal*** in a Boolean formula is either an atomic proposition (a simple Boolean variable), or an atomic proposition preceded by a single negation symbol. So P, Q, and $\neg P$ are all literals. A ***positive literal*** is a literal that is not preceded by a negation symbol. A ***negative literal*** is a literal that is preceded by a negation symbol.

A ***clause*** is either a single literal or the disjunction of two or more literals. So P, $P \vee \neg P$, and $P \vee \neg Q \vee R \vee S$ are all clauses.

Conjunctive Normal Form

A well-formed formula (wff) of Boolean logic is in ***conjunctive normal form*** iff it is either a single clause or the conjunction of two or more clauses. So all of the following formulas are in conjunctive normal form:

P.
$P \vee \neg Q \vee R \vee S$.
$(P \vee \neg Q \vee R \vee S) \wedge (\neg P \vee \neg R)$.

The following formulas are not in conjunctive normal form:

$P \rightarrow Q$.
$\neg(P \vee \neg Q)$.
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$.

Theorem 33.1 Conjunctive Normal Form Theorem

Theorem: Given w, an arbitrary wff of Boolean logic, there exists a wff w' that is in conjunctive normal form and that is equivalent to w.

Proof: The proof is by construction. The following algorithm *conjunctiveBoolean* computes *w′* given *w*:

conjunctiveBoolean(*w*: wff of Boolean logic) =

1. Eliminate $\rightarrow$ and $\leftrightarrow$ from *w*, using the fact that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.
2. Reduce the scope of each $\neg$ to a single term, using the facts:
   - Double negation: $\neg(\neg P) = P$.
   - deMorgan's laws:
     - $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.
     - $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.
3. Convert *w* to a conjunction of clauses using the fact that both $\vee$ and $\wedge$ are associative and the fact that $\vee$ and $\wedge$ distribute over each other.

■

Example 33.1 Boolean Conjunctive Normal Form

Let *w* be the wff $P \rightarrow \neg(R \vee \neg Q)$. Then *w* can be converted to conjunctive normal form as follows:

- Step 1 produces $\neg P \vee \neg(R \vee \neg Q)$.
- Step 2 produces $\neg P \vee (\neg R \wedge Q)$.
- Step 3 produces $(\neg P \vee \neg R) \wedge (\neg P \vee Q)$.

Conjunctive normal form is useful as a basis for describing 3-conjunctive normal form, as we are about to do. It is also important because its extension to first-order logic formulas is useful, as we'll see below, in a variety of applications that require automatic theorem proving.

3-Conjunctive Normal Form

A well-formed formula (wff) of Boolean logic is in ***3-conjunctive normal form*** (***3-CNF***) iff it is in conjunctive normal form and each clause contains exactly three literals. So the following formulas are in 3-conjunctive normal form:

$(\neg Q \vee R \vee S)$.
$(\neg Q \vee R \vee S) \wedge (\neg P \vee \neg R \vee \neg Q)$.

3-conjunctive normal form is important because it allows us to define 3-SAT = {*w* : *w* is a wff in Boolean logic, *w* is in 3-conjunctive normal form and *w* is satisfiable}. 3-SAT is important because it is NP-complete and reduction from it can often be used to show that a new language is also NP-complete.

Theorem 33.2 3-Conjunctive Normal Form Theorem

Theorem: Given a Boolean wff *w* in conjunctive normal form, there exists an algorithm that constructs a new wff *w′* that is in 3-conjunctive normal form and that is satisfiable iff *w* is.

Proof: The following algorithm *3-conjunctiveBoolean* computes *w′* given *w*:

3-conjunctiveBoolean(*w*: wff of Boolean logic) =

1. If, in *w*, there are any clauses with more than three literals, split them apart, add additional variables as necessary, and form a conjunction of the resulting clauses. Specifically, if $n > 3$ and there is a clause of the following form:

   $(l_1 \vee l_2 \vee l_3 \vee \ldots \vee l_n)$,

   then it will be replaced by the following conjunction of n-2 clauses that can be constructed by introducing a set of literals Z_1 - Z_{n-3} that do not otherwise occur in the formula:

   $(l_1 \vee l_2 \vee Z_1) \wedge (\neg Z_1 \vee l_3 \vee Z_2) \wedge \ldots \wedge (\neg Z_{n-3} \vee l_{n-1} \vee l_n)$

2. If there is any clause with only one or two literals, replicate one of those literals once or twice so that there is a total of three literals in the clause.

In Exercise 28.4, we prove that *w′* is satisfiable iff *w* is. We also prove that *3-conjunctiveBoolean* runs in polynomial time.

■

Example 33.2 Boolean 3-Conjunctive Normal Form

Let *w* be the wff $(\neg P \vee \neg R) \wedge (\neg P \vee Q \vee R \vee S)$. We note that *w* is already in conjunctive normal form. It can be converted to 3-conjunctive normal form as follows:

- The first clause can be rewritten as $(\neg P \vee \neg R \vee \neg R)$.
- The second clause can be rewritten as $(\neg P \vee Q \vee Z_1) \wedge (\neg Z_1 \vee R \vee S)$.

So the result of converting *w* to 3-conjunctive normal form is:

$$(\neg P \vee \neg R \vee \neg R) \wedge (\neg P \vee Q \vee Z_1) \wedge (\neg Z_1 \vee R \vee S).$$

Disjunctive Normal Form

We now consider an alternative normal form in which conjunctions of literals are connected by disjunction (rather than the other way around). A well-formed formula (wff) of Boolean logic is in ***disjunctive normal form*** iff it is the disjunction of one or more disjuncts, each of which is either a single literal or the conjunction of two or more literals. All of P, $\neg P \wedge \neg R$, and $P \wedge \neg Q \wedge R \wedge S$ are disjuncts, and all of the following formulas are in disjunctive normal form:

P.
$P \vee \neg Q \vee R \vee S$.
$(P \wedge \neg Q \wedge R \wedge S) \vee (\neg P \wedge \neg R)$.

> Disjunctive normal form is the basis for a convenient notation for writing queries against relational databases ℂ 690.

Theorem 33.3 Disjunctive Normal Form Theorem

Theorem: Given *w*, an arbitrary wff of Boolean logic, there exists a wff *w′* that is in disjunctive normal form and that is equivalent to *w*.

Proof: The proof is by a construction similar to the one used to prove Theorem 33.1. The following algorithm *disjunctiveBoolean* computes *w′* given *w*:

disjunctiveBoolean(*w*: wff of Boolean logic) =
1. Eliminate $\rightarrow$ and $\leftrightarrow$ from *w*, using the fact that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.
2. Reduce the scope of each $\neg$ in *w* to a single atomic proposition using the facts:
   - Double negation: $\neg(\neg P) = P$.
   - deMorgan's laws:
     - $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.
     - $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.
3. Convert *w* to a disjunction of disjuncts using the fact that both $\vee$ and $\wedge$ are associative and the fact that $\vee$ and $\wedge$ distribute over each other.

■

Example 33.3 Boolean Disjunctive Normal Form

Let w be the wff $P \wedge (Q \rightarrow \neg(R \wedge T))$. Then w can be converted to disjunctive normal form as follows:

- Step 1 produces: $P \wedge (\neg Q \vee \neg(R \wedge T))$.
- Step 2 produces: $P \wedge (\neg Q \vee \neg R \vee \neg T)$.
- Step 3 produces: $(P \wedge \neg Q) \vee (P \wedge \neg R) \vee (P \wedge \neg T)$.

33.1.2 Boolean Resolution

Two of the most important operations on Boolean formulas are:

1) Satisfiability checking: given a wff *ST*, is it satisfiable or not? Recall that a wff is satisfiable iff it is true for at least one assignment of truth values to the variables it contains.

2) Theorem proving: given a set of axioms *A* and a wff *ST*, does *A* entail *ST*? Recall that *A* entails *Q* iff, whenever all of the wffs in *A* are true, *Q* is also true.

But note that *A* entails *ST* iff $A \wedge \neg ST$ is unsatisfiable. So an algorithm for determining unsatisfiability can also be exploited as a theorem prover. The technique that we present next is significant not just because it can be used to reason about Boolean logic formulas. More importantly, its extension to first-order logic launched the field of automatic theorem proving.

Resolution: The Inference Rule

The name "resolution" is used both for an inference rule and a theorem-proving technique that is based on that rule. We first describe the inference rule. Let Q, $\neg Q$, P and R be wffs. Then define:

- ***Resolution***: From the premises: $(P \vee Q)$ and $(R \vee \neg Q)$,
  Conclude: $(P \vee R)$.

The soundness of the resolution rule is based on the following observation: Assume that both $(P \vee Q)$ and $(R \vee \neg Q)$ are *True*. Then:

- If Q is *True*, R must be *True*.
- If $\neg Q$ *True*, P must be *True*.

Since either Q or $\neg Q$ must be *True*, $P \vee R$ must be *True*. To prove resolution's soundness, it suffices to write out its truth table. We leave that as an exercise.

Resolution: The Algorithm

We next present a theorem-proving technique called resolution. It relies on the inference rule called resolution that we just defined. The core of the prover is an algorithm that detects unsatisfiability. So a resolution proof is a proof by contradiction (often called refutation). A resolution proof of a statement *ST*, given a set of axioms *A*, is a demonstration that $A \wedge \neg ST$ is unsatisfiable. If $\neg ST$ cannot be true given A, *ST* must be.

The resolution procedure takes as its input a list of clauses. So, before it can be used, we must convert the axioms in *A* to such a list. We do that as follows:

1. Convert each formula in *A* to conjunctive normal form.
2. Build *L*, a list of the clauses that are constructed in step 1.

Example 33.4 Making a List of Clauses

Suppose that we are given the set A of axioms as shown in column 1. We convert each axiom to conjunctive normal form, as shown in column 2:

Given Axioms:	***Converted to Conjunctive Normal Form:***
P.	P.
$(P \wedge Q) \rightarrow R$.	$\neg P \vee \neg Q \vee R$.
$(S \vee T) \rightarrow Q$.	$(\neg S \vee Q) \wedge (\neg T \vee Q)$.
T.	T.

Then the list L of clauses constructed in this process is: P, $\neg P \vee \neg Q \vee R$, $\neg S \vee Q$, $\neg T \vee Q$, and T.

To prove that a formula ST is entailed by A, we construct the formula $\neg ST$, convert it to conjunctive normal form, and add all of the resulting clauses to the list of clauses produced from A. Then resolution, which we describe next, can begin.

A pair of ***complementary literals*** is a pair of literals that are not mutually satisfiable. So two literals are complementary iff one is positive, one is negative, and they contain the same propositional symbol. For example, Q and $\neg Q$ are complementary literals. We'll say that two clauses C_1 and C_2 contain a pair of complementary literals iff C_1 contains one element of the pair and C_2 contains the other. For example, the clauses $(P \vee Q \vee \neg R)$ and $(T \vee \neg Q)$ contain the complementary literals Q and $\neg Q$.

Consider a pair of clauses that contain a pair of complementary literals, which, without loss of generality, we'll call Q and $\neg Q$. So we might have $C_1 = R_1 \vee R_2 \vee \ldots \vee R_j \vee Q$ and $C_2 = S_1 \vee S_2 \vee \ldots \vee S_k \vee \neg Q$. Given C_1 and C_2, resolution (the inference rule) allows us to conclude $R_1 \vee R_2 \vee \ldots \vee R_j \vee S_1 \vee S_2 \vee \ldots \vee S_k$. When we apply the resolution rule in this way, we'll say that we have ***resolved*** the ***parents***, C_1 and C_2, to generate a new clause, which we'll call the ***resolvent***.

The resolution algorithm proceeds in a sequence of steps. At each step it chooses from L two clauses that contain complementary literals. It resolves those two clauses together to create a new clause, the resolvent, which it adds to L. If any step generates an unsatisfiable clause, then a contradiction has been found. For historical reasons, the ***empty clause*** is commonly called *nil*, the name given to an empty list in ***Lisp***, the language in which many resolution provers have been built. The empty clause is unsatisfiable since it contains no literals that can be made *True*. So if it is ever generated, the resolution procedure halts and reports that, since adding $\neg ST$ to A has led to a contradiction, ST is a theorem given A.

We'll describe Lisp and illustrate its use for symbolic reasoning, including theorem proving, in ℂ 671.

We can state the algorithm as follows:

resolve-Boolean(A: set of axioms in conjunctive normal form, ST: a wff to be proven) =

1. Construct L, the list of clauses from A.
2. Negate ST, convert the result to conjunctive normal form, and add the resulting clauses to L.
3. Until either the empty clause (*nil*) is generated or no progress is being made do:
   3.1. Choose from L two clauses that contain a pair of complementary literals. Call them the parent clauses.
   3.2. Resolve the parent clauses together. The resulting clause, called the resolvent, will be the disjunction of all the literals in both parent clauses except for one pair of complementary literals.
   3.3. If the resolvent is not *nil* and is not already in L, add it to L.
4. If *nil* was generated, a contradiction has been found. Return success. ST must be true.
5. If *nil* was not generated and there was nothing left to do, return failure.

Example 33.5 Boolean Resolution

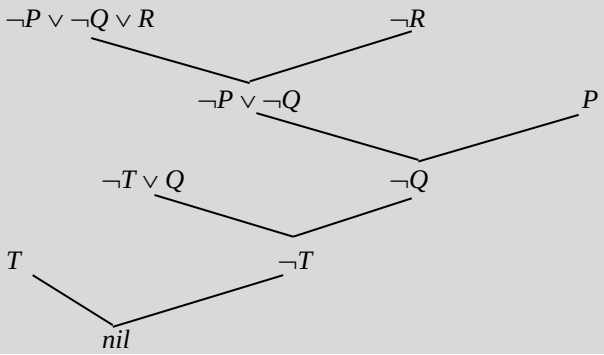
Given the axioms that we presented in Example 33.4, prove R. The axioms, and the clauses they generate are:

Given Axioms:	***Generate the Clauses:***
P.	P.
$(P \wedge Q) \rightarrow R$.	$\neg P \vee \neg Q \vee R$.
$(S \vee T) \rightarrow Q$.	$\neg S \vee Q$.
	$\neg T \vee Q$.
T.	T.

We negate R. The result is already in conjunctive normal form, so we simply add it to the list of clauses:

$\neg R$.

We illustrate the resolution process by connecting each pair of parent clauses to the resolvent that they produce:



In the simple example that we just did, *Resolve-Boolean* found a proof without trying any unnecessary steps. In general, though, it conducts a search through a space of possible resolvents. Its efficiency can be affected by the choice of parents in step 3.1. In particular, the following strategies may be useful:

- ***Unit preference***: All other things being equal, choose one parent that consists of just a single clause. Then the resolvent will be one clause shorter than the other parent and thus one clause closer to being the empty clause.
- ***Set-of-support***: Begin by identifying some subset S of L with the property that we can prove that any contradiction must rely on at least one clause from S. For example, if we assume that the set of axioms is consistent, then every contradiction must rely on at least one clause from $\neg ST$. So we could choose S to be just the clauses in $\neg ST$. Then, in every resolution step, choose at least one parent from S and then add the resolvent to S.

Resolve-Boolean's efficiency can also be affected by optimizing step 3.3. One way to do that is based on the observation that, if the resolvent is subsumed by some clause already in L, adding it to L puts the process no closer to finding a contradiction. It should simply be discarded. For example, if P is already in L, it makes no sense to add $P \vee P$ or $P \vee Q$. At the extreme, if the resolvent is a tautology, it is subsumed by everything. So adding it to L puts the process no closer to finding a contradiction. It should simply be discarded. For example, it never makes sense to add a clause such as $P \vee \neg P$.

It is possible to prove that the procedure *resolve-Boolean* is sound. It is also possible to prove that, as long as *resolve-Boolean* systematically explores the entire space of possible resolutions, it is ***refutation-complete***. By that we mean that, if $A \wedge \neg ST$ is unsatisfiable, *resolve-Boolean* will generate *nil* and thus discover the contradiction.

But it is important to keep in mind the complexity results that we present in Chapter 28. We prove, as Theorem 28.16, that the language SAT = {<w> : w is a wff in Boolean logic and w is satisfiable} is NP-complete. No polynomial-time algorithm for deciding it is known and it is unlikely that one exists. Unsatisfiability checking appears to be even harder since unsatisfiability, unlike satisfiability, cannot be verified just by checking one set of assignments of values

to the propositional symbols. As we see in Section 28.7, the language UNSAT = {<*w*> : *w* is a wff in Boolean logic and *w* is not satisfiable} is in co-NP, i.e., it is the complement of a language in NP. But it is thought not to be in NP, much less in P. There are ways to improve the performance of *resolve-Boolean* in many cases. But, in the worst case, the time it requires grows exponentially with the number of clauses in $A \wedge \neg ST$.

33.1.3 Efficient SAT Solvers and Ordered Binary Decision Diagrams

Satisfiability checking plays an important role in many applications 💻, including the design and analysis of digital circuits, the use of model checking to verify properties of programs, and the planning algorithms that determine the behavior of robots and other intelligent systems. While solving SAT in the general case remains hard, substantial research on the development of efficient satisfiability checkers (or SAT solvers) has led to the development of practical systems that work very well. In this section, we'll describe one technique that plays an important role in many efficient SAT solvers. What we'll do is to describe a new normal form for Boolean formulas. Its advantage is that it often produces a compact representation that can be exploited efficiently.

For many applications, we will find it useful to think of a Boolean formula *P* as a binary function of its inputs, so we'll use that notation in the rest of this example, rather than the wff notation that we introduced in 𝔄 556. So let *f* be a binary function of any number of variables. We'll encode *True* as 1 and *False* as 0.

One straightforward way to represent *f* is as a truth table, as we did in 𝔄 556. An alternative is as an ordered binary decision tree. In any such tree, each non-terminal node corresponds to a variable and each terminal node corresponds to the output of the function along the path that reached that node. From each nonterminal node there are two edges, one (which we'll draw to the left, with a dashed line) corresponds to the case where the value of the variable at the parent node is 0. The other (which we'll draw to the right, with a solid line) corresponds to the case where the value of the variable at the parent node is 1. To define such a tree for a binary function *f*, we begin by defining a total ordering ($v_1 < v_2 < \ldots < v_n$) on the *n* variables that represent the inputs to *f*. Any ordering will work, but the efficiency of the modified structure that we will present below may depend on the ordering that has been chosen. Given an ordering, we can draw the tree with v_1 at the root, v_2 at the next level, and so forth.

As an example, consider the function $f_1(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge x_3$. We can represent f_1 , as shown in Figure 33.1, as either a truth table or as a binary decision tree (where the tree is built using the variable ordering ($x_1 < x_2 < x_3$).

x_1	x_2	x_3	f_1
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

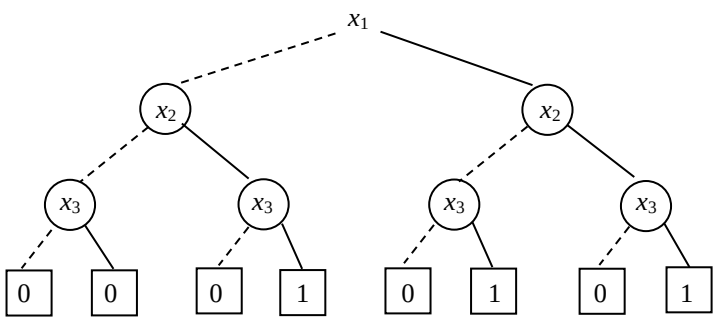


Figure 33.1 Representing a function as a truth table or a binary decision tree

The size of both the truth table representation and the binary decision tree for a function *f* of *n* variables is $\mathcal{O}(2^n)$. Any program *M* that reasons about *f* by manipulating either of those representations (assuming it must consider all of *f*) will consume at least $\mathcal{O}(2^n)$ space and $\mathcal{O}(2^n)$ time. So $timereq(M) \in \Omega(2^n)$ and $spacereq(M) \in \Omega(2^n)$. If we could reduce the size of the representation, it might be possible to reduce both the time and space requirements of any program that uses it.

If we choose the decision tree representation, it is often possible to perform such a reduction. We can convert the tree into a directed acyclic graph, called an ***ordered binary decision diagram*** or ***OBDD*** 💻. OBDDs, along with algorithms for manipulating them, were introduced in [Bryant 1986]. Our discussion of them is modeled after [Bryant 1992] and the examples we show here were taken from that paper.

We can optimize an OBDD by guaranteeing that none of its subtrees occurs more than once. Starting from the bottom, we will collapse all instances of a duplicate subtree into a single one. We'll then adjust the links into that unique tree appropriately. So we begin by creating only two terminal nodes, one labeled 0 and the other labeled 1. Then we'll move upward collapsing subtrees whenever possible. In the tree we just drew for f_1, for example, observe that the subtree whose root is x_3, whose left branch is the terminal 0 and whose right branch is the terminal 1 occurs three times. So the three can be collapsed into one. After collapsing them, we get the diagram shown in Figure 33.2 (a). At this point, notice the two nodes shown with double circles. Each of them has the property that its two outgoing edges both go to the same place. In essence, the value of the variable at that node has no effect on the value that f_1 returns. So the node itself can be eliminated. Doing that, we get the diagram shown in Figure 33.2 (b).

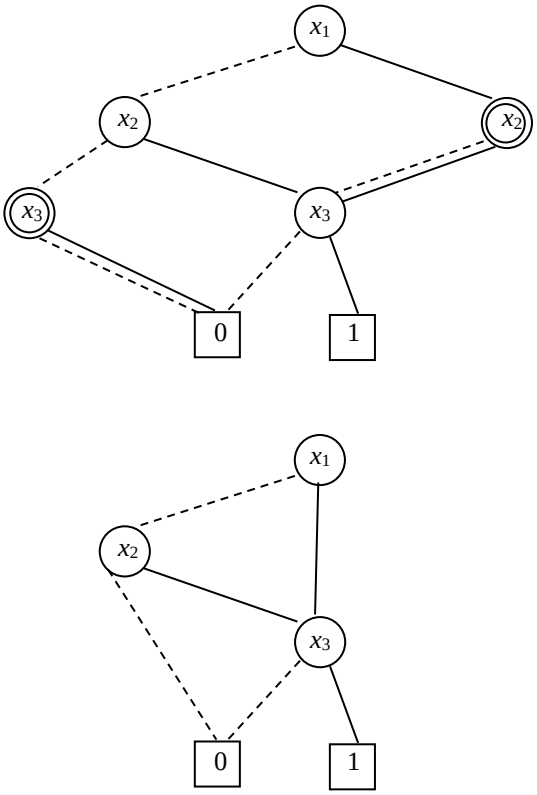


Figure 33.2 Collapsing nodes to get an efficient OBDD

The process we just described can be executed by the following function *createOBDD*:

createOBDDfromtree(*d*: ordered binary decision tree) =

1. Eliminate redundant terminal nodes by creating a single node for each label and redirecting edges as necessary.
2. Until one pass is made during which no reductions occurred do:
   2.1. Eliminate redundant nonterminal nodes (i.e., duplicated subtrees) by collapsing them into one and redirecting edges as necessary.
   2.2. Eliminate redundant tests by erasing any node whose two output edges go to the same place. Redirect edges as necessary.

This process will create a maximally reduced OBDD, by which we mean that there is no smaller one that describes the same function and that considers the variables in the same order. It is common to reserve the term OBDD for such maximally reduced structures. Given a particular ordering ($v_1 < v_2 < \ldots < v_n$) on the n variables that represent the inputs to some function f, any two OBDDs for f will be isomorphic to each other (i.e., the OBDD for f is unique up to the order in which the edges are drawn). Thus the OBDD structure is a canonical form for the representation of Boolean functions, given a particular variable ordering.

Since the OBDD for a function is unique up to isomorphism, some operations on it can be performed in constant time. For example, a function f corresponds to a valid wff (i.e., one that is a tautology) iff its OBDD is identical to the one shown in Figure 33.3 (a). A function f corresponds to a satisfiable wff iff its OBDD is not identical to the one shown in Figure 33.3 (b).

(a)

(b)

Figure 33.3 Exploiting canonical forms

If the only way to build a reduced OBDD were to start with a decision tree and reduce it by applying *createOBDDfromtree*, it would not be practical to work with functions of a large number of variables, even if the reduced OBDD were of manageable size. Fortunately, it is possible 💻 to build a reduced OBDD directly, without starting with a complete decision tree.

The size of the OBDD that can be built for a function f may depend critically on the order that we impose on f's inputs. For example, in the original decision tree that we built for f_1 above, we considered the inputs in the order x_1, x_2, x_3. We could have produced a slightly smaller OBDD (one with one fewer edge) if we had instead used the order x_3, x_1, x_2. We leave doing that as an exercise.

In some cases though, the effect of variable ordering is much more significant. Particularly in many cases of practical interest, in which there are systematic relationships within clusters of variables, it is possible to build a maximally reduced OBDD that is substantially smaller than the original decision tree. Consider the binary function:

$$f_2(a_1, b_1, a_2, b_2, a_3, b_3) = (a_1 \wedge b_1) \vee (a_2 \wedge b_2) \vee (a_3 \wedge b_3).$$

We'll consider two different variable orderings and the OBDDs that can be created for them. The first ordering, shown in Figure 33.4 (a), respects the relationship between the a, b pairs. The second, shown in Figure 33.4 (b), does not and pays a price.

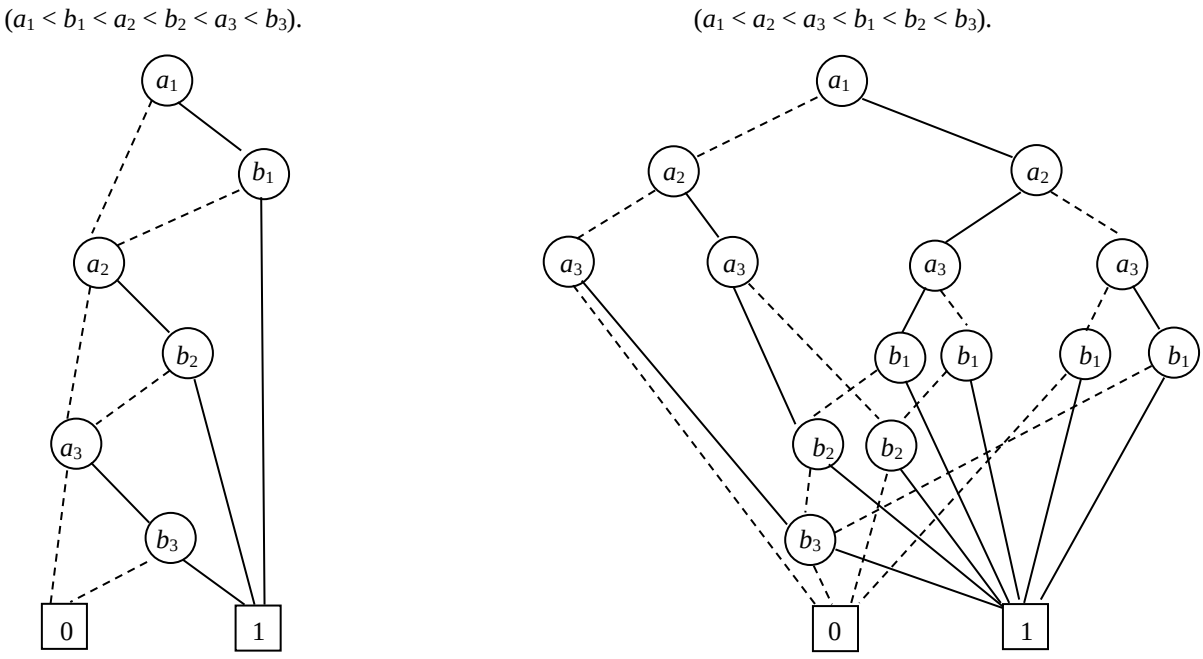


Figure 33.4 The order of the variables matters

Fortunately, for many classes of important problems there exist heuristics 💻 that find the variable orderings that make small structures possible. Unfortunately, however, there are problems for which no small OBDD exists. For example, consider a circuit that implements binary multiplication. Let *f* be the Boolean function corresponding to either of the two middle digits of the result of an *n*-bit multiplication. The size of any OBDD for *f* grows exponentially with *n*.

Programs that solve problems for which small OBDDs exist may have manageable requirements for both time and space. In particular, it is known that most common operations on OBDDs can be done in time that is $\mathcal{O}(nm)$, where *n* and *m* are sizes of the input OBDDs. So the OBDD structure improves the expected performance (with respect to both time and space) of many algorithms on many practical problems.

> Model checkers based on OBDDs are routinely used to prove properties of systems whose state description contains 10^{20} states. ℂ 679.

However, because small OBDDs do not exist for all problems, the structure does not change the worst-case complexity of those problems. Theorem 28.5.2 (the Cook-Levin Theorem) tells us that Boolean satisfiability is NP-complete. No polynomial algorithm for solving it for all cases is known. So, if we can impose no constraints on the form of the input, the worst case time complexity of any algorithm is likely to be $\mathcal{O}(2^n)$. While there is no proof that it is not possible to do better than that, it appears unlikely that we can.

33.2 Working with First-Order Formulas: Clause Form and Resolution

We can extend to first-order logic (FOL) the normal forms and the resolution theorem-proving procedure that we defined for Boolean logic in the last section.

33.2.1 Clause Form

Suppose that we want to build a first-order logic theorem prover that we can use as the basis for a practical reasoning system. One of the first things that we observe is that the standard first-order language (the one that we defined in 𝔄 558) allows quantifiers and connectors to be embedded in arbitrary ways.

Example 33.6 A Fact About Marcus

Consider the following sentence *F*:

$$\forall x\ ((Roman(x) \wedge Know(x, Marcus)) \rightarrow (Hate(x, Caesar) \vee \forall y\ (\exists z\ (Hate(y, z)) \rightarrow Thinkcrazy(x, y))))).$$

F says that any Roman who knows Marcus either hates Caesar or thinks that anyone who hates anyone is crazy. So if we knew that Paulus was a Roman who knew Marcus and who didn't hate Caesar, we could use *F* to conclude that Paulus thinks that anyone who hates anyone is crazy. Or, if we knew that Paulus was a Roman who knew Marcus, and that Augustus hates Flavius but Paulus doesn't think Augustus is crazy, then we could use *F* to conclude that Paulus hates Caesar. Or, if we knew that Paulus knows Marcus, doesn't hate Caesar, and doesn't think that Augustus, who hates Flavius is crazy, then we could use *F* to conclude that Paulus is not a Roman.

Each of the arguments that we have just described requires a different way of matching the other facts we already know against the fact about Marcus's friends. We'd like one technique that works for all of them.

One approach to solving this problem is to exploit the idea of a normal form, just as we did in dealing with Boolean logic formulas. In particular, we can extend the notions of conjunctive and disjunctive normal forms, to first-order logic. Now we must be concerned both with the structure of the logical connectors (just as we were for Boolean logic) as well as the structure of the quantifiers and variables. The motivation for the definition of the normal forms we are about to describe is the need to build theorem-proving programs. The syntax for an arbitrary sentence in first-order logic allows a great deal of flexibility, making it hard to write programs that can reason with all the facts that they may be given.

A sentence in first-order logic is in ***prenex normal form*** iff it is of the form:

<*quantifier list*> <*matrix*>,

where <*quantifier list*> is a list of quantified variables and <*matrix*> is quantifier-free.

Example 33.7 Prenex Normal Form

$\forall x\ (\exists y\ ((P(x) \wedge Q(y)) \rightarrow \forall z\ (R(x, y, z)))$ is not in prenex normal form.
$\forall x\ \exists y\ \forall z\ (P(x) \wedge Q(y)) \rightarrow R(x, y, z)$ is in prenex normal form. Its matrix is $(P(x) \wedge Q(y)) \rightarrow R(x, y, z)$.

Any sentence can be converted to an equivalent sentence in prenex normal form by the following procedure:

1. If necessary, rename the variables so that each quantifier binds a lexically distinct variable.
2. Move all the quantifiers to the left, without changing their relative order.

We define the terms literal, clause, and conjunctive normal form for sentences in first-order logic analogously to the way they were defined for Boolean logic:

- A ***literal*** is either a single predicate symbol, along with its argument list, or it is such a predicate preceded by a single negation symbol. So $P(x, f(y))$ and $\neg Q(x, f(y), 2)$ are literals. A ***positive literal*** is a literal that is not preceded by a negation symbol. A ***negative literal*** is a literal that is preceded by a negation symbol.
- A ***clause*** is either a single literal or the disjunction of two or more literals.
- A sentence in first-order logic is in ***conjunctive normal form*** iff its matrix is either a single clause or the conjunction of two or more clauses.

A ***ground instance*** is a first-order logic expression that contains no variables. So, for example, *Major-of*(*Sandy*, *Math*) is a ground instance, but $\forall x\ (\exists y\ ((Major\text{-}of(x, y)))$ is not.

A sentence in first-order logic is in ***clause form*** iff:

- it has been converted to prenex normal form,
- its quantifier list contains only universal quantifiers,
- its quantifier list is no longer explicitly represented,
- it is in conjunctive normal form, and
- there are no variable names that appear in more than one clause. This last condition is important because there will no longer be explicit quantifiers to delimit the scope of the variables. The only way to tell one variable from another will be by their names.

Example 33.8 Clause Form

The following sentence is not in clause form:

$$\forall x\,(P(x) \rightarrow Q(x)) \wedge \forall y\,(S(y)).$$

When it is converted to prenex normal form, we get:

$$\forall x\,\forall y\,(P(x) \rightarrow Q(x)) \wedge S(y).$$

Then, when it is converted to clause form, we get the conjunction of two clauses:

$$(\neg P(x) \vee Q(x)) \wedge S(y).$$

We are going to use clause form as the basis for a first-order, resolution-based proof procedure analogous to the Boolean procedure that we defined in the last section. To do that, we need to be able to convert an arbitrary first-order sentence w into a new sentence w' such that w' is in clause form and w' is unsatisfiable iff w is. In the proof of the next theorem, we provide an algorithm that does this conversion. All of the steps are straightforward except the one that eliminates existential quantifiers, so we'll discuss it briefly before we present the algorithm.

Let *Mother-of*(y, x) be true whenever y is the mother of x. Consider the sentence $\forall x\,(\exists y\,((\textit{Mother-of}(y, x)))$. Everyone has a mother. There is not a single individual who is the mother of everyone. But, given a value for x, some mother exists. We can eliminate the existentially quantified variable y using a technique called ***Skolemization***, based on an idea due to Thoralf Skolem [Skolem 1928]. We replace y by a function of x. We know nothing about that function. (We don't know, for example, that it is computable.) We know only that it exists. So we assign the function a name that is not already being used, say f_1. Then:

$\forall x\,(\exists y\,((\textit{Mother-of}(y, x)))$ becomes $\forall x\,((\textit{Mother-of}(f_1(x), x))$.

Multiple existential quantifiers in the same sentence can be handled similarly, but it is important that a new function be created for each existential quantifier. For example, consider the predicate *Student-data*(x, y, z) that is true iff student x enrolled at date y and has major z. Then:

$\forall x\,(\exists y\,(\exists z\,(\textit{Student-data}(x, y, z))))$ becomes $\forall x\,(\textit{Student-data}(x, f_2(x), f_3(x)))$.

The function $f_2(x)$ produces a date and the function $f_3(x)$ produces a major. Now consider the predicate *Sum*(x, y, z), that is true iff the sum of x and y is z. Then:

$\forall x\,(\forall y\,(\exists z\,(\textit{Sum}(x, y, z))))$ becomes $\forall x\,(\forall y\,(\textit{Sum}(x, y, f_4(x, y))))$.

In this case, the value of z that must exist (and be produced by f_4) is a function of both x and y. More generally, if an existentially quantified variable occurs inside the scope of n universally quantified variables, it can be replaced by a function of n arguments corresponding to those n variables. In the simple case in which an existentially quantified variable occurs inside the scope of no universally quantified variables, it can be replaced by a constant (i.e., a function of no arguments). So:

$\exists x\ (Student(x))$ becomes $Student(f_4)$.

The functions that are introduced in this way are called ***Skolem functions*** and ***Skolem constants***.

Skolemization plays a key role in theorem-proving systems (particularly resolution-based ones) because the Skolemization of a sentence w is unsatisfiable iff w is. But note that we have not said that a Skolemization is necessarily equivalent to the original sentence from which it was derived. Consider the simple sentence, $\exists x\ (P(x))$. It can be Skolemized as $P(f_1)$. But now observe that:

- $\exists x\ (P(x)) \rightarrow \exists x\ (P(x))$ is valid (i.e., it is a tautology). It is true in all interpretations.
- $\exists x\ (P(x)) \rightarrow P(f_1)$ is satisfiable since it is *True* if $P(f_1)$ is *True*. But it is not valid, since it is *False* if P is true for some value of x that is different from f_1 but *False* for f_1.

So $\exists x\ (P(x))$ and $P(f_1)$ are not logically equivalent.

The proof of the Clause Form Theorem, which we state next, exploits Skolemization, in combination with standard logical identities, as the basis of an algorithm that converts any first-order sentence w into another sentence w' that is in clause form and that is unsatisfiable iff w is.

Theorem 33.4 Clause Form Theorem

Theorem: Given w, a sentence in first-order logic, there exists a clause form representation w' such that w' is unsatisfiable iff w is.

Proof: The proof is by construction. The following algorithm *converttoclauseform* computes a clause form representation of w:

converttoclauseform(w: first-order sentence) =

1. Eliminate $\rightarrow$ and $\leftrightarrow$, using the fact that $P \rightarrow Q$ is equivalent to $\neg P \vee Q$.

2. Reduce the scope of each $\neg$ to a single term, using the facts:
   - Double negation: $\neg(\neg P) = P$.
   - deMorgan's laws:
     - $\neg(P \wedge Q) \equiv (\neg P \vee \neg Q)$.
     - $\neg(P \vee Q) \equiv (\neg P \wedge \neg Q)$.
   - Quantifier exchange:
     - $\neg\forall x\ (P(x)) \equiv \exists x\ (\neg P(x))$.
     - $\neg\exists x\ (P(x)) \equiv \forall x\ (\neg P(x))$.

3. Standardize apart the variables so that each quantifier binds a unique variable. For example, given the sentence:

   $\forall x\ (P(x)) \vee \forall x\ (Q(x))$,

   the variables can be standardized apart to produce:

   $\forall x\ (P(x)) \vee \forall y\ (Q(y))$.

4. Move all quantifiers to the left without changing their relative order. At this point, the sentence is in prenex normal form.

5. Eliminate existential quantifiers via Skolemization, as described above.

6. Drop the prefix since all remaining quantifiers are universal.

7. Convert the matrix to a conjunction of clauses by using the fact that both $\vee$ and $\wedge$ are associative and the fact that $\vee$ and $\wedge$ distribute over each other.

8. Standardize apart the variables so that no variable occurs in more than one clause.

■

Example 33.9 Converting the Marcus Fact to Clause Form

We now return to *F*, the statement about Marcus's friends that we introduced in Example 33.6:

$$\forall x ((Roman(x) \wedge Know(x, Marcus)) \rightarrow (Hate(x, Caesar) \vee \forall y (\exists z (Hate(y, z)) \rightarrow Thinkcrazy(x, y)))).$$

We convert *F* to clause form as follows:

Step 1: Eliminate $\rightarrow$. This step produces:

$$\forall x (\neg(Roman(x) \wedge Know(x, Marcus)) \vee (Hate(x, Caesar) \vee \forall y (\neg\exists z (Hate(y, z)) \vee Thinkcrazy(x, y)))).$$

Step 2: Reduce the scope of $\neg$. This step produces: (Notice that the existential quantifier disappears here.)

$$\forall x (\neg Roman(x) \vee \neg Know(x, Marcus) \vee (Hate(x, Caesar) \vee \forall y (\forall z (\neg Hate(y, z)) \vee Thinkcrazy(x, y)))).$$

Steps 3 and 4: Standardize apart and shift the quantifiers to the left. This step produces:

$$\forall x \forall y \forall z (\neg Roman(x) \vee \neg Know(x, Marcus) \vee (Hate(x, Caesar) \vee \neg Hate(y, z) \vee Thinkcrazy(x, y))).$$

Steps 5 – 8: These last steps produce:

$$\neg Roman(x) \vee \neg Know(x, Marcus) \vee Hate(x, Caesar) \vee \neg Hate(y, z) \vee Thinkcrazy(x, y).$$

Example 33.10 Handling Existential Quantifiers and Standardizing Apart

We convert the following sentence to clause form:

$$\forall x (Person(x) \rightarrow (\exists y (Mother\text{-}of(y, x)) \wedge \exists y (Father\text{-}of(y, x)))).$$

Step 1: Eliminate $\rightarrow$. This step produces:

$$\forall x (\neg Person(x) \vee (\exists y (Mother\text{-}of(y, x)) \wedge \exists y (Father\text{-}of(y, x)))).$$

Step 2: Reduce the scope of $\neg$. This step is not necessary.

Step 3: Standardize apart the variables so that each quantifier binds a unique variable.

$$\forall x (\neg Person(x) \vee (\exists y_1 (Mother\text{-}of(y_1, x)) \wedge \exists y_2 (Father\text{-}of(y_2, x)))).$$

Step 4: Move all quantifiers to the left without changing their relative order.

$$\forall x \exists y_1 \exists y_2 (\neg Person(x) \vee (Mother\text{-}of(y_1, x) \wedge Father\text{-}of(y_2, x))).$$

Step 5: Eliminate existential quantifiers via Skolemization.

$$\forall x\ (\neg Person(x) \vee (Mother\text{-}of(f_1(x), x) \wedge Father\text{-}of(f_2(x), x))).$$

Step 6: Drop the prefix since all remaining quantifiers are universal.

$$\neg Person(x) \vee (Mother\text{-}of(f_1(x), x) \wedge Father\text{-}of(f_2(x), x)).$$

Step 7: Convert the matrix to a conjunction of clauses.

$$(\neg Person(x) \vee Mother\text{-}of(f_1(x), x)) \wedge (\neg Person(x) \vee Father\text{-}of(f_2(x), x)).$$

Step 8: Standardize apart the variables so that no variable occurs in more than one clause.

$$(\neg Person(x_1) \vee Mother\text{-}of(f_1(x_1), x_1)) \wedge (\neg Person(x_2) \vee Father\text{-}of(f_2(x_2), x_2)).$$

Now the two clauses can be treated as independent clauses, regardless of the fact that they were derived from the same original sentence.

The design of a theorem prover can be simplified if all of the inputs to the theorem prover have been converted to clause form.

Example 33.11 Using the Marcus Fact in a Proof

We now return again to *F*, the statement about Marcus's friends that we introduced in Example 33.6:

$$\forall x\ ((Roman(x) \wedge Know(x, Marcus)) \rightarrow (Hate(x, Caesar) \vee \forall y\ (\exists z\ (Hate(y, z)) \rightarrow Thinkcrazy(x, y)))).$$

When we convert this statement to clause form, we get, as we showed in the last example, the formula that we will call F_C:

$$(F_C) \qquad \neg Roman(x) \vee \neg Know(x, Marcus) \vee Hate(x, Caesar) \vee \neg Hate(y, z) \vee Thinkcrazy(x, y).$$

In its original form, *F* is not obviously a way to prove that someone isn't a Roman. But, in clause form it is easy to use for that purpose. Suppose we add the following facts:

Know(*Paulus*, *Marcus*).
¬*Hate*(*Paulus*, *Caesar*).
Hate(*Augustus*, *Flavius*).
¬*Thinkcrazy*(*Paulus*, *Augustus*).

We can now prove that Paulus is not a Roman. Paulus knows Marcus, doesn't hate Caesar, and doesn't think that Augustus, who hates Flavius is crazy. The general statement about Marcus's friends must hold for all values of *x*. In the case of Paulus, we've ruled out four of the five literals that could make it true. The one that remains is ¬*Roman*(*Paulus*). Note that to implement the reasoning that we just did, we need a way to match literals like *Know*(*Paulus*, *Marcus*) and ¬*Know*(*x*, *Marcus*). We'll present unification, a technique for doing that, in the next section.

33.2.2 First-Order Logic Resolution

First-order logic is undecidable. We stated that result as Theorem 22.4: The language $FOL_{theorem}$ = {<*A*, *w*> : *A* is a decidable set of axioms in first-order logic, *w* is a sentence in first-order logic, and *w* is entailed by *A*} is not in D. As a proof, we sketched Turing's proof. So there is no algorithm to *decide* whether or not a statement is a theorem. But, as we showed in Theorem 22.3, the language $FOL_{theorem}$ is semidecidable by an algorithm that constructs a lexicographic enumeration of the valid proofs given *A*. Given a statement *w*, that algorithm will *discover* a proof if

one exists. To make theorem-proving useful in practical problem domains, however, we need techniques that are substantially more efficient than the exhaustive enumeration method. Fortunately, such techniques exist. And finding even better ones remains an active area of research. Keep in mind, however, that every first-order logic theorem prover has the limitation that, if asked to prove a nontheorem, it may not be able to tell that no proof exists.

In this section we describe one important proof technique: the extension to first-order logic of the resolution algorithm that we presented for Boolean logic in 𝔅 609. First-order resolution was introduced in [Robinson 1965] and has served, since then, as the basis for several generations of automatic theorem-proving programs. It is sound (i.e., it can prove only theorems that are entailed by the axioms it is given). And it is refutation-complete, by which we mean the following: Given a set of axioms *A* and a sentence *ST*, if *ST* is a theorem then $A \wedge \neg ST$ will derive a contradiction and the resolution algorithm, assuming it uses a systematic strategy for exploring the space of possible resolution steps, will (eventually) find it. We note, however, that first-order resolution is not complete in the sense that there may be theorems that will not be generated by any resolution step.

> First-order logic resolution is the basis for logic programming languages such as Prolog, ℭ 762. It has played a key role in the evolution of the field of artificial intelligence, ℭ 760. It has been used to solve problems in domains ranging from program verification, ℭ 679, to medical reasoning. One noteworthy application in mathematics was the proof of the Robbins Algebra Conjecture, which had outwitted mathematicians for 60 years 💻.

A first-order logic resolution theorem prover 💻 works in essentially the same way a Boolean one does. It begins with *A*, a set of axioms that have been converted to clause form. To prove a statement *ST*, it negates *ST*, converts the result to clause form, and adds it to *A*. Then, at each resolution step, it chooses two parent clauses that contain complementary literals, resolves the two clauses together, creates a resolvent, and adds it to *A*. If the unsatisfiable clause *nil* is ever generated, a contradiction has been found.

Unification

The only new issue that we must face is how to handle variables and functions. In particular, what does it now mean to say that two literals are complementary? As before, two literals are complementary iff they are inconsistent. Two literals are inconsistent iff one of them is positive, one of them is negative (i.e., begins with ¬), they both contain the same predicate, and they are about intersecting sets of individuals. In other words, two literals are inconsistent, and thus complementary, iff they make conflicting claims about at least one individual. To check for this, resolution exploits a matching process called ***unification***. Unification takes as its arguments two literals, each with any leading ¬ removed. It will return *Fail* if the two literals do not match, either because their predicates are different or because it is not certain that the intersection of the sets of individuals that they are about is not empty. It will succeed if they do match. And, in that case, it will return a list of substitutions that describes how one literal was transformed into the other so that they match and the nonempty intersection was found.

When are two literals about intersecting sets of individuals? Recall that all clause variables are universally quantified. So the domains of any two variables overlap. For example, $P(x)$ and $\neg P(y)$ are complementary literals. One says *P* is true of everyone; the other says that *P* is false of everyone. The domain of any one variable necessarily includes all specific values. So $P(x)$ and $\neg P(Marcus)$ are complementary since *P* cannot be true of everyone but not true of *Marcus*. $P(Caesar)$ and $\neg P(Marcus)$ are not complementary since *P* can be true of *Caesar* but not of *Marcus*. $P(f(Marcus))$ and $\neg P(f(Marcus))$ are complementary, but $P(f(Marcus))$ and $\neg P(f(Caesar))$ are not. While it is possible that $f(Marcus)$ and $f(Caesar)$ refer to the same individual, it is not certain that they do. Unification will handle functions by recursively invoking itself. It will check that function symbols match in the same way that it checks that predicate symbols match.

If the same variable occurs more than once in a clause, any substitutions that are made to it must be made consistently to all of its occurrences. So the unification algorithm must, each time it makes a substitution, apply it to the remainder of both clauses before it can continue. For example, consider unifying $Know(x, x)$ (everyone knows him/her/itself) and $Know(Marcus, Marcus)$. Unification will match the first *x* with the first *Marcus*, substituting *Marcus* for *x*. It will then substitute *Marcus* for the second occurrence of *x* before it continues. It will succeed when it next matches *Marcus* with *Marcus*. But now consider unifying $Know(x, x)$ and $Know(Marcus, Caesar)$. The second literal,

Know(*Marcus*, *Caesar*), is not about knowing oneself. Unification will fail in this case because it will substitute *Marcus* for *x*, apply that substitution to the second *x*, and then fail to match *Marcus* and *Caesar*.

Each invocation of the unification procedure will return either the special value *Fail* or a list of substitutions. We will write each list as ($subst_1$, $subst_2$, …). We will write each substitution as sub_1/sub_2, meaning that sub_1 is to be written in place of sub_2. If unification succeeds without performing any substitutions, the substitution list will be *nil* (the empty list). We are now ready to state the unification procedure:

unify-for-resolution(lit_1, lit_2: variables, constants, function expressions or positive literals) =

1. If either lit_1 or lit_2 is a variable or a constant then:
   1.1. Case (checking the conditions in order and executing only the first one that matches):
   - lit_1 and lit_2 are identical: return *nil*. /* Succeed with no substitution required.
   - lit_1 is a variable that occurs in lit_2: return *Fail*. /* These two cases implement the
   - lit_2 is a variable that occurs in lit_1: return *Fail*. occur check. See note below.
   - lit_1 is a variable: return (lit_2/lit_1).
   - lit_2 is a variable: return (lit_1/lit_2).
   - otherwise: return *Fail*. /* Two different constants do not match.
2. If the initial predicate or function symbols of lit_1 and lit_2 are not the same, return *Fail*.
3. If lit_1 and lit_2 do not have the same number of arguments, return *Fail*.
4. *substitution-list* = *nil*.
5. For $i = 1$ to the number of arguments of lit_1 do:
   5.1. Let *S* be the result of invoking *unify-for-resolution* on the i^{th} argument of lit_1 and of lit_2.
   5.2. If *S* contains *Fail*, return *Fail*.
   5.3. If *S* is not equal to *nil* then:
      5.3.1. Apply *S* to the remainder of both lit_1 and lit_2.
      5.3.2. Append *S* to *substitution-list*.
6. Return *substitution-list*.

In step 1.1, *unify-for resolution* performs a check called the ***occur check***. Consider attempting to unify $f(x)$ with $f(g(x))$. Without the occur check, the function expression $g(x)$ could be unified with x, producing the substitution $g(x)/x$. But now there is no way to apply that substitution consistently to the new occurrence of x. In this case, the problem might simply not be noticed, with the consequence that any theorem prover that uses the result of the unification may produce an unsound inference. The problem is even clearer in the following case: Consider attempting to unify $f(x, x)$ with $f(g(x), g(x))$. Without the occur check, $g(x)$ could be unified with x, again producing the substitution $g(x)/x$. But now, the unification algorithm must apply that substitution to the remainder of the two clauses before it can continue. So (, x) and (, $g(x)$) become (, $g(x)$) and (, $g(g(x))$). But now it has to substitute again, and so forth. Unfortunately, the occur check is expensive. So some theorem-proving programs omit it and take a chance.

Example 33.12 Unification Examples

We show the result of *unify-for-resolution* in each of the following cases:

	Inputs	*Result*	*Substitution*
[1]	*Roman*(*x*), *Roman*(*Paulus*).	Succeed	*Paulus*/*x*.
[2]	*Roman*(*x*), *Ancient*(*Paulus*).	Fail	
[3]	*Roman*(*father-of*(*Marcus*)), *Roman*(*x*).	Succeed	*father-of*(*Marcus*)/*x*.
[4]	*Roman*(*father-of*(*Marcus*)), *Roman*(*Flavius*).	Fail	
[5]	*Roman*(*x*), *Roman*(*y*).	Succeed	*x*/*y*.
[6]	*Roman*(*father-of*(*x*)), *Roman*(*x*).	Fail (fails occur check)	
[7]	*Likes*(*x, y*), *Likes*(*Flavius, Marcus*).	Succeed	*Flavius*/*x*, *Marcus*/*y*.

Notice that *unify-for-resolution* is conservative. It returns a match only if it is certain that its two arguments describe intersecting sets of individuals. For example, *father-of*(*Marcus*) and *Flavius* may (but do not necessarily) refer to the same individual. Without additional information, we do not want resolution to assert a contradiction between *Roman*(*father-of*(*Marcus*)) and ¬*Roman*(*Flavius*).

One other property of *unify-for-resolution* is worth noting: Consider unifying *Roman*(*x*) with *Roman*(*y*). The algorithm as given here returns the substitution *x*/*y*. We could, equivalently, have defined it so that it would return *y*/*x*. That choice was arbitrary. But we could also have defined it so that it returned the substitution *Marcus*/*x*, *Marcus*/*y*. That substitution effectively converts a statement that had applied to all individuals into one that applies only to *Marcus*. This restricted statement is entailed by the more general one, so a theorem prover that exploited such a match would still be sound. But proving statements would become more difficult because resolution is going to look for contradictions. General statements lead to more contradictions than specific ones do. So we can maximize the performance of a resolution theorem prover if we exploit a unification algorithm that returns what we will call a ***most general unifier***, namely a substitution with the property that no other substitution that preserves soundness imposes fewer restrictions on the values of the variables. The algorithm that we have presented always returns a most general unifier.

We can now define complementary literals analogously to the way they were defined for Boolean logic. Two literals are ***complementary literals*** iff they unify and one of them is the negation of the other. So, for example ¬*Roman*(*x*) and *Roman*(*Paulus*) are complementary literals. Just as in the case of Boolean logic, the conjunction of a pair of complementary literals is unsatisfiable.

Resolution: The Algorithm

Now that we have a way to identify complementary literals, we can define the resolution algorithm for first-order logic. It works the same way that resolution works in Boolean logic except that two new things need to happen after each resolution step:

- The substitution that was produced when the two complementary literals were unified must be applied to the resolvent clause. To see why this is important, consider resolving $P(x) \vee Q(x)$ with $\neg Q(Marcus)$. The first clause says that, for all values of *x*, at least one of *P* or *Q* must be true. The second one says that, in the specific case of *Marcus*, *Q* is not true. From those two clauses, it follows that, in the specific case of *Marcus*, *P* must be true. It does not follow that *P* must be true of all values of *x*. The result of unifying *Q*(*x*) with ¬*Q*(*Marcus*) is the substitution *Marcus*/*x*. So we can construct the result of resolving these two clauses by first building the clause that is the disjunction of all literals except the two complementary ones. That gives us *P*(*x*). We then apply the substitution *Marcus*/*x* to that, which produces *P*(*Marcus*).

- We must guarantee that the variable names in the resolvent are distinct from all the variable names that already occur in any of the clauses in *L*. If this is not done, it is possible that later resolution steps will treat two different

variables that just happen to have the same name as though they were a single variable to which consistent substitutions must be applied. For a concrete example of the problem that this can cause, see Exercise 33.8).

resolve-FOL(*A*: set of axioms in clause form, *ST*: a statement to be proven) =

1. Construct *L*, the list of clauses from *A*.
2. Rename all variables in *ST* so that they do not conflict with any variables in *L*.
3. Negate *ST*, convert the result to clause form, and add the resulting clauses to *L*.
4. Until either the empty clause (called *nil*) is generated or no progress is being made do:
   4.1. Choose from *L* two clauses that contain a pair *CL* of complementary literals. Call them the parent clauses.
   4.2. Resolve the parent clauses together to produce a new clause called the resolvent:
      4.2.1. Initially, let the resolvent be the disjunction of all of the literals in both parent clauses except for the two literals in *CL*.
      4.2.2. Apply to all of the literals in the resolvent the substitution that was constructed when the literals in *CL* were unified.
      4.2.3. Rename all of the variables in the resolvent so that they do not conflict with any of the variables in *L*.
   4.3. If the resolvent is not *nil* and is not already in *L*, add it to *L*.
5. If *nil* was generated, a contradiction has been found. Return success. *ST* must be true.
6. If *nil* was not generated and there was nothing left to do, return failure. *ST* may or may not be true. But no proof of *ST* has been found.

Example 33.13 FOL Resolution

Assume that we are given the following axioms (in clause form):

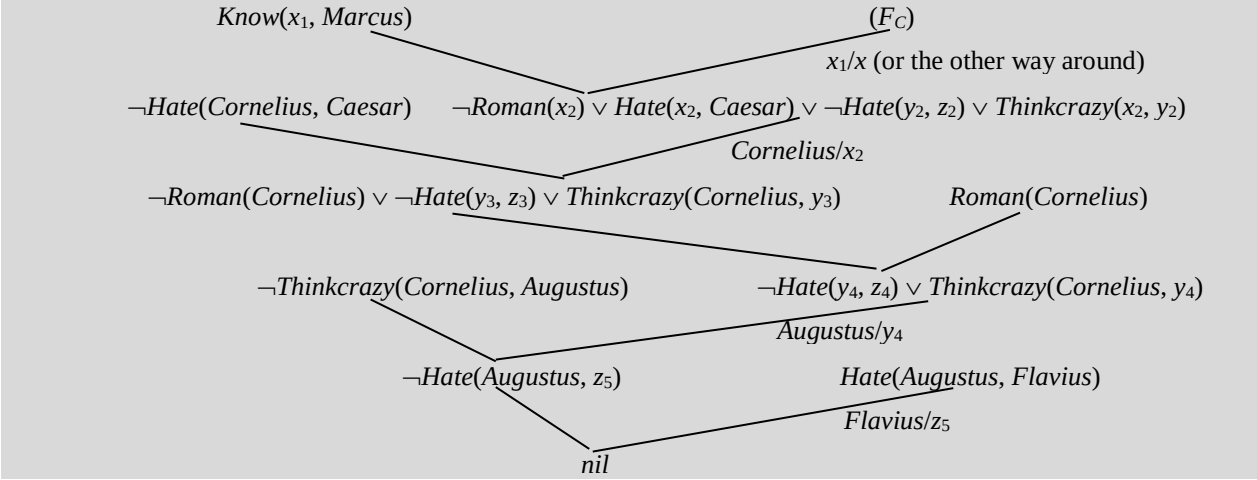
(F_C) $\neg Roman(x) \vee \neg Know(x, Marcus) \vee Hate(x, Caesar) \vee \neg Hate(y, z) \vee Thinkcrazy(x, y)$.
$\neg Hate(Cornelius, Caesar)$.
$Hate(Augustus, Flavius)$.
$\neg Thinkcrazy(Cornelius, Augustus)$.
$Roman(Cornelius)$.

We will use resolution to prove $\exists x\ (\neg Know(x, Marcus))$.

We negate $\exists x\ (\neg Knows(x, Marcus))$, producing $\neg(\exists x\ (\neg Know(x, Marcus)))$ or $\forall x\ (Know(x, Marcus))$. Converting this to clause form (and standardizing apart the variables), we get:

$Know(x_1, Marcus)$.

Resolution can now proceed as follows. (But note that this is just one path it could pursue. It could choose parent clauses in a different order.) We show, at each resolution step, the substitution that the unification process produced. Note also that the variable names have been standardized apart at each step.



Resolve-FOL must typically search a space of possible resolution paths. As we saw in the case of *resolve-Boolean*, the efficiency of the search process can be affected by the order in which parent clauses (and complementary literals within them) are chosen. In particular both the unit preference strategy and the set-of-support strategy may be useful. The efficiency of the process can also be improved by failing to insert into *L* those resolvents that put the process no closer to finding a contradiction (for example because they are subsumed by clauses that are already present). Other optimizations are also possible 💻. Even with them, however, the size of the search space that must be explored may grow too fast to make resolution a practical solution for many kinds of problems. One way to cut down the space is to limit the form of the clauses that are allowed. For example, logic programming languages, such as Prolog, work only with Horn clauses, which may have at most one positive literal. See ℂ 762 for a brief introduction to Prolog and some of its applications.

Resolution: The Inference Rule

Recall that, in our discussion of resolution in Boolean logic, we pointed out that resolution is both an inference rule and an algorithm for checking unsatisfiability. The same is true for resolution in first-order logic. Using the definitions of unification and substitution that we have just provided, we can state resolution as an inference rule. Let Q, $\neg Q$, P_1, P_2, … , P_n and R_1, R_2, … , R_n be literals, let *substitution-list* be the substitution that is returned by *unify-for-resolution* when it unifies Q and $\neg Q$, and let *substitute*(*clause*, *substitution-list*) be the result of applying *substitution-list* to *clause*. Then define:

- ***Resolution***: From the premises: $(P_1 \vee P_2 \vee \ldots \vee P_n \vee Q)$ and $(R_1 \vee R_2 \vee \ldots \vee R_m \vee \neg Q)$,
  Conclude: *substitute*$((P_1 \vee P_2 \vee \ldots \vee P_n \vee R_1 \vee R_2 \vee \ldots \vee R_m)$, *substitution-list*$)$.

33.3 Exercises

1) Convert each of the following Boolean formulas to conjunctive normal form:
   a) $(a \wedge b) \rightarrow c$.
   b) $\neg(a \rightarrow (b \wedge c))$.
   c) $(a \vee b) \rightarrow (c \wedge d)$.
   d) $\neg(p \rightarrow \neg(q \vee (\neg r \wedge s)))$.

2) Convert each of the following Boolean formulas to 3-conjunctive normal form:
   a) $(a \vee b) \wedge (a \wedge \neg b \wedge \neg c \wedge d \wedge e)$.
   b) $\neg(a \rightarrow (b \wedge c))$.

3) Convert each of the following Boolean formulas to disjunctive normal form:
   a) $(a \vee b) \wedge (c \vee d)$.
   b) $(a \vee b) \rightarrow (c \wedge d)$.

4) Use a truth table to show that Boolean resolution is sound.

5) Use resolution to show that the following premises are inconsistent:

$a \vee \neg b \vee c$, $b \vee \neg d$, $\neg c \vee d$, $b \vee c \vee d$, $\neg a \vee \neg b$, and $\neg d \vee \neg b$.

6) Prove that the conclusion $b \wedge c$ follows from the premises: $a \rightarrow (c \vee d)$, $b \rightarrow a$, $d \rightarrow c$, and b.
   a) Convert the premises and the negation of the conclusion to conjunctive normal form.
   b) Use resolution to prove the conclusion.

7) Consider the Boolean function $f_1(x_1, x_2, x_3) = (x_1 \vee x_2) \wedge x_3$. that we used as an example in 𝔅 612. Show how f_1 can be converted to an OBDD using the variable ordering $(x_3 < x_1 < x_2)$.

8) In this problem, we consider the importance of standardizing apart the variables that occur in a first-order sentence in clause form. Assume that we are given a single axiom, $\forall x$ (*Likes*(*x*, *Ice cream*)). And we want to prove $\exists x$ (*Likes*(*Mikey*, *x*)). Use resolution to do this but don't standardize apart the two occurrences of *x*. What happens?

9) Begin with the following fact from Example 33.6:

   [1] $\forall x$ ((*Roman*(*x*) $\wedge$ *Know*(*x*, *Marcus*)) $\rightarrow$
   (*Hate*(*x*, *Caesar*) $\vee$ $\forall y$ ($\exists z$ (*Hate*(*y*, *z*)) $\rightarrow$ *Thinkcrazy*(*x*, *y*)))).

   Add the following facts:

   [2] $\forall x$ ((*Roman*(*x*) $\wedge$ *Gladiator*(*x*)) $\rightarrow$ *Know*(*x*, *Marcus*)).
   [3] *Roman*(Claudius).
   [4] $\neg\exists x$ (*Thinkcrazy*(*Claudius*, *x*)).
   [5] $\neg\exists x$ (*Hate*(*Claudius*, *x*)).
   [6] *Hate*(*Isaac*, *Caesar*).
   [7] $\forall x$ ((*Roman*(*x*) $\wedge$ *Famous*(*x*)) $\rightarrow$ (*Politician*(*x*) $\vee$ *Gladiator*(*x*))).
   [8] *Famous*(*Isaac*).
   [9] *Roman*(*Isaac*).
   [10] $\neg$*Know*(*Isaac*, *Marcus*).

   a) Convert each of these facts to clause form.
   b) Use resolution and this knowledge base to prove $\neg$*Gladiator*(*Claudius*).
   c) Use resolution and this knowledge base to prove *Politician*(*Isaac*).

10) In ℭ 762, we describe a restricted form of first-order resolution called SLD resolution. This problem explores an issue that arises in that discussion. In particular, we wish to show that SLD resolution is not refutation-complete for knowledge bases that are not in Horn clause form. Consider the following knowledge base *B* (that is not in Horn clause form):

   [1] $P(x_1) \vee Q(x_1)$.
   [2] $\neg P(x_2) \vee Q(x_2)$.
   [3] $P(x_3) \vee \neg Q(x_3)$.
   [4] $\neg P(x_4) \vee \neg Q(x_4)$.

   a) Use resolution to show that *B* is inconsistent (i.e., show that the empty clause *nil* can be derived).
   b) Show that SLD resolution cannot derive *nil* from *B*.

34 Part II: Finite State Machines and Regular Languages

In this chapter, we will do, in gory detail, one proof of the correctness of a construction algorithm.

Theorem 5.3 asserts that, given a nondeterministic FSM *M* that accepts some language *L*, there exists an equivalent deterministic FSM that accepts *L*. We proved this theorem by construction. We described the following algorithm:

ndfsmtodfsm(*M*: NDFSM) =
1. For each state *q* in *K* do:
   Compute *eps*(*q*). /* These values will be used below.
2. *s′* = *eps*(*s*)
3. Compute δ':
   3.1. *active-states* = {*s′*}. /* We will build a list of all states that are reachable from the start state. Each element of *active-states* is a set of states drawn from *K*.
   3.2. δ' = ∅.
   3.3. While there exists some element *Q* of *active-states* for which δ' has not yet been computed do:
      For each character *c* in Σ do:
         new-state = ∅.
         For each state *q* in *Q* do:
            For each state *p* such that (*q*, *c*, *p*) ∈ Δ do:
               new-state = *new-state* ∪ *eps*(*p*).
         Add the transition (*Q*, *c*, *new-state*) to δ'.
         If *new-state* ∉ *active-states* then insert it into *active-states*.
4. *K′* = *active-states.*
5. *A′* = {*Q* ∈ *K′* : *Q* ∩ *A* ≠ ∅ }.

From any NDFSM *M*, *ndfsmtodfsm* constructs a DFSM *M′*, which we claimed is both (1) deterministic and (2) equivalent to *M*. We prove those claims here.

Proving (1) is trivial. By the definition in step 3 of δ', we are guaranteed that δ' is defined for all reachable elements of *K′* and all possible input characters. Further, step 3 inserts a single value into δ' for each state, input pair, so *M′* is deterministic.

Next we must prove (2). In other words, we must prove that *M′* accepts a string *w* if and only if *M* accepts *w*. We constructed the transition function δ' of *M′* so that *M′* mimics an "all paths" simulation of *M*. We must now prove that that simulation returns the same result that *M* would. In particular, δ' defines each individual step of the behavior of *M′*. We must show that a sequence of steps of *M′* mimics the corresponding sequence of steps of *M* and then that the results of the two sequences are identical.

So we begin by proving the following lemma, which asserts that entire sequences of moves of *M′* behave as they should:

> ***Lemma***: Let *w* be any string in Σ*, let *p* and *q* be any states in *K*, and let *P* be any state in *K′*. Then:
>
> $(q, w) \vdash_M^* (p, \varepsilon)$ iff $(eps(q), w) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P$.
>
> In other words, if the original NDFSM *M* starts in state *q* and, after reading the string *w*, can land in state *p* (along at least one of its paths), then the new machine *M′* must behave as follows: when started in the state that corresponds to the set of states the original machine *M* could get to from *q* without consuming any input, *M′* reads the string *w* and lands in one of its new "set" states that contains *p*. Furthermore, because of the only-if part of the lemma, *M′* must end up in a "set" state that contains only states that *M* could get to from *q* after reading *w* and following any available ε-transitions.

To prove the lemma we must show that δ' has been defined so that the individual steps of *M'*, when taken together, do the right thing for an input string *w* of any length. Since we know what happens one step at a time, we will prove the lemma by induction on |*w*|.

We must first prove that the lemma is true for the base case, where |*w*| = 0 (i.e., *w* = ε). To do this, we actually have to do two proofs, one to establish it for the *if* part of the lemma, and the other to establish it for the *only if* part:

Base step, *if* part: prove that $(q, w) \vdash_M^* (p, \varepsilon)$ if $(eps(q), w) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P$. Or, turning it around to make it a little clearer:

$$[\ (eps(q), w) \vdash_{M'}^* (P, \varepsilon) \text{ and } p \in P\ ] \rightarrow [\ (q, w) \vdash_M^* (p, \varepsilon)\ ].$$

If |*w*| = 0, then, since *M'* contains no ε-transitions, *M'* makes no moves. So it must end in the same state it started in, namely *eps*(*q*). So *P* = *eps*(*q*). If *P* contains *p*, then $p \in eps(q)$. But, given our definition of the function *eps*, that means exactly that, in the original machine *M*, *p* is reachable from *q* just by following ε-transitions, which is exactly what we need to show.

Base step, *only if* part: we need to show:

$$[\ (q, w) \vdash_M^* (p, \varepsilon)\ ] \rightarrow [\ (eps(q), w) \vdash_{M'}^* (P, \varepsilon) \text{ and } p \in P\ ].$$

If |*w*| = 0 and the original machine *M* goes from *q* to *p* with only *w* as input, it must go from *q* to *p* following just ε-transitions. In other words $p \in eps(q)$. Now consider the new machine *M'*. It starts in *eps*(*q*), the set state that includes all the states that are reachable from *q* via ε transitions. Since *M'* contains no ε-transitions, it will make no moves at all if its input is ε. So it will halt in exactly the same state it started in, namely *eps*(*q*). So *P* = *eps*(*q*) and thus contains *p*.. So *M'* has halted in a set state that includes *p*, which is exactly what we needed to show.

Next we'll prove that, if the lemma is true for all strings *w* of length *k*, where $k \geq 0$, then it is true for all strings of length *k* + 1. Any string of length *k* + 1 must contain at least one character. So we can rewrite any such string as *zx*, where *x* is a single character and *z* is a string of length *k*. The way that *M* and *M'* process *z* will thus be covered by the induction hypothesis. We use the definition of δ', which specifies how each individual step of *M'* operates, to show that, assuming that the machines behave identically for the first *k* characters, they behave identically for the last character also and thus for the entire string of length *k* + 1. Recall the definition of δ':

$$\delta'(Q, c) = \cup\{eps(p) : \exists q \in Q\ ((q, c, p) \in \Delta)\}.$$

To prove the lemma, we must show a relationship between the behavior of:

The computation of the NDFSM *M*:	$(q, w) \vdash_M^* (p, \varepsilon)$, and
The computation of the DFSM *M'*:	$(eps(q), w) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P$.

Rewriting *w* as *zx*, we have:

The computation of the NDFSM *M*:	$(q, zx) \vdash_M^* (p, \varepsilon)$, and
The computation of the DFSM *M'*:	$(eps(q), zx) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P$.

Breaking each of these computations into two pieces, the processing of *z* followed by the processing of the single remaining character *x*, we have:

The computation of the NDFSM *M*:	$(q, zx) \vdash_M^* (s_i, x) \vdash_M (p, \varepsilon)$, and
The computation of the DFSM *M'*:	$(eps(q), zx) \vdash_{M'}^* (Q, x) \vdash_{M'}^* (P, \varepsilon)$ and $p \in P$.

In other words, after processing *z*, *M* will be in some set of states *S*, whose elements we'll write as s_i. *M'* will be in some "set" state that we will call *Q*. Again, we'll split the proof into two parts:

Induction step, *if* part: we must prove:

$$[\ (eps(q), zx) \vdash_{M'}^* (Q, x) \vdash_{M'}^* (P, \varepsilon) \text{ and } p \in P\ ] \rightarrow [\ (q, zx) \vdash_M^* (s_i, x) \vdash_M (p, \varepsilon)\ ].$$

If, after reading z, M' is in state Q, we know, from the induction hypothesis, that the original machine M, after reading z, must be in some set of states S and that Q is precisely that set. Now we just have to describe what happens at the last step when the two machines read x. If we have that M', starting in Q and reading x lands in P, then we know, from the definition of δ' above, that P contains precisely the states that M could land in after starting in any state in S and reading x. Thus if $p \in P$, p must be a state that M could land in if started in s_i on reading x.

Induction step, *only if* part: we must prove:

$$[\ (q, zx) \vdash_M^* (s_i, x) \vdash_M (p, \varepsilon)\ ] \rightarrow [\ \ (eps(q), zx) \vdash_{M'}^* (Q, x) \vdash_{M'}^* (P, \varepsilon) \text{ and } p \in P\ ].$$

By the induction hypothesis, we know that if M, after processing z, can reach some set of states S, then Q (the state M' is in after processing z) must contain precisely all the states in S. Knowing that, and our definition of δ', we know that from Q, reading x, M' must be in some set state P that contains precisely the states that M can reach starting in any of the states in S, reading x, and then following all ε transitions. So, after consuming w (i.e., zx), M', when started in $eps(q)$, must end up in a state P that contains all and only the states p that M, when started in q, could end up in.

Now that we have proved the lemma, we can complete the proof that M' is equivalent to M. Consider any string $w \in \Sigma^*$:

If $w \in L(M)$ (i.e., the original machine M accepts w) then the following two statements must be true:

1. The original machine M, when started in its start state, can consume w and end up in an accepting state. This must be true given the definition of what it means for a machine to accept a string.
2. $(eps(s), w) \vdash_{M'}^* (Q, \varepsilon)$ for some Q containing some $a \in A$. In other words, the new machine, when started in its start state, can consume w and end up in one of its accepting states. This follows from the lemma, which is more general and describes a computation from any state to any other. But if we use the lemma and let q equal s (i.e., M begins in its start state) and $p = a$ for some $a \in A$ (i.e., M ends in an accepting state), then we have that the new machine M', when started in its start state, $eps(s)$, will consume w and end in a state that contains a. But if M' does that, then it has ended up in one of its accepting states (by the definition of A' in step 5 of the algorithm). So M' accepts w (by the definition of what it means for a machine to accept a string).

If $w \notin L(M)$ (i.e., the original machine M does not accept w) then the following two statements must be true:

1. The original machine M, when started in its start state, will not be able to end up in an accepting state after reading w. This must be true given the definition of what it means for a machine to accept a string.
2. If $(eps(s), w) \vdash_{M'}^* (Q, \varepsilon)$, then Q contains no state $a \in A$. In other words, the new machine, when started in its start state, cannot consume w and end up in one of its accepting states. This follows directly from the lemma.

Thus M' accepts precisely the same set of strings that M does.

■

35 Part III: Context-Free Languages and PDAs

In this chapter, we will provide the proofs of three claims that we made introduced in Part III, during our discussion of the context-free languages.

35.1 Proof of the Greibach Normal Form Theorem

In this section, we prove the result that we stated as Theorem 11.2, namely that, given a context-free grammar G, there exists an equivalent Greibach normal form grammar G' such that $L(G') = L(G) - \{\varepsilon\}$.

Recall that a grammar $G = (V, \Sigma, R, S)$ is in Greibach normal form iff every rule in R has the form:

$X \rightarrow a\ \beta$, where $a \in \Sigma$ and $\beta \in (V - \Sigma)^*$.

So the following kinds of rules violate the Greibach normal form constraints:

- Epsilon productions, i.e., productions of the form $A \rightarrow \varepsilon$: given a grammar G, ε-rules can be removed by the procedure *removeEps* that we defined in Section 11.7.4. The resulting grammar G' will have the property that $L(G') = L(G) - \{\varepsilon\}$.
- Unit productions, i.e., productions of the form $A \rightarrow B$, where B is a single element of $V - \Sigma$: given a grammar G, unit productions can be removed by the procedure *removeUnits* that we defined in Section 11.8.3. The resulting grammar G' will have the property that $L(G') = L(G)$.
- Productions, such as $X \rightarrow A\mathtt{a}B$, whose right-hand sides have terminal symbols in positions other than the left-most. given a grammar G, these productions can be removed by the procedure *removeMixed* that we defined in Section 11.8.3. The resulting grammar G' will have the property that $L(G') = L(G)$. Note that *removeMixed* actually goes farther than we need to, since it removes all terminals except those that stand alone on a right-hand side. So it will rewrite the rule $X \rightarrow \mathtt{a}AB$, even though it is in Greibach normal form.
- Productions, such as $X \rightarrow AB$, whose right hand side begins with a nonterminal symbol: we must define a new procedure to handle these productions.

The process of converting a grammar to Chomsky normal form removes all rules in the first three of these classes. So the algorithm that we are about to present for converting a grammar G to Greibach normal form will begin by converting G to Chomsky normal form, using the algorithm that we presented in Section 11.8.3. Note, however, that Greibach normal form allows rules, such as $X \rightarrow \mathtt{a}A$ and $X \rightarrow \mathtt{a}ABCD$, that are not allowed in Chomsky normal form. So there exist more efficient Greibach normal form conversion algorithms than the one we are about to describe 💻.

Our algorithm will also exploit the following operations that we have described elsewhere:

- Rule substitution allows nonterminals to be replaced, in right-hand sides, by the strings that they can derive. Suppose that $G = (V, \Sigma, R, S)$ contains a rule r of the form $X \rightarrow \alpha Y \beta$, where α and β are elements of V^* and $Y \in (V - \Sigma)$. Let $Y \rightarrow \gamma_1 \mid \gamma_2 \mid \ldots \mid \gamma_n$ be all of G's Y rules. And let G' be the result of removing from R the rule r and replacing it by the rules $X \rightarrow \alpha\gamma_1\beta$, $X \rightarrow \alpha\gamma_2\beta$, …, $X \rightarrow \alpha\gamma_n\beta$. Then Theorem 11.3 tells us that $L(G') = L(G)$.

- The procedure *removeleftrecursion*, which we defined in Section 15.2.2 as part of our discussion of top-down parsing, removes direct left-recursion and replaces it by right-recursion. So, for example, if the A rules of G are $\{A \rightarrow A\mathtt{b}, A \rightarrow \mathtt{c}\}$, *removeleftrecursion* will replace those rules with the rules $\{A \rightarrow \mathtt{c}A', A \rightarrow \mathtt{c}, A' \rightarrow \mathtt{b}A', A' \rightarrow \mathtt{b}\}$. Note that the right-hand side of every rule that is introduced by *removeleftrecursion* begins with either a terminal symbol or an element of $(V - \Sigma)$. None of these right-hand sides begins with an introduced nonterminal (such as A').

Example 35.1 Using Substitution to Convert a Very Simple Grammar

To see how these procedures are used, consider the following grammar, which is in Chomsky normal form but not in Greibach normal form:

$S \rightarrow AB$
$A \rightarrow XY \mid \mathtt{c}$
$X \rightarrow \mathtt{a}$
$Y \rightarrow \mathtt{b}$
$B \rightarrow \mathtt{c}$

To convert this grammar to Greibach normal form, we:

1. Use rule substitution to replace the rule $S \rightarrow AB$ with the rules $S \rightarrow XYB$ and $S \rightarrow \mathtt{c}B$ (since A can derive XY and $\mathtt{c}$). The second of these new rules is in Greibach normal form.
2. Use rule substitution on the first of the new rules and replace it with the rule $S \rightarrow \mathtt{a}YB$ (since X can derive $\mathtt{a}$). This new rule is in Greibach normal form.
3. Use rule substitution to replace the rule $A \rightarrow XY$ with the rule $A \rightarrow \mathtt{a}Y$ (since X can derive $\mathtt{a}$). This new rule is in Greibach normal form.

Since the remaining three rules are already in Greibach normal form, the process ends with the grammar containing the rules $\{S \rightarrow \mathtt{a}YB, S \rightarrow \mathtt{c}B, A \rightarrow \mathtt{a}Y, X \rightarrow \mathtt{a}, Y \rightarrow \mathtt{b}, B \rightarrow \mathtt{c}\}$.

Example 35.2 Dealing with Left Recursion

But now consider the following grammar:

$S \rightarrow SA \mid BA$
$A \rightarrow \mathtt{a}$
$B \rightarrow \mathtt{b}$

The first rule is left-recursive. If we apply rule substitution to it, we get two new rules, $S \rightarrow SSA$ and $S \rightarrow BAA$. But now we still have a rule whose left-hand side begins with S. We can apply rule substitution again, but no matter how many times we apply it, we will get a new rule whose left-hand side begins with S. To solve this problem, we must exploit *removeleftrecursion* to eliminate direct left-recursion before we apply rule substitution. Doing that, we get:

$S \rightarrow BAS' \mid BA$
$A \rightarrow \mathtt{a}$
$B \rightarrow \mathtt{b}$
$S' \rightarrow AS' \mid A$

Now, to convert this grammar to Greibach normal form, we:

1. Use rule substitution to replace the rule $S \rightarrow BAS'$ with the rule $S \rightarrow \mathtt{b}AS'$. This new rule is in Greibach normal form.
2. Use rule substitution to replace the rule $S \rightarrow BA$ with the rule $S \rightarrow \mathtt{b}A$. This new rule is in Greibach normal form.
3. Use rule substitution to replace the rule $S' \rightarrow AS'$ with the rule $S' \rightarrow \mathtt{a}S'$. This new rule is in Greibach normal form.
4. Use rule substitution to replace the rule $S' \rightarrow A$ with the rule $S' \rightarrow \mathtt{a}$. This new rule is in Greibach normal form.

The remaining two rules are already in Greibach normal form, so the process terminates.

More realistic grammars typically contain more than a few nonterminals and those nonterminals may derive each other in arbitrary ways. To handle such grammars, we need a systematic way to organize the substitutions that will be performed.

So the conversion algorithm we will exploit is the following. It will return a new grammar it calls G_G.

converttoGreibach(*G*: CFG in Chomsky normal form) =

1. Choose an ordering of the nonterminals in *G*. Any ordering will work as long as the start symbol comes first. Let G_G initially be *G*.
2. Rewrite the rules of G_G so that each rule whose left-hand side is one of *G*'s original nonterminals is in one of the following two forms:
   - $X \rightarrow a\ \beta$, where $a \in \Sigma$ and $\beta \in (V - \Sigma)^*$ (in other words, the rule is in Greibach normal form), or
   - $X \rightarrow Y\ \beta$, where $Y \in V - \Sigma$ and *Y* occurs after *X* in the ordering defined in step 1.

   Call the constraint we have just described the ***rule-order constraint***. Note that, if any of *G*'s rules are directly left-recursive, this step will add some new rules whose left-hand sides are new nonterminals. We will not require that these new rules satisfy the rule-order constraint, since the new nonterminals are unnumbered. But note that no newly introduced nonterminal will occur as the first symbol in any rule's right-hand side.
3. Consider each of *G*'s original nonterminals, starting with the highest numbered one, and working backwards. For each such nonterminal *N*, perform substitutions on the rules in G_G so that the right-hand sides of all *N* rules begin with a terminal symbol.
4. Consider each nonterminal *N* that was introduced by *removeleftrecursion*. Perform substitutions on the rules of G_G so that the right hand sides of all *N* rules start with a terminal symbol.
5. Return G_G.

The grammar G_G that *converttoGreibach* returns will be in Greibach normal form. And $L(G_G) = L(G)$. We'll now describe how to perform steps 2, 3, and 4. Define A_k to be the k^{th} nonterminal, as defined in step 1.

Step 2: we will first rewrite all the A_1 rules so that they meet the rule-order constraint. Then we'll do the same for the A_2 rules, and so forth. For each *k*, as we begin to transform the rules for A_k, we assume that all rules for nonterminals numbered from 1 to *k*-1 already satisfy the rule-order constraint.

Any A_k rule whose right-hand side starts with a terminal symbol already satisfies the constraint and can be ignored. But we must consider all A_k rules of the form:

$$A_k \rightarrow A_j\beta$$

Group those rules into the following three cases and consider them in this order:

i. $j > k$: no action is required.

ii. $j < k$: replace the rule $A_k \rightarrow A_j\beta$ by the set of rules that results from substituting, for A_j, the right-hand sides of all the A_j rules. Since all A_j rules have already been transformed so that they satisfy the rule-order constraint, the right-hand sides of all A_j rules start with either terminal symbols or nonterminals numbered greater than *j*. They may still be numbered less than *k*, however. If any of them are, repeat the substitution process. Since the indices must increase by at least 1 each time, it will be necessary to do this no more than *k*-1 times.

iii. $j = k$: all such rules are of the form: $A_k \rightarrow A_k\beta$. They are directly left-recursive. Use *removeleftrecursion* to remove the left-recursion from all A_k rules. Recall that *removeleftrecursion* will create a new set of A_k rules. The right-hand side of each such rule will begin with a string that corresponds to the right-hand side of some nonrecursive A_k rule. But, as a result of handling all the rules in case 2, above, all of those right-hand sides must start with either a terminal symbol or a non-terminal symbol numbered above *k*. So all A_k rules now satisfy the rule-order constraint.

Example 35.3 Performing Step 2 of the Conversion Algorithm

We'll begin with the following grammar in Chomsky normal form:

$S \rightarrow SB \mid AB \mid \mathtt{d}$
$A \rightarrow SA \mid \mathtt{a}$
$B \rightarrow SA$

We'll order the three nonterminals *S*, *A*, *B*. So first we must rewrite the three *S* rules so that they satisfy the rule-order constraint. The second and third of them already do. But we must rewrite the first one, which is left-recursive. Using *removeleftrecursion*, we get the new grammar:

$S \rightarrow AB \mid ABS' \mid \mathtt{d} \mid \mathtt{d}S'$
$A \rightarrow SA \mid \mathtt{a}$
$B \rightarrow SA$
$S' \rightarrow B \mid BS'$

Now we consider the *A* rules. The second one starts with a terminal symbol, but the first one violates the rule-order constraint since *A* is numbered 2 and *S* is numbered 1. We use rule substitution and replace it with four new rules, one for each *S* rule. That produces the following set of *A* rules:

$A \rightarrow ABA \mid ABS'A \mid \mathtt{d}A \mid \mathtt{d}S'A \mid \mathtt{a}$

But now the first two of these are left-recursive. So we use *removeleftrecursion* and get the following set of *A* and *A'* rules. The *A* rules now satisfy the rule-order constraint:

$A \rightarrow \mathtt{d}A \mid \mathtt{d}AA' \mid \mathtt{d}S'A \mid \mathtt{d}S'AA' \mid \mathtt{a} \mid \mathtt{a}A'$
$A' \rightarrow BA \mid BAA' \mid BS'A \mid BS'AA'$

Finally, we consider the single *B* rule, $B \rightarrow SA$. It fails to satisfy the rule-order constraint since *B* is numbered 3 and *S* is numbered 1. We use rule substitution and replace it with four new rules, one for each *S* rule. That produces the following set of *B* rules:

$B \rightarrow ABA \mid ABS'A \mid \mathtt{d}A \mid \mathtt{d}S'A$

The first two of these fail to satisfy the rule-order constraint since *B* is numbered 3 and *A* is numbered 2. So we use rule substitution again. The first *B* rule is replaced by the rules:

$B \rightarrow \mathtt{d}ABA \mid \mathtt{d}AA'BA \mid \mathtt{d}S'ABA \mid \mathtt{d}S'AA'BA \mid \mathtt{a}BA \mid \mathtt{a}A'BA$

And the second *B* rule is replaced by the rules:

$B \rightarrow \mathtt{d}ABS'A \mid \mathtt{d}AA'BS'A \mid \mathtt{d}S'ABS'A \mid \mathtt{d}S'AA'BS'A \mid \mathtt{a}BS'A \mid \mathtt{a}A'BS'A$

At this point, the complete grammar is the following (where the *B* rules are broken up just for clarity):

$S \rightarrow AB \mid ABS' \mid \mathtt{d} \mid \mathtt{d}S'$
$A \rightarrow \mathtt{d}A \mid \mathtt{d}AA' \mid \mathtt{d}S'A \mid \mathtt{d}S'AA' \mid \mathtt{a} \mid \mathtt{a}A'$
$B \rightarrow \mathtt{d}A \mid \mathtt{d}S'A$
$B \rightarrow \mathtt{d}ABA \mid \mathtt{d}AA'BA \mid \mathtt{d}S'ABA \mid \mathtt{d}S'AA'BA \mid \mathtt{a}BA \mid \mathtt{a}A'BA$
$B \rightarrow \mathtt{d}ABS'A \mid \mathtt{d}AA'BS'A \mid \mathtt{d}S'ABS'A \mid \mathtt{d}S'AA'BS'A \mid \mathtt{a}BS'A \mid \mathtt{a}A'BS'A$
$S' \rightarrow B \mid BS'$
$A' \rightarrow BA \mid BAA' \mid BS'A \mid BS'AA'$

This grammar satisfies the rule-order constraint. But it is substantially larger and messier than the original grammar was. This is typical of what happens when a grammar is converted to Greibach normal form.

At the end of step 2, all rules whose left-hand sides contain any of G's original nonterminals satisfy the rule-order constraint. Note also that step 2 preserves the three properties initially established by conversion to Chomsky normal form: There are no ε-rules, there are no unit productions, and, in every right-hand side, all symbols after the first must be nonterminals.

Step 3: let n be the number of original nonterminals in G. Then A_n is the last of them (given the order from step 1). The right-hand sides of all A_n rules must begin with a terminal symbol. This must be true since there are no original nonterminals numbered higher than n. Now consider the A_{n-1} rules. Their right-hand sides must begin with a terminal symbol or A_n. Use substitution to replace all the rules whose right-hand sides start with A_n. After doing that, the right hand sides of all the A_{n-1} will all start with terminal symbols. Continue working backwards until the A_1 rules have been processed in this way. This step also preserves the three properties initially established by conversion to Chomsky normal form. So, at the end of this step, all rules whose left-hand sides contain any of G's original nonterminals are in Greibach normal form.

Step 4: the *removeleftrecursion* procedure introduces new nonterminal symbols and new rules with those symbols as their left-hand sides. So there will be new rules like $S' \rightarrow AS' \mid A$. The new nonterminals are independent of each other, so the right-hand sides of all of their rules consist only of terminals and original nonterminals. If r is one of those rules and r is not already in Greibach normal form then it is $N \rightarrow A_j\beta$ for some original nonterminal A_j. As a result of step 3, all A_j rules are already in Greibach normal form. So a single substitution for A_j will replace r by a set of N rules in Greibach normal form. This step preserves all of the properties that were true at the end of step 3. So, at the end of this step, G_G is in Greibach normal form.

Example 35.4 Performing Steps 3 and 4 of the Conversion

We'll continue with the grammar from Example 35.3. After step 2, it was:

$S \rightarrow AB \mid ABS' \mid \mathtt{d} \mid \mathtt{d}S'$
$A \rightarrow \mathtt{d}A \mid \mathtt{d}AA' \mid \mathtt{d}S'A \mid \mathtt{d}S'AA' \mid \mathtt{a} \mid \mathtt{a}A'$
$B \rightarrow \mathtt{d}A \mid \mathtt{d}S'A$
$B \rightarrow \mathtt{d}ABA \mid \mathtt{d}AA'BA \mid \mathtt{d}S'ABA \mid \mathtt{d}S'AA'BA \mid \mathtt{a}BA \mid \mathtt{a}A'BA$
$B \rightarrow \mathtt{d}ABS'A \mid \mathtt{d}AA'BS'A \mid \mathtt{d}S'ABS'A \mid \mathtt{d}S'AA'BS'A \mid \mathtt{a}BS'A \mid \mathtt{a}A'BS'A$
$S' \rightarrow B \mid BS'$
$A' \rightarrow BA \mid BAA' \mid BS'A \mid BS'AA'$

Step 3: all the B rules must be in Greibach normal form. It turns out that, in this example, the A rules are also. But then we must consider the S rules. The first two of them have right-hand sides that do not begin with terminal symbols. So they must be rewritten using substitution. After doing that, the complete set of S rules is:

$S \rightarrow \mathtt{d}AB \mid \mathtt{d}AA'B \mid \mathtt{d}S'AB \mid \mathtt{d}S'AA'B \mid \mathtt{a}B \mid \mathtt{a}A'B$
$S \rightarrow \mathtt{d}ABS' \mid \mathtt{d}AA'BS' \mid \mathtt{d}S'ABS' \mid \mathtt{d}S'AA'BS' \mid \mathtt{a}BS' \mid \mathtt{a}A'BS'$
$S \rightarrow \mathtt{d} \mid \mathtt{d}S'$

Step 4: we must use substitution on both of the S' rules. The two of them will be replaced by the following set of S' rules:

$S' \rightarrow \mathtt{d}A \mid \mathtt{d}S'A$
$S' \rightarrow \mathtt{d}ABA \mid \mathtt{d}AA'BA \mid \mathtt{d}S'ABA \mid \mathtt{d}S'AA'BA \mid \mathtt{a}BA \mid \mathtt{a}A'BA$
$S' \rightarrow \mathtt{d}ABS'A \mid \mathtt{d}AA'BS'A \mid \mathtt{d}S'ABS'A \mid \mathtt{d}S'AA'BS'A \mid \mathtt{a}BS'A \mid \mathtt{a}A'BS'A$
$S' \rightarrow \mathtt{d}AS' \mid \mathtt{d}S'AS'$
$S' \rightarrow \mathtt{d}ABAS' \mid \mathtt{d}AA'BAS' \mid \mathtt{d}S'ABAS' \mid \mathtt{d}S'AA'BAS' \mid \mathtt{a}BAS' \mid \mathtt{a}A'BAS'$
$S' \rightarrow \mathtt{d}ABS'AS' \mid \mathtt{d}AA'BS'AS' \mid \mathtt{d}S'ABS'AS' \mid \mathtt{d}S'AA'BS'AS' \mid \mathtt{a}BS'AS' \mid \mathtt{a}A'BS'AS'$

And similarly for the A' rules. We'll skip writing them all out. There are 14 (the number of B rules) · 4 (the number of A' rules) = 56 of them.

So the original, 6-rule grammar in Chomsky normal form becomes a 118-rule grammar in Greibach normal form.

Theorem 35.1 Greibach Normal Form

Theorem: Given a context-free grammar G, there exists an equivalent Greibach normal form grammar G_G such that $L(G_G) = L(G) - \{\varepsilon\}$.

Proof: The proof is by construction, using the algorithm *converttoGreibach* described above.

■

35.2 Proof that the Deterministic Context-Free Languages are Closed Under Complement

In this section, we prove the result that we stated as Theorem 13.10:

Theorem 35.2 Closure Under Complement

Theorem: The deterministic context-free languages are closed under complement.

Proof: The proof is by construction. The construction exploits techniques that we used to prove several other properties of the context-free languages, but now we must be careful to preserve the property that the PDA we are working with is deterministic.

If L is a deterministic context-free language over the alphabet Σ, then $L\$$ is accepted by some deterministic PDA $M = (K, \Sigma \cup \{\$\}, \Gamma, \Delta, s, A)$. We need to describe an algorithm that constructs a new deterministic PDA that accepts $(\neg L)\$$. The algorithm will proceed in two main steps:

1. Convert M to an equivalent PDA M'''' that is in a constrained form that we will call ***deterministic normal form***.
2. From M'''', build $M\#$ to accept $(\neg L)\$$.

The design of deterministic normal form is motivated by the observation that a deterministic PDA may fail to accept an input string w for any one of several reasons:

1. Its computation ends before it finishes reading w.
2. Its computation ends in an accepting state but the stack is not empty.
3. Its computation loops forever, following ε-transitions, without ever halting in either an accepting or a nonaccepting state.
4. Its computation ends in a nonaccepting state.

If we attempt to build $M\#$ by simply swapping the accepting and nonaccepting states of M, we will build a machine that correctly fails to accept every string that M would have accepted (i.e., every string in $L\$$). But it cannot be guaranteed to accept every string in $(\neg L)\$$. To do that, we must also address issues 1 – 3 above. Converting M to deterministic normal form will solve those problems since any deterministic PDA in restricted normal form will, on any input $w\$$:

- read all of w,
- empty its stack, and
- halt.

One additional problem is that we don't want to accept $\neg L(M)$. That includes strings that do not end in \$. We must accept only strings that do end in \$ and that are in $(\neg L)$\$.

Given a deterministic PDA M, we convert it into deterministic normal form in a sequence of steps, being careful, at each step, not to introduce nondeterminism.

In the first step, we will create M', which will contain two complete copies of M's states and transitions. M' will operate in the first copy until it reads the end-of-input symbol \$. After that, it will operate in the second copy. Call the states in the first copy the pre\$ states. Call the states in the second copy the post\$ states. If q is a pre\$ state, call the corresponding post\$ state q'. If q is an accepting state, then add q' to the set of accepting states and remove q from the set. If M contains the transition $((q, \$, \alpha), (p, \beta))$ and q is a pre\$ state, remove that transition and replace it with the transition $((q, \$, \alpha), (p', \beta))$. Now view M' as a directed graph but ignore the actual labels on the transitions. If there are states that are unreachable from the start state, delete them. If M was deterministic, then M' also is and $L(M') = L(M)$.

If M' ever follows a transition from a post\$ state that reads any input then it must not accept. So we can remove all such transitions without changing the language that is accepted. Remove them. Now all transitions out of post\$ states read no input. So they are one of the following:

- stack-ε-transitions: $((p, \varepsilon, \varepsilon), (q, \gamma))$ (nothing is popped), or
- stack-productive transitions: $((p, \varepsilon, \alpha), (q, \gamma))$, where $\alpha \in \Gamma^+$ (something is popped).

Next we remove all stack-ε-transitions from post\$ states. To construct an algorithm to do this, observe:

- Since M' is deterministic, if it contains the stack-ε-transition $((p, \varepsilon, \varepsilon), (q, \gamma))$ then it contains no other transitions from p.
- If $((p, \varepsilon, \varepsilon), (q, \gamma))$ ever plays a role in causing M' to accept a string then there must be a path from q that eventually reaches an accepting state and clears the stack.

So we can eliminate the stack-ε-transition $((p, \varepsilon, \varepsilon), (q, \gamma))$ as follows: First, if q is accepting, make p accepting. Next, delete $((p, \varepsilon, \varepsilon), (q, \gamma))$ and replace it by transitions that go directly from p to wherever q could go, skipping the move to q. So consider every transition $((q, \varepsilon, \alpha), (r, \beta))$. If $\alpha = \varepsilon$, then add the transition $((p, \varepsilon, \varepsilon), (r, \beta\gamma))$. Otherwise, if $\gamma = \varepsilon$ then add the transition $((p, \varepsilon, \alpha), (r, \beta))$. Otherwise, suppose that γ is $\gamma_1\gamma_2\ldots\gamma_n$. If $\alpha = \gamma_1\gamma_2\ldots\gamma_k$ for some $k \leq n$, then add the transition $((p, \varepsilon, \varepsilon), (r, \beta\gamma_{k+1} \ldots \gamma_n))$. In other words, don't bother to push the part that the second transition would have popped off. If $\alpha = \gamma\eta$ for some $\eta \neq \varepsilon$, then add the transition $((p, \varepsilon, \eta), (r, \beta))$. In other words, skip pushing γ and then popping it. Just pop the rest of what the second transition would pop. If any new stack-ε-transitions from p have been created, then replace them as just described except that, if the process creates a transition of the form $((p, \varepsilon, \varepsilon), (p, \gamma'))$, where γ' is not shorter than γ from the first transition that was removed, then the new transition is not describing a path that can ever lead to M' clearing its stack and accepting. So simply delete it. Continue until all stack-ε-transitions have been removed. With a bound on the length of the string that gets pushed when a new transition is created, this process must eventually halt. Since there was no nondeterminism out of q, there won't be nondeterminism out of p when p simply copies the transitions from q.

At this point, M' has the following properties:

- Every transition out of a post\$ state pops at least once character off the stack.
- No transition out of a post\$ state reads any input.
- All accepting states are post\$ states.

Next, we consider problem 2 above (M doesn't accept because its stack isn't empty). That problem would go away if our definition of acceptance were by accepting state alone, rather than by accepting state and empty stack. Recall that, in Example 12.14, we presented an algorithm that constructs, from any PDA that accepts by accepting state and empty stack, an equivalent one that accepts by accepting state alone. The resulting PDA has a new start state s' that pushes a new symbol # onto the stack. It also has a single accepting state, a new state q_a, which is reachable only when the

original machine would have reached an accepting state and had an empty stack. Our next step will be to apply that algorithm to M' to produce M''. Once we've done that, we can later make q_a nonaccepting and thus reject every string in $L\$$. At the same time, we are assured that doing so will not cause M'' to reject any string that was not in $L\$$, since no such string can drive M'' to q_a. The only issue we must confront is that the algorithm of Example 12.14 may convert a deterministic PDA into a nondeterministic one because transitions into q_a may compete with other transitions that were already present (as one does in the example we considered when we presented the algorithm). But that cannot happen in the machine M'' that results when the algorithm is applied to M'. Each new transition into the new state q_a has the form $((a, \varepsilon, \#), (q_a, \varepsilon))$, where a is a post\$ state. No transition in M' pops # since # is not in its stack alphabet. And there are no stack-ε-transitions from a post\$ state in M' (because all such transitions have already been eliminated). So we can guarantee that M'' is equivalent to M' and is still deterministic. We also know that, whenever M'' is in any state except the new start state s' and the new accepting state q_a, there is exactly one # on the stack and it is on the bottom.

Note that we have not switched PDA definitions. We will still accept by accepting state and empty stack. So it will be necessary later to make sure that the final machine that we build can empty its stack on any input it needs to accept.

Next we consider problem 1 above (M halts without reading all its input). We must complete M'', by adding a dead state, in order to guarantee that, from any configuration in which there may be unread input characters (i.e., any configuration with a pre\$ state), M'' has a move that it can make. The problem is that it is not sufficient simply to assure that there is a move for every input character. Consider for example a PDA $M\#$, where $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, $\Gamma = \{\#, \mathtt{1}, \mathtt{2}\}$, and the transitions from state q are $((q, \mathtt{a}, \mathtt{1}), (p, \mathtt{2}))$ and $((q, \mathtt{b}, \mathtt{1}), (r, \mathtt{2}))$. If $M\#$ is in state q and the character on the top of the stack is 2, $M\#$ cannot move.

We can't solve this problem just by requiring that there be some element of Δ for each (input character, stack character) pair because we allow arbitrarily long strings to be popped from the stack on a single move. For example, again let $\Sigma = \{\mathtt{a}, \mathtt{b}\}$ and $\Gamma = \{\#, \mathtt{1}, \mathtt{2}\}$. Suppose that the transitions from state q are:

$((q, \mathtt{a}, \mathtt{12}), (p, \mathtt{2}))$,
$((q, \mathtt{a}, \mathtt{21}), (p, \mathtt{2}))$,
$((q, \mathtt{b}, \mathtt{122}), (r, \mathtt{2}))$,
$((q, \mathtt{b}, \mathtt{211}), (r, \mathtt{2}))$.

If the top of the stack is 22 and the next input character is a or b, $M\#$ cannot move.

So our next step is to convert M'' into a new machine M''' with the following property: every transition, except the one from the start state s', pops exactly one symbol. Note that this is possible because, in every state except s' and the one accepting state q_a, # is on the bottom of the stack. And there are no transitions out of q_a. So there always exists at least one symbol that can be popped. To build M''' we use a slight variant of the technique we used in the algorithm *convertpdatorestriced* that we described in Section 12.3. We replace any transition that popped nothing with a set of transitions, one for each element of Γ''. These transitions pop a symbol and then push it back on. And we replace any transition that popped more than one symbol with a sequence of transitions that pops them one at a time. To guarantee that no nondeterminism is introduced when we do this, it is necessary to be careful when creating new states as described in step 6. If, from some state q, there is more than one transition that pops the same initial sequence of characters, all of them must stay on the same path until they actually pop something different or read a different input character.

Next we add two new dead states, d and d'. The new dead state d will contain strings that do not end in \$. The new dead state d' will contain strings that do end in \$. For every character $c \in \Sigma$, add the transition $((d, c, \varepsilon), (d, \varepsilon))$. So, if M''' ever goes to d, it can loop in d and finish reading its input up until it hits \$. Then add the transition $((d, \$, \varepsilon), (d', \varepsilon))$. So M''' moves from d to d' when it encounters \$. Finally, we must make sure that, from d', M''' can clear its stack. So, for every symbol γ in Γ, add the transition $((d', \varepsilon, \gamma), (d', \varepsilon))$. After adding those transitions, every symbol except # can be removed. Note that none of these new transitions compete with each other, so M''' is still deterministic.

Now we can modify M''' so that it always has a move to make from any pre$ state. To do this, we add transitions into the new dead states. M''' always has a move from s', so we don't have to consider it further. In order to guarantee that M''' will always be able to make a move from any other pre$ state q, it must be the case that, for every (q, c, γ), where q is a pre$ state, $c \in \Sigma \cup \{\$\}$, and $\gamma \in \Gamma'''$, there exists some (p, α) such that either:

- $\Delta_{M'''}$ contains the ε-transition $((q, \varepsilon, \gamma), (p, \alpha))$, or
- $\Delta_{M'''}$ contains the transition $((q, c, \gamma), (p, \alpha))$.

Since M''' is deterministic, it is not possible for $\Delta_{M'''}$ to contain both those transitions. Now consider any stack symbol γ and state q. If M''' contains an ε-transition $((q, \varepsilon, \gamma), (p, \alpha))$, no others from q that pop γ are required. If it does not, then there must be one for each character c in $\Sigma \cup \{\$\}$. If there is no transition $((q, \$, \gamma), (p, \alpha))$, then we add to M''' the transition $((q, \$, \gamma), (d', \varepsilon))$. If, for any other character c, there is no transition $((q, c, \gamma), (p, \alpha))$, then we add to M''' the transition $((q, c, \gamma), (d, \varepsilon))$.

At this point, we know that, until it has read all its input, M''' will always have a move to make. And we know that any string that drives it to d' is in $(\neg L)\$$. So, in the complement machine we are eventually trying to build, d' should accept any strings it sees. To do that, it must first clear the stack.

Next we make sure that, from every post$ state except q_a, M''' always has a move it can make. There is no input to be read, so we must assure that, for every post$ state q (except q_a) and every stack symbol $\gamma \in \Gamma$, there is a move. When M would have died, M''' needs to move to a state that knows that $ has been read and that can clear the stack (so that its complement will eventually be able to accept). That state is d'. So, if (q, ε, γ) is a triple for which no move is defined, add the transition $((q, \varepsilon, \gamma), (d', \varepsilon))$.

Next, we must make sure that M''' never gets into a loop that is not making progress toward at least one of the two things that must occur before it can accept: emptying the stack and consuming the input. M''' determines its next move by considering only its current state, the top stack symbol and the current input character. Any transition that reads an input character makes progress, so we need only worry about those that do not. Suppose that some triple (q, ε, γ) matches against M''''s current configuration. If that triple ever matches again and no progress has been made, then none will ever be made because M''', since it is deterministic, will simply do the same thing the second time. So we must find all the triples with the property that, when they match M''''s configuration, no progress occurs. Call these triples ***dead triples***. We now build a new machine M'''' which is identical to M''' except that all dead triples that originate in a pre$ state will drive M'''' to d and all dead triples that originate in a post$ state will drive M'''' to d'. So $M'''' = M'''$ except:

- If (q, ε, γ) is a dead triple and q is a pre$ state then delete any transition $((q, \varepsilon, \gamma), (p, \beta))$ and replace it by $((q, \varepsilon, \gamma), (d, \varepsilon))$.
- If (q, ε, γ) is a dead triple and q is a post$ state then delete any transition $((q, \varepsilon, \gamma), (p, \beta))$ and replace it by $((q, \varepsilon, \gamma), (d', \varepsilon))$.

Now M'''' has the following properties:

1. On input $w$$, if M's computation would have ended before all of $w$$ were read, M'''' will be able to reach state d' and have the stack empty except for #.
2. On input $w$$, if M's computation would have looped forever, following ε-transitions, without ever halting in either an accepting or a nonaccepting state, M'''' will be able to reach state d' and have the stack empty except for #.
3. On input $w$$, iff M's computation would have accepted, M'''' will be in state q_a and its stack will be empty.
4. On any input that does not end in $, M'''' will be in some pre$ state.

Our final step will be to construct $M\#$ that accepts $(\neg L)\$$. We'll do that by starting with M'''', making q_a nonaccepting, and creating a path by which d' can pop the remaining # and go to an accepting state. But, before we can do that, we must consider two remaining cases:

5. On input w\$, M would have finished reading w\$ but not emptied its stack.
6. On input w\$, M would have finished reading w\$ and landed in a nonaccepting state.

We need to make sure that, in both of those cases, our final machine will be able to accept. Note that we only want to accept after reading \$, so we need only worry about what M'''' should do once it has reached some post\$ state. We first guarantee that M'''' can clear its stack except for #. We do that as follows: For every post\$ state q in M'''' (except q_a) and every symbol c in Γ, if M'''' does not contain a transition for the triple (q, ε, c), add the transition $((q, \varepsilon, c), (d', \varepsilon))$. (If M'''' already contains a transition for the triple (q, ε, c) then that transition must be on a path to clearing the stack or it would already have been eliminated.)

It's now the case that every string of the form w\$, where $w \in \Sigma^*$, will drive M'''' to some post\$ state and either the state is q_a, in which case the stack will be empty, or the state is something else, in which case the stack contains exactly #. So our next step is to add a new state d''. From every post\$ state q except q_a and any states from which there is a transition into q_a, add the transition $((q, \varepsilon, \#), (d'', \varepsilon))$. Since there were no transitions on # from any of those states, the resulting machine is still deterministic.

At this point, M'''' is in deterministic normal form. We can now define:

convertPDAtodetnormalform(M: deterministic PDA) =
1. Return M'''', constructed as described above.

Note that M'''' still accepts L\$ and it is deterministic. It is also in restricted normal form (as defined in Section 12.3.2).

All that remains is to build $M\#$ to accept $(\neg L)$\$. Let $M\# = M''''$ except that d'' will be the only accepting state. There are no transitions out of d'', so there is never competition between accepting and taking some transition. All and only strings of the form w\$, where $w \in \Sigma^*$ and w\$ was not accepted by M will drive $M\#$ to d'' with an empty stack. So $M\#$ accepts $(\neg L)$\$ and it is deterministic.

■

35.3 Proof of Parikh's Theorem

The background for Parikh's Theorem and the definitions of ψ and Ψ are given in Section 13.7.

Theorem 35.3 Parikh's Theorem

Theorem: Every context-free language is letter-equivalent to some regular language.

Proof: We will break the proof into two parts. We will first show that, for every context-free language L, $\Psi(L)$ is semilinear. Then we will show that if $\Psi(L)$ is semilinear then L is letter-equivalent to some regular language.

For purposes of the following discussion, define:

- The sum of two vectors v_1 and v_2, written $v_1 + v_2$, to be the pairwise sum of their elements. So (1, 2) + (5, 7) = (6, 9).
- The product of an integer n and a vector $v = (i_1, i_2, \ldots i_k)$, written nv, to be $(ni_1, ni_2, \ldots ni_k)$. So 4(1, 2) = (4, 8).

A set V of integer vectors of the form $(i_1, i_2, \ldots i_k)$ is ***linear*** iff there exists a finite basis set B and a second finite set C of vectors $c_1, c_2, \ldots$, such that:

$$V = \{v: (v = b + n_1c_1 + n_2c_2 + \ldots + n_{|C|}c_{|C|})\}, \text{ where } n_1, n_2, \ldots, n_{|C|} \text{ are integers, } b \in B, \text{ and } c_1, c_2, \ldots, c_{|C|} \in C.$$

For example:

- $\{(2i, i) : 0 \le i\} = \{(0, 0), (2, 1), (4, 2), (6, 3), \ldots\}$ is linear: $B = \{(0, 0)\}$ and $C = \{(2, 1)\}$.

- $\{(i, j) : 0 \le i \le j\} = \{(0, 0), (0, 1), (0, 2), (1, 3), \ldots, (3, 8), \ldots\}$ is linear: $B = \{(0, 0)\}$ and $C = \{(0, 1), (1, 1)\}$.

A set V of integer vectors of the form $(i_1, i_2, \ldots i_k)$ is ***semilinear*** iff it is the finite union of linear sets. For example, $V = \{(i, j) : i < j \text{ or } j < i\}$ is semilinear because $V = V_1 \cup V_2$, where:

- $V_1 = \{(0, 1), (0, 2), \ldots, (1, 2), (1, 3), \ldots, (3, 8), \ldots\}$ is linear: $B = \{(0, 1)\}$ and $C = \{(0, 1), (1, 1)\}$, and
- $V_2 = \{(1, 0), (2, 0), \ldots, (2, 1), (3, 1), \ldots, (8, 3), \ldots\}$ is linear: $B = \{(1, 0)\}$ and $C = \{(1, 0), (1, 1)\}$.

The core of the proof of Parikh's Theorem is a proof of the claim that if a language L is context-free, then $\Psi(L)$ is semilinear. In fact, sometimes that claim, which we prove next, is called Parikh's Theorem.

Let L be an arbitrary context-free language. Then L is defined by some context-free grammar $G = (V, \Sigma, R, S)$. Let $n = |V - \Sigma|$ (i.e., the number of nonterminals in G) and let b be the branching factor of G (i.e., the length of the longest right hand side of any rule in R). Every string in L has at least one parse tree that can be generated by G. For each such string, choose one of its "smallest" parse trees. In other words, choose one such that there is no other one with fewer nodes. Let T be the set of all such chosen trees. So T contains one smallest tree for each element of L. For any parse tree t, let $yield(t)$ be the string that is the yield of t.

Let t be an arbitrary element of T. Then either:

- The tree t contains no paths that contain repeated nonterminals. By the same argument we used in the proof of the Pumping Theorem, the maximum length of the yield of t is b^n. Call the subset of T that contains all such trees *Short*. *Short* contains a finite number of elements (because there is a bound on the length of the yields and each yield may correspond to only one tree).

- The tree t contains at least one path that contains at least one repeated nonterminal, as shown in Figure 35.1. As we did in the proof of the Pumping Theorem, we can choose one such path and find the first repeated nonterminal, coming up from the bottom, along that path. Call the subtree rooted at the upper instance [1] and the subtree rooted at the lower instance [2]. We can excise the subtree rooted at [1] and replace it by the subtree rooted at [2]. Call the resulting tree t'. There exist values for u, v, x, y, and z such that $yield(t) = uvxyz$, $yield(t') = uxz$, $vy \neq \varepsilon$, $|vxy| \le b^{n+1}$, $|uxz| < |uvxyz|$, and uxz is also in L. If t' still contains any paths that contain any repeated nonterminals, then another vy can be pumped out to yield yet another shorter string in L, and so forth until a string whose parse tree is in *Short* is produced.

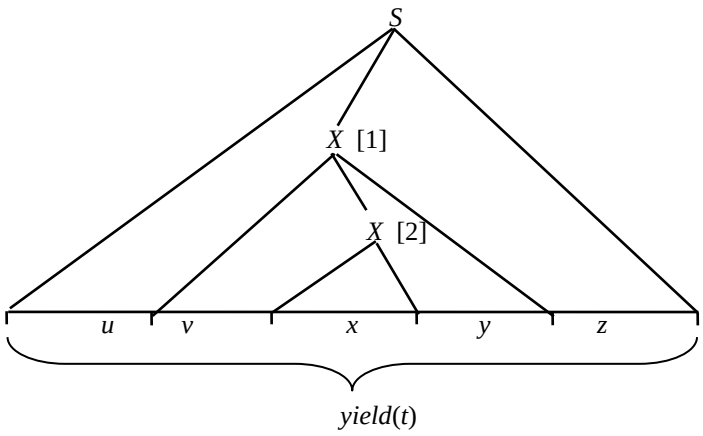


Figure 35.1 A parse tree whose height is greater than *n*

Let *t* be an arbitrary element of *Short*. We will define *produce*(*t*) to be the smallest set of strings that includes *yield*(*t*) plus all the longer strings in *L* that pump down to *yield*(*t*). Or think of it as the smallest set that includes *yield*(*t*) and all the longer strings that can be generated by pumping into *t*.

Since there is a bound on |*vy*|, the number of distinct values for *vy* is finite. For any tree *t* in *Short*, define *pumps*(*t*) to be the set of *vy* strings that can be pumped out of any element of *produce*(*t*) by a single pumping operation. The value of *pumps*(*t*) depends only *t* and the rules of *G*.

We now return to describing strings just by the number of each character that they contain. Let *w* be an element of *produce*(*t*). Then *w* contains all the characters in *yield*(*t*). It also contains all the characters in each *vy* pair that was pumped out of *w* in the process of shrinking *w* down to *yield*(*t*). Let *VY* be a list of all those *vy* pairs (so repeats are included) and let *q* be the length of *VY* (i.e., the number of times some string *vy* was pumped out of *w* to produce *yield*(*t*)). Note that each element of *VY* must be an element of *pumps*(*t*). Then:

$$\psi(w) = \psi(yield(t)) + \sum_{i=1}^{q} \psi(VY_i).$$

We can now prove that $\Psi(produce(t))$ is linear, with:

$B = \{\psi(yield(t))\}$ and $C = pumps(t)$.

For this to be true, it must be the case that:

- Let *w* be an arbitrary element of *produce*(*t*). Then $\psi(w)$ is a linear combination of $\psi(yield(t))$, the single vector in *B*, and some finite number of vectors $c_1, c_2, \ldots$, all of which are drawn from *C*. We just saw that that is true.
- Let *v* be an arbitrary vector that is a linear combination of $\psi(yield(t))$ and some finite number of vectors $c_1, c_2, \ldots$, all of which are drawn from *C*. Then there must exist some string *w* in *produce*(*t*) such that $\psi(w) = v$. This follows from the fact that the Pumping Theorem tells us that any *vy* string that can be pumped out can also be pumped in any number of times.

Now we can prove that $\Psi(L)$ is semilinear. There are a finite number of elements in *Short*. Every string in *L* is an element of *produce*(*t*) for some *t* in *Short*. So $\Psi(L)$ is the finite union of linear sets:

$$\Psi(L) = \bigcup_{t \in Short} \Psi(produce(t)).$$

The last step in the proof of Parikh's Theorem is to show that, given any semilinear set *V*, there exists a regular language *L* such that $\Psi(L) = V$. Let ψ^{-1} be a function that maps from an integer vector *v* to the lexicographically first string *w* such that $\psi(w) = v$. For example, if $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, then $\psi^{-1}((2, 1, 3)) = \texttt{aabccc}$.

We begin by showing that, given any *linear* set V_1, there exists a regular language L_1 such that $\Psi(L_1) = V_1$. Since V_1 is linear, it can be described by the sets *B* and *C*. From them we can produce a regular expression that describes L_1. Let $B = \{b_1, b_2, \ldots\}$ and let $C = \{c_1, c_2, \ldots\}$. Then define $R(V_1)$ to be the following regular expression:

$$(\psi^{-1}(b_1) \cup \psi^{-1}(b_2) \cup \ldots)(\psi^{-1}(c_1) \cup \psi^{-1}(c_2) \cup \ldots)^*.$$

If *L* is the language defined by $R(V_1)$, then $\Psi(L) = V_1$.

For example, if $\Sigma = \{\texttt{a}, \texttt{b}, \texttt{c}\}$, and *V* is defined by $B = \{(1, 2, 3)\}$ and $C = \{(1, 0, 0), (0, 0, 1)\}$, then $R(V) =$

$$(\texttt{abbccc})(\texttt{a} \cup \texttt{c})^*.$$

Now we return to the problem of showing that, given any *semilinear* set *V*, there exists a regular language *L* such that $\Psi(L) = V$. If *V* is semilinear then it is the finite union of linear sets $V_1, V_2, \ldots$. Then *L* is the language described by the regular expression:

$R(V_1) \cup R(V_2) \ldots$

So we have:

- If L is context-free then $\Psi(L)$ is semilinear.
- If $\Psi(L)$ is semilinear then there is some regular language L' such that $\Psi(L') = \Psi(L)$.

Thus, if L is context-free, L is letter-equivalent to some regular language.

■

36 Part IV: Turing Machines and Undecidability

In this chapter, we will prove some of the claims that were made but not proved in Part IV.

36.1 Proof that Nondeterminism Does Not Add Power to Turing Machines

In this section we complete the proof of Theorem 17.2:

Theorem 17.2 Nondeterminism in Deciding and Semideciding Turing Machines

Theorem: If a nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ decides a language L, then there exists a deterministic Turing machine M' that decides L. If a nondeterministic Turing machine M semidecides a language L, then there exists a deterministic Turing machine M' that semidecides L.

Proof Discussion: The proof is by construction of M'. When we sketched this proof in Section 17.3.2, we suggested using breadth-first search as the basis for the construction. The main obstacle that we face in doing that is bookkeeping. If we use breadth-first search, then M' will need to keep track of the partial paths that it is exploring. One approach would be for it to start down path 1, stop after 1 move, remember the path, go one move down path 2, remember it, and so forth, until all paths have been explored for one step. It could then return to path 1 (which has been stored somewhere on the tape), explore each of its branches for one more move, store them somewhere, find path 2 on the tape, continue it for one more move, and so forth. But this approach has two drawbacks:

- The amount of memory (tape space) required to keep track of all the partial paths grows exponentially with the depth of the search, and
- Unlike conventional computers with random access memory, the work required for a Turing machine to scan the tape to find each path in turn and then shift everything to allow for insertion of new nodes into a path could dominate all the work that it would do in actually exploring paths.

Iterative deepening, a hybrid between depth-first search and breadth-first search, avoids both the infinite path pitfall of depth-first search and the exponentially growing memory requirement of breadth-first search. The idea of iterative deepening is simple. We can state the algorithm as follows:

ID(T: search tree) =
1. $d = 1$. /* set the initial depth limit to 1.
2. Loop until a solution is found:
   2.1. Starting at the root node of *T*, use depth-first to explore all paths in *T* of depth *d*.
   2.2. If a solution is found, exit and return it.
   2.3. Otherwise, $d = d + 1$.

Iterative deepening avoids the infinite path pitfall of depth-first search by exploring each path to depth *d* before trying any path to depth *d*+1. So, if there is a finite-length path to a solution, it will be found. And iterative deepening avoids the memory pitfall of breadth-first search by throwing away each partial path when it backs up. Of course, we do pay a price for that: each time we start down a path of length *d*+1 we recreate that path up to length *d*. That seems like a heavy price, but let's look at it more closely.

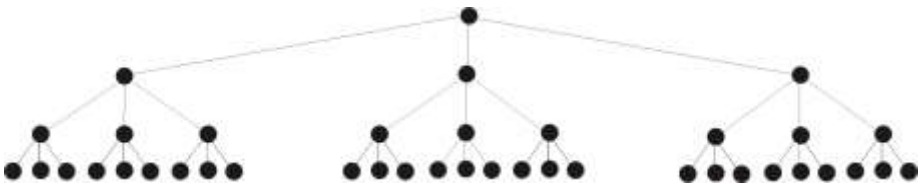


Figure 36.1 A simple search tree

Consider a tree such as the one shown in Fgiure 36.1. Each node in the tree represents a configuration of *M* and each edge represents a step in a computational path that *M* might follow.Observe first that, in iterative deepening, the nodes at the top of the tree get generated a lot of times. The nodes at the very bottom get generated only once, the ones at the level above that only twice, and so forth. Fortunately, there aren't very many nodes at the top of tree. In fact, the number of nodes at any level *d* is larger than the *total* number of nodes at all previous levels by approximately a factor of (*b*-1), where *b* is the branching factor of the tree.

So starting over every time is not as bad as it at first seems. In fact, the relatively inefficient implementation of iterative deepening that we will use examines only a factor of approximately *h* (the height of the tree that is eventually explored) more nodes than does a simple breadth-first search to the correct depth. See B 647 for a proof of this claim.

Proof: We can now return to the task of proving that, for any nondeterministic Turing machine $M = (K, \Sigma, \Gamma, \Delta, s, H)$ there exists an equivalent deterministic Turing machine *M'*. The proof is by construction of a deterministic *M′* that simulates the execution of *M*. *M′* will operate as follows: start with the initial configuration of *M*. Use iterative deepening to try longer and longer computational paths. If any path eventually accepts, *M′* will discover that and accept. If all paths reject, *M′* will discover that and reject. So, if *M* is a deciding Turing machine, *M′* will always halt. If *M* is only a semidecider, however, then *M′* may loop forever.

All that remains is to describe how to perform iterative deepening on a Turing machine. Iterative deepening is usually implemented as a form of bounded depth-first search. For each depth-limited search, a stack is used to keep track of the path so far and the search process backs up whenever it reaches its depth limit. To simplify our implementation, we will choose an approach that does not require a stack. Instead we will create each path, starting from the root, each time.

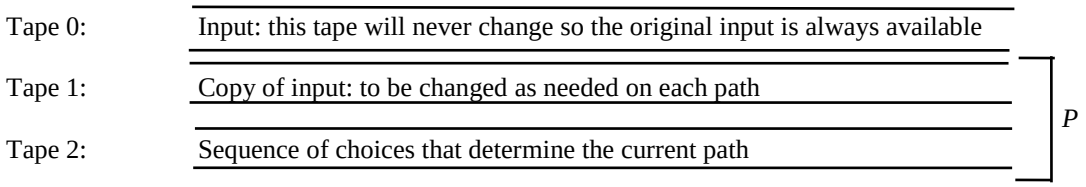


Figure 36.2 Iterative deepening on a three-tape Turing machine

M′ will use three tapes, as shown in Figure 36.2. Tapes 1 and 2 correspond to the current path. To see how *M′* works, we will first define a subroutine *P* that uses tapes 1 and 2 and follows one specific path for some specified number of steps. Then we will see how *M′* can invoke *P* on a sequence of longer and longer paths.

Suppose we want to specify some one specific path through the search tree that *M* explores. To do this, we first need to be able to write down the set of alternatives for each move in some order so that it makes sense to say, "Choose option 1 this time. Choose option 2 the second time, and so forth". Imagine that we have that. (We will describe such a method shortly.) Then we could specify any finite path as a ***move vector***: a finite sequence of integers such as "2, 5, 1, 3", which we will interpret to mean: "Follow a path of length 4. For the first move, choose option 2. For the next, choose option 5. For the third, choose option 1. For the fourth, choose option 3. Halt." The Turing machine *P* that we mentioned above, the one that follows one particular finite path and reports its answer, is then a machine that follows a move vector such as "2, 5, 1, 3".

So now we need a way for *P* to interpret a move vector. To solve this problem we first observe that there is a maximum number *B* of branches at any point in *M*'s execution. For its next move, *M* chooses from among $|K|$ states to go to, from among $|\Gamma|$ characters to write on the tape, and between moving left and moving right. Thus

$$B = 2 \cdot |K| \cdot |\Gamma|.$$

Since there are only at most B choices at each point, the largest number that can occur in any move vector for M is B. Of course, it will often happen that Δ offers M many fewer choices given its current state and the character under its read/write head. Suppose that we imagine organizing Δ so that, for each (q, c) pair, we get a list (in some arbitrary order) of the moves that M may make if it is in state q and c is under the read/write head. We can enter that information into an indexable table T as shown in Table 36.1(a). We assume that we can sequentially number both the states and the elements of Γ. Each move choice is an element of $(K \times \Gamma \times \{\rightarrow, \leftarrow\})$. But what happens if P is told to choose move j and fewer than j choices are available? To solve this problem, we will fill out T by repeating the sequence of allowable moves as many times across each row as necessary to fill up the row. So T will actually be as shown in Table 36.1(b).

	1	2	3	…	*B*
(state 1, char 1)	move choice 1	move choice 2			
(state 1, char 2)	move choice 1	move choice 2	move choice 3	move choice 4	
…	move choice 1				
(state 2, char 1)	move choice 1	move choice 2	move choice 3		
….	move choice 1	move choice 2			
(state $\|K\|$, char $\|\Gamma\|$)	move choice 1				

(a)

	1	2	3	…	*B*
(state 1, char 1)	move choice 1	move choice 2	move choice 1	move choice 2	move choice 1
(state 1, char 2)	move choice 1	move choice 2	move choice 3	move choice 4	move choice 1
…	move choice 1	move choice 1	move choice 1	move choice 1	move choice 1
(state 2, char 1)	move choice 1	move choice 2	move choice 3	move choice 1	move choice 2
….	move choice 1	move choice 2	move choice 1	move choice 2	move choice 1
(state $\|K\|$, char $\|\Gamma\|$)	move choice 1	move choice 1	move choice 1	move choice 1	move choice 1

(b)

Table 36.1 A table that lists all of *M*'s move choices

There are two important things about each row in this table:

- Every entry is a move that is allowed by Δ, and
- Every move that is allowed by Δ appears at least once.

Also notice that, given a particular nondeterministic Turing machine M, we can build this table and it will contain a finite number of cells. In addition, there is a finite number $|K|$ of different states that M can be in at any point in following some path. So, given M, we can build a new Turing machine P to follow one of M's paths and we can encode all of the move table, as well as the current simulated state of M, in P's finite state controller.

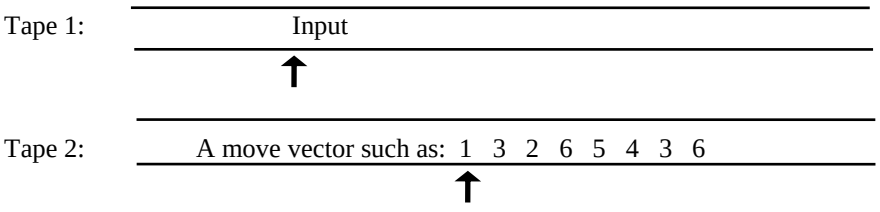


Figure 36.3 Using two tapes to simulate one path of fixed length

We are now ready to define *P* (the Turing machine that follows one finite path that *M* could take). *P* uses two tapes, as shown in Figure 36.3. The table *T* and the current (simulated) state of *M* are encoded in the state of *P*, which operates as follows:

1. For *i* = 1 to length of the move vector on Tape 2 do:
   1.1. Determine *c*, the character under the read/write head of Tape 1.
   1.2. Consider *q*, the current simulated state of *M*. If *q* is a halting state, halt. Otherwise:
   1.3. Determine *v*, the value of square *i* of the move vector.
   1.4. Look in *T* to determine the value in the row labeled (*q*, *c*) and the column labeled *v*. Call it *m*.
   1.5. Make move *m* (by writing on tape 1, moving tape 1's read/write head, and changing the simulated state as specified by *m*).

Whatever happens, *P* halts after at most *n* steps, where *n* = |Tape 2|.

Now that we have specified *P*, we are ready to specify *M′*, the deterministic Turing machine that is equivalent to *M*. *M′* uses three tapes: Tape 0 holds the original input to *M*. It will not change throughout the computation. Tapes 1 and 2 will be used by instantiations of *P*. *M′*'s job is to invoke *P* with all paths of length 0, then all paths of length 1, all paths of length 2, and so forth. For example, suppose that *B* = 4. Then the value on Tape 2 at the first several calls by *M′* to *P* will be: ε; 1; 2; 3; 4; 1,1; 1,2; 1,3; 1,4; 2,1; …; 2,4; 3,1; …; 3,4; 4,1; …; 4,4; 1,1,1; 1,1,2; …

To see how *M′* can use *P*, let's first consider the simplest case, namely the one on which *M′* is a semideciding machine that will accept if any path of *M* accepts; otherwise it will simply loop looking for some accepting path. In this case, *M′* operates as follows on input *w*:

1. Write ε (corresponding to a path of length 0) on Tape 2.
2. Until *P* accepts do:
   2.1. Copy *w* from Tape 0 to Tape 1.
   2.2. Invoke *P* (i.e., simulate *M* for |Tape 2| steps following the path specified on Tape 2).
   2.3. If *P* discovers that *M* would have accepted then halt and accept.
   2.4. Otherwise, generate the lexicographically next string on Tape 2.

Next we consider what must happen if *M* is to be able to reject as well as to accept. It can only do reject if every path eventually halts and rejects. So now we need to design *M′* so that it will halt as soon as one of the following things happens:

- It discovers a path along which *M* halts and accepts. In this case, *M′* accepts.
- It has tried all paths until they halt, but all have rejected. In this case, *M′* rejects.

The first of these conditions can be checked as described above. The second is a bit more difficult. Suppose that *M′* discovers that *M* would halt and reject on the path 2, 1, 4. *M* must continue to try to find some accepting path. But it restarts every path at the beginning. How is it to know not to try 2, 1, 4, 1, or any other path starting with 2, 1, 4? It's hard to make it do that, but we can make it notice if it tries every path of length *n*, for some *n*, and all of them have halted.

If every path of *M* halts, then there is some number *n* that is the maximum number of moves made by any path before it halts. *M′* should be able to notice that every path of length *n* halts. At that point, it need not consider any longer paths. So we'll modify *M′* so that, in its finite state controller, it remembers the value of a Boolean variable we can call *nothalted*, which we'll initialize to *False*. Whenever *M′* tries a path that hasn't yet halted, it will set *nothalted* to *True*. Now consider the procedure that generates the lexicographically next string on tape 2. Whenever it is about to generate a string that is one symbol longer than its predecessor (i.e., it is about to start looking at longer paths), it will check the value of *nothalted*. If it is *False*, then all paths of the length it was just considering halted. *M′* can quit. If, on the other hand, *nothalted* is *True*, then there was at least one path that hasn't yet halted. *M′* needs to try longer paths. The variable *nothalted* must be reset to *False*, and the next longer set of paths considered. So *M′* operates as follows on input *w*:

1. Write ε (corresponding to a path of length 0) on Tape 2.
2. Set *nothalted* to *False*.
3. Until *P* accepts or rejects do:
   3.1. Copy *w* from Tape 0 to Tape 1.
   3.2. Invoke *P* (i.e., simulate *M* for |Tape 2| steps following the path specified on Tape 2).
   3.3. If *P* discovers that *M* would have accepted then accept.
   3.4. If *P* discovers that *M* would not have halted, then set *nothalted* to *True*.
   3.5. If the lexicographically next string on Tape 2 would be longer than the current one then:
      Check the value of *nothalted*. If it is *False*, then reject. All paths of the current length halted but none of them accepted.
      Otherwise, set *nothalted* to *False*. We'll try again with paths of the next longer length and see if all of them halt.
   3.6. Generate the lexicographically next string on Tape 2.

If *M* is a semideciding Turing machine, then *M′* will accept iff *M* would. If *M* is a deciding Turing machine, all of its paths must eventually halt. If one of them accepts, *M′* will find it and accept. If all of them reject, *M′* will notice when all paths of a given length have halted without accepting. At that point, it will reject. So *M′* is a deciding machine for *L*(*M*).

■

36.2 An Analysis of Iterative Deepening

Consider a complete tree *T* with branching factor *b* and height *h*. Assume that each node of *T*, including the root, corresponds to a state and each edge corresponds to a move from one state to another. We want to compare the number of moves that will be considered for each of three search strategies.

We first consider a straightforward ***breadth-first search***. There are b^d edges between nodes at level $d-1$ and nodes at level d. So the number of moves that will be considered by breadth-first search, to depth *h*, will be:

$$\sum_{d=1}^{h} b^d = \frac{b(b^h - 1)}{b-1} = \mathrm{O}(b^h).$$

Now suppose that we use standard ***iterative deepening***, defined as follows:

ID(*T*: search tree) =
1. *d* = 1. /* Set the initial depth limit to 1.
2. Loop until a solution is found:
   2.1. Starting at the root node of *T*, use depth-first to explore all paths in *T* of depth *d*.
   2.2. If a solution is found, exit and return it.
   2.3. Otherwise, *d* = *d* + 1.

Assume that *ID* halts with a solution at depth *h*. Then the number of moves that it considered is:

at least: $\sum_{d=1}^{h-1}\left(\sum_{k=1}^{d} b^k\right) + h$, and at most: $\sum_{d=1}^{h}\left(\sum_{k=1}^{d} b^k\right) = \frac{b^{h+2} - (h+1)b^2 + hb}{(b-1)^2} = \mathrm{O}(b^h)$.

The lower bound comes from the fact that *ID* must have explored at least one path at depth *h* or it would have halted at depth *h*-1. The upper bound corresponds to it finding a solution on the very last path in the tree. To see where that upper bound formula comes from, notice that *ID* makes one pass through its loop for each value of *d*, so we must sum over all of them. On the d^{th} pass, it does a simple depth-first search of a tree of depth *d* and branching factor *b*.

Now consider a variant of iterative deepening in which, instead of doing a backtracking search at each depth limit *d*, we start each path over again at the root. So each path of length 1 is considered. Then each path of length 2 is considered, starting each from the root. Then each path of length 3 is considered, starting from the root, and so forth. This is the technique we used in Section 36.1 to prove Theorem 17.2. Because reaching each of the b^d nodes at level d requires d moves, the number of moves that this algorithm considers is:

at least: $\sum_{d=1}^{b-1} d\,b^d + b$, and at most: $\sum_{d=1}^{b} d\,b^d = \frac{bb^{b+2} - (b+1)b^{b+1} + b}{(b-1)^2} = \mathrm{O}(bb^b)$.

36.3 The Power of Reduction

Define a ***planar grid*** to be a set of lines with two properties:

- No two lines are co-linear, and
- Each line is either parallel or perpendicular to every other one.

We'll call each position at which two lines intersect a ***grid point*** or just a ***point***. Now consider the following problem from [Dijkstra EWD-1248]:

> Show that, for any finite set of grid points in the plane, we can colour each of the points either red or blue such that on each grid line the number of red points differs by at most 1 from the number of blue points.

Figure 36.4 A grid problem and its corresponding graph version

An instance of this problem could be the grid shown in Figure 36.4(a). The selected grid points are shown as circles. One way to attack the problem is directly: we could prove the claim using operations on the grid. An alternative is to reduce the problem to one that is stated in some other terms that give us useful tools for finding a solution.

[Misra 1996] suggests reducing this grid problem to a graph problem. The reduction described there works as follows: Given a grid and a finite set of points on the grid, construct a graph in which each grid line becomes a vertex and there is an edge between two vertices iff the corresponding grid lines share one of the given points. The graph that is produced from our example grid is shown in Figure 36.4(b).

Notice that the number of edges in the constructed graph is finite (since the number of points in the grid problem is finite). The problem to be solved is now to show that there exists a way to color the *edges* of the graph in such a way that the polarity of each vertex is at most one. Define the ***polarity*** of a vertex to be the absolute value of the difference between the number of red and blue edges incident on it. We'll show that the required coloring exists by describing an algorithm to construct it.

Observe that, in any graph that this reduction builds, each vertex corresponds either to a vertical or to a horizontal grid line. Since each edge connects a "vertical" vertex to a "horizontal" vertex, the graph must be bipartite. (In other words, it is possible to divide the vertices into two sets, in this case the "horizontal" ones and the "vertical" ones, in such a way that no edge is incident on two vertices in the same set.)

Now we can exploit anything we know about bipartite graphs. In particular, we'll use the fact that, in a bipartite graph, every cycle has an even number of edges. So, in any cycle, we can color the edges alternately, red and blue, without affecting the polarity of any vertex. Hence, we may remove all cycles from the graph (in arbitrary order) and solve the coloring problem over the remaining edges. After removing the cycles, we are left with an acyclic undirected graph, i.e., a tree or a forest of trees.

If the forest is not connected, then each maximal connected tree within it can be colored independently of the others since no pair of such trees shares any vertices. To color each tree, begin by designating some vertex to be the root. Color the edges incident on the root alternately. Then pick any vertex that has both colored and uncolored incident edges. If there is no such vertex then all edges have been colored. Otherwise, the vertex has exactly one colored edge, say red, incident on it; color the incident uncolored edges alternately starting with blue, so as to meet the polarity constraint.

36.4 The Undecidability of the Post Correspondence Problem

In Section 22.2, we defined the language:

- PCP = {<*P*> : *P* is an instance of the Post Correspondence problem and *P* has a solution}.

Theorem 22.1 asserts that PCP is in SD/D. We proved that it is in SD by presenting the algorithm, M_{PCP}, that semidecides it. We will now present the proof that it is not in D.

We begin by defining a related language MPCP (modified PCP). An instance of MPCP looks exactly like an instance of PCP. So it is a string <*P*> of the form:

$$<P> = (x_1, x_2, x_3, \ldots, x_n)(y_1, y_2, y_3, \ldots, y_n), \text{ where } \forall j\ (x_j \in \Sigma^+ \text{ and } y_j \in \Sigma^+).$$

The difference between PCP and MPCP is in the definition of a solution. A solution to an MPCP instance is a finite sequence 1, i_2, …i_k of integers such that:

$$\forall j\ (1 \le i_j \le n \text{ and } x_1 x_{i2} \ldots x_{ik} = y_1 y x_{i2} \ldots y_{ik}).$$

In other words, the first index in any solution must be 1.

Recall that Theorem 23.3 tells us that the language L_a = {<*G*, *w*> : *G* is an unrestricted grammar and $w \in L(G)$} is not in D. We will show that PCP is not in D in two steps. We will prove that:

- $L_a \le$ MPCP, so MPCP is not in D because L_a isn't.
- MPCP ≤ PCP, so PCP is not in D because MPCP isn't.

Theorem 36.1 MPCP is not in D

Theorem: MPCP is not in D.

Proof: The proof is by reduction from L_a = {<*G*, *w*> : *G* is an unrestricted grammar and $w \in L(G)$}. Given a string <*G*, *w*>, we show how to construct an instance *P* of MPCP with the property that *P* has a solution iff *G* generates *w* (and thus <*G*, *w*> is in L_a). The idea is that we'll construct the *X* and *Y* lists of *P* so that they can be used to build up strings that describe derivations that *G* can produce. We'll make sure that it is possible to build the same string from the two lists exactly in case *G* can generate *w*.

Let $G = (V, \Sigma, R, S)$ be an unrestricted grammar. Suppose that G can derive w. Then there is a string of the following form that describes the derivation:

$$S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow w$$

Let % and & be two symbols that are not in V. We'll use % to mark the beginning of a derivation and & to mark the end. Using this convention, a derivation will look like:

$$\%S \Rightarrow x_1 \Rightarrow x_2 \Rightarrow \ldots \Rightarrow w\&$$

From G and w, the reduction that we are about to define will construct an MPCP instance with the property that both the X list and the Y list can be used to generate such derivation strings. We'll design the two lists so that when we use the X list we are one derivation step ahead of where we are when we use the Y list. So the only way for the two lists to end up generating the same derivation string will be to choose a final index that lets Y catch up. We'll make sure that that can happen only when the final generated string is w.

	X	**Y**		*Comment*
1	$\%S \Rightarrow$	%		Get started, with X one step ahead.
	&	$\Rightarrow w\&$		End, with Y doing its last step and catching up.
	c	c	For every symbol c in V	Copy characters that don't change.
	β	α	For every rule $\alpha \rightarrow \beta$ in R	Apply each rule. X will generate β when Y is one step behind and so is generating α.
	$\Rightarrow$	$\Rightarrow$		

Table 36.2 Building an MPCP instance from a grammar and a string

Specifically given $G = (V, \Sigma, R, S)$ and w, we will build the X and Y lists as shown in Table 36.2. The entry that is listed on line one must be on line one. Since any solution to an MPCP problem must be a sequence that starts with 1, we thus guarantee that any solution must generate a string that begins with $\%S \Rightarrow$. The other entries may occur in any order. Notice that the entries that correspond to the rules of G are "backwards". This happens because the X-generated string is one derivation ahead of the Y-generated one.

To see how this construction works, we'll consider a simple example. Let $G = (\{S, A, B, \mathtt{a}, \mathtt{b}, \mathtt{c}\}, \{\mathtt{a}, \mathtt{b}, \mathtt{c}\}, R, S)$, where $R =$

$S \rightarrow AB\mathtt{c}$
$S \rightarrow ABS\mathtt{c}$
$AB \rightarrow BA$
$B\mathtt{c} \rightarrow \mathtt{bc}$
$BA \rightarrow \mathtt{a}$
$A \rightarrow \mathtt{a}$

Given G and the string $w = \mathtt{ac}$, the reduction will build the MPCP instance shown in Table 36.3. G can derive ac. So this MPCP instance has a solution, (1, 9, 15, 11, 6, 15, 13, 6, 2), shown in Figure 36.5.

	X	Y			X	Y
1	%S ⇒	%		9	ABc	S
2	&	⇒ ac&		10	ABSc	S
3	S	S		11	BA	AB
4	A	A		12	bc	Bc
5	B	B		13	a	BA
6	c	c		14	a	A
7	b	b		15	⇒	⇒
8	a	a				

Table 36.3 An example of building an MPCP instance from a grammar and a string

	1	9	15	11	6	15	13	6	2
X	% S ⇒	ABc	⇒	BA	c	⇒	a	c	&
Y	%	S	⇒	AB	c	⇒	BA	c	⇒ ac&

Figure 36.5 This MPCP instance has (1, 9, 15, 11, 6, 15, 13, 6, 2) as a solution

To complete the proof, we must show that the MPCP instance, P, that is built from the input, $<G, w>$, has a solution iff G can derive w. The formal argument can be made by induction on the length of a derivation. We omit it. The general idea is as we suggested above. Any MPCP solution starts with the index 1. Given the lists as we have described them, this means that the X-generated string starts out with one more derivation step than does the Y-generated string. The only way for the Y-generated string to "catch up" is to use the second entry in the table. If that is done, then the final generated string can only be w. In between the first index and the last one, all the table entries have been constructed so that all and only derivations that match the rules in G can be generated. So the two lists will correspond iff G can generate w.

So we have that $L_a \leq$ MPCP. Let $R(<G, w>)$ be the reduction that we have just described. If there existed a Turing machine M that decided MPCP, then $M(R(<G, w>))$ would decide L_a. But L_a is not in D, so no decider for it exists. So M does not exist either and MPCP is not in D.

■

Theorem 36.2 PCP is not in D

Theorem: PCP is not in D.

Proof: The proof is by reduction from MPCP. In moving from MPCP to PCP, we lose the constraint that a solution necessarily starts with 1. But we can effectively retain that constraint by modifying the X and Y lists so that the only sequences that will cause the two lists to generate the same string must start with 1. Given an MPCP instance $<X, Y>$, we will create a PCP instance $<A, B>$ with the property that $<X, Y>$ has a solution iff $<A, B>$ does. The new lists $<A, B>$ will differ from the original ones in two ways: each list will contain two new strings and each string will be made twice as long by inserting a special symbol after each original symbol (in the case of the X list) or before each original symbol (in the case of the Y list).

Let $MP = <X, Y>$ be an instance of MPCP with alphabet Σ and size n. Let ¢ and \$ be two characters that are not in Σ. We will build $P = <A, B>$, an instance of PCP with alphabet $\Sigma \cup \{¢, \$\}$ and size $n+2$ as follows:

Assume that:	$X = x_1, x_2, \ldots x_n$	and	$Y = y_1, y_2, \ldots y_n$.
We construct:	$A = a_0, a_1, a_2, \ldots a_n, a_{n+1}$	and	$B = b_0, b_1, b_2, \ldots b_n, b_{n+1}$.

For values of i between 1 and n, construct the elements of the A and B lists as follows:

- Let a_i be x_i except that the symbol ¢ will be inserted *after* each symbol of x_i. For example, if x_i is `aab` then a_i will be `a¢a¢b¢`.
- Let b_i be y_i except that the symbol ¢ will be inserted *before* each symbol of y_i. For example, if y_i is `aab` then b_i will be `¢a¢a¢b`.

Then let: $a_0 = ¢a_1$,
$a_{n+1} = \$$,
$b_0 = b_1$, and
$b_{n+1} = ¢\$$.

For example:

If:	$X =$ `a, baa`	and	$Y =$ `ab, aa`
Then:	$A =$ `¢a¢, a¢, b¢a¢a¢, $`	and	$B =$ `¢a¢b, ¢a¢b, ¢a¢a, ¢$`

Now we must show that $MP = \langle X, Y \rangle$ has an MPCP solution iff $P = \langle A, B \rangle$ has a PCP solution:

If $MP = \langle X, Y \rangle$ has an MPCP solution, then it is of the form $(1, j_2, j_3, \ldots j_k)$, for some k. In that case, the sequence $(0, j_2, j_3, \ldots j_k, n+1)$ is a solution to $P = \langle A, B \rangle$. The string that this new sequence produces will be identical to the string that the original sequence produced except that there will be the symbol ¢ between each pair of other symbols and the string will start with ¢ and end with ¢$. We choose the first element of the sequence to be 0 rather than 1 to create the initial ¢ in the A-generated list, and we add the final element, $n+1$, so that the B-generated sequence can catch up and contain the final ¢.

If $P = \langle A, B \rangle$ has a PCP solution S, then it must be of the form $(0, \langle main\ part \rangle, n+1)$, where $\langle main\ part \rangle$ is all of S minus its first and last elements. We know that S has to start with 0 because every string in the B list starts with the symbol ¢. The only string in the A list that starts with the symbol ¢ is the first one (which we've numbered 0). So the only way that the strings that are generated from the two lists can match is for the first index to be 0. We know that S must end with $n+1$ because every string in the A list except the last ends with ¢. But no string in the B list does. But B's $n+1^{st}$ element provides that final ¢ and it provides nothing else except the final $. The string that S produces is identical, if we remove all instances of ¢ and $, to the string that the sequence $(1, \langle main\ part \rangle)$ would produce given $\langle X, Y \rangle$. This is true because we constructed elements 1 through n of $\langle A, B \rangle$ to be identical to the corresponding elements of $\langle X, Y \rangle$ except for the insertion of ¢ and $. And we guaranteed, again ignoring ¢ and $, that $a_0 = a_1 = x_1$. So the sequence $(1, \langle main\ part \rangle)$ generates the same string from both the X and Y lists and its first element is 1. So it is an MPCP solution for $MP = \langle X, Y \rangle$.

So we have that MPCP ≤ PCP. Let $R(\langle X, Y \rangle)$ be the reduction that we have just described. If there existed a Turing machine M that decided PCP, then $M(R(\langle X, Y \rangle))$ would decide MPCP. But MPCP is not in D, so no decider for it exists. So M does not exist either and PCP is not in D.

■

37 Part V: Complexity

In this chapter, we will prove some of the claims that were made but not proved in Part V.

37.1 Asymptotic Dominance

In this section we prove the claims made in Section 27.5.

Theorem 37.1 Facts about $\mathcal{O}$

We will prove separately each of the claims made in the theorem. The basis for these proofs is the definition of the relation $\mathcal{O}$: $f(n) \in \mathcal{O}(g(n))$ iff there exists a positive integer k and a positive constant c such that:

$$\forall n \geq k \; (f(n) \leq c \; g(n)).$$

Let f, f_1, f_2, g, g_1, and g_2 be functions from the natural numbers to the positive reals, let a and b be arbitrary real constants, and let c, c_0, c_1, … c_k be any positive real constants. Then:

Fact 1: $f(n) \in \mathcal{O}(f(n))$.

Let $k = 0$ and $c = 1$. Then $\forall n \geq k \; (f(n) \leq c \; f(n))$.

Fact 2: Addition:

1. **$\mathcal{O}(f(n)) = \mathcal{O}(f(n) + c_0)$ (if we make the assumption, which will always be true for the functions we will be considering, that $1 \in \mathcal{O}(f(n))$).**

   We first note that, for any function $g(n)$, if $g(n) \in \mathcal{O}(f(n))$ then it must also be true that $g(n) \in \mathcal{O}(f(n) + c_0)$ since $f(n) \leq f(n) + c_0$. Now we show the other direction: Since $1 \in \mathcal{O}(f(n))$, there must exist k_1 and c_1 such that:

   $$\forall n \geq k_1 \;\; (1 \leq c_1 \; f(n)) \\ (c_0 \leq c_0 \; c_1 \; f(n)). \qquad (1)$$

   If $g(n) \in \mathcal{O}(f(n) + c_0)$ then there must exist k_2 and c_2 such that:

   $$\forall n \geq k_2 \; (g(n) \leq c_2(f(n) + c_0)).$$

   Combining that with (1), we get:

   $$\forall n \geq max(k_1, k_2) \; (g(n) \leq c_2(f(n) + c_0 \; c_1 \; f(n)) \\ \leq (c_2 + c_0 \; c_1)(f(n))).$$

   So let $k = max(k_1, k_2)$ and $c = c_2 + c_0 \; c_1$. Then:

   $$\forall n \geq k \;\; (g(n) \leq c \; f(n)).$$

2. **If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) + f_2(n) \in \mathcal{O}(g_1(n) + g_2(n))$.**

   If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$, then there must exist k_1, c_1, k_2 and c_2 such that:

   $$\forall n \geq k_1 \; (f_1(n) \leq c_1 \; g_1(n)).$$
   $$\forall n \geq k_2 \; (f_2(n) \leq c_2 \; g_2(n)).$$

   So: $$\forall n \geq max(k_1, k_2) \; (f_1(n) + f_2(n) \leq c_1 \; g_1(n) + c_2 \; g_2(n) \\ \leq max(c_1, c_2)(g_1(n) + g_2(n))).$$

So let $k = max(k_1, k_2)$ and $c = max(c_1, c_2)$. Then:

$$\forall n \geq k \quad (f_1(n) + f_2(n) \leq c(g_1(n) + g_2(n))).$$

3. $\mathcal{O}(f_1(n) + f_2(n)) = \mathcal{O}(max(f_1(n), f_2(n)))$.

We first show that if $g(n) \in \mathcal{O}(f_1(n) + f_2(n))$ then $g(n) \in \mathcal{O}(max(f_1(n), f_2(n)))$: If $g(n) \in \mathcal{O}(f_1(n) + f_2(n))$, then there must exist k_1 and c_1 such that:

$$\begin{aligned}\forall n \geq k_1 \ (g(n) &\leq c_1(f_1(n) + f_2(n)) \\ &\leq 2c_1 \cdot max(f_1(n), f_2(n))).\end{aligned}$$

So let $k = k_1$ and $c = 2c_1$. Then:

$$\forall n \geq k \quad (g(n) \leq c \ max(f_1(n), f_2(n))).$$

Next we show that if $g(n) \in \mathcal{O}(max(f_1(n), f_2(n)))$ then $g(n) \in \mathcal{O}(f_1(n) + f_2(n))$: If $g(n) \in \mathcal{O}(max(f_1(n), f_2(n)))$, then there must exist k_1 and c_1 such that:

$$\begin{aligned}\forall n \geq k_1 \ (g(n) &\leq c_1 \ max(f_1(n), f_2(n)) \\ &\leq c_1 \ (f_1(n) + f_2(n))).\end{aligned}$$

So let $k = k_1$ and $c = c_1$. Then:

$$\forall n \geq k \quad (g(n) \leq c(f_1(n) + f_2(n))).$$

Fact 3: Multiplication:

1. $\mathcal{O}(f(n)) = \mathcal{O}(c_0 \ f(n))$.

We first show that, if $g(n) \in \mathcal{O}(f(n))$, then $g(n) \in \mathcal{O}(c_0 \ f(n))$. If $g(n) \in \mathcal{O}(f(n))$, then there must exist k_1 and c_1 such that:

$$\forall n \geq k_1 \ (g(n) \leq c_1 \ f(n)).$$

So let $k = k_1$ and $c = c_1/c_0$. (Thus $c_1 = c \ c_0$.) Then:

$$\forall n \geq k \ (g(n) \leq c \ c_0 \ f(n)).$$

Next we show that, if $g(n) \in \mathcal{O}(c_0 \ f(n))$, then $g(n) \in \mathcal{O}(f(n))$. If $g(n) \in \mathcal{O}(c_0 \ f(n))$, then there must exist k_1 and c_1 such that:

$$\forall n \geq k_1 \ (g(n) \leq c_1 \ c_0 \ f(n)).$$

So let $k = k_1$ and $c = c_1 \ c_0$. Then:

$$\forall n \geq k \ (g(n) \leq c \ f(n)).$$

2. If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$ then $f_1(n) \ f_2(n) \in \mathcal{O}(g_1(n) \ g_2(n))$.

If $f_1(n) \in \mathcal{O}(g_1(n))$ and $f_2(n) \in \mathcal{O}(g_2(n))$, then there must exist k_1, c_1, k_2 and c_2 such that:

$$\forall n \geq k_1 \ (f_1(n) \leq c_1 \ g_1(n))$$

$\forall n \geq k_2\ (f_2(n) \leq c_2\, g_2(n))$.

Thus: $\forall n \geq max(k_1, k_2)\ (f_1(n)\, f_2(n) \leq c_1\, c_2\, g_1(n)\, (g_2(n))$.

So let $k = max(k_1, k_2)$ and $c = c_1\, c_2$. Then:

$\forall n \geq k\ \ (f_1(n)\, f_2(n) \leq c\, g_1(n)\, (g_2(n))$.

Fact 4: Polynomials:

1. **If $a \leq b$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^b)$.**

   If $g(n) \in \mathcal{O}(n^a)$, then there must exist k_1 and c_1 such that:

   $$\forall n \geq k_1\ (g(n) \leq c_1\, n^a \leq c_1\, n^b) \qquad (\text{since } (a \leq b) \rightarrow (n^a \leq n^b)).$$

   So let $k = k_1$ and $c = c_1$. Then:

   $\forall n \geq k\ \ (g(n) \leq c\, n^b)$.

2. **If $f(n) = c_j n^j + c_{j-1} n^{j-1} + \ldots\ c_1 n + c_0$, then $f(n) \in \mathcal{O}(n^j)$.**

   $$\forall n \geq 1\ (c_j n^j + c_{j-1} n^{j-1} + \ldots\ c_1 n + c_0 \leq c_j n^j + c_{j-1} n^j + \ldots\ c_1 n^j + c_0\, n^j \leq (c_j + c_{j-1} + \ldots\ c_1 + c_0)\, n^j).$$

   So let $k = 1$ and $c = (c_j + c_{j-1} + \ldots\ c_1 + c_0)$. Then:

   $\forall n \geq k\ \ (f(n) \leq c\, n^j)$.

Fact 5: Logarithms:

1. **For a and $b > 1$, $\mathcal{O}(\log_a n) = \mathcal{O}(\log_b n)$.**

   Without loss of generality, it suffices to show that $\mathcal{O}(\log_a n) \subseteq \mathcal{O}(\log_b n)$. If $g(n) \in \mathcal{O}(\log_a n)$, then there must exist k_1 and c_1 such that:

   $\forall n \geq k_1\ (g(n) \leq c_1 \log_a n)$.

   Note that $\log_a n = \log_a b \log_b n$. So let $k = k_1$ and $c = c_1 \log_a b$. Then:

   $\forall n \geq k\ \ (g(n) \leq c \log_b n)$.

2. **If $0 < a < b$ and $c > 1$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^a \log_c n) \subseteq \mathcal{O}(n^b)$.**

   First we show that $\mathcal{O}(n^a) \subseteq \mathcal{O}(n^a \log_c n)$. For any $n \geq c$, $\log_c n \geq 1$, so $n^a \leq n^a \log_c n$. If $g(n) \in \mathcal{O}(n^a)$, then there must exist k_1 and c_1 such that:

   $\forall n \geq k_1\ (g(n) \leq c_1 n^a)$.

   So let $k = max(k_1, c)$ and $c_0 = c$. Then:

   $\forall n \geq k\ \ (g(n) \leq c_0\, n^a \log_c n)$.

   Next we show that $\mathcal{O}(n^a \log_c n) \subseteq \mathcal{O}(n^b)$. First, notice that, for $p > 0$ and $n \geq 1$, we have:

$$\log_e n = \int_1^n x^{-1}dx \le \int_1^n x^{-1+p}dx = \frac{1}{p}(n^p - 1) < \frac{1}{p}n^p.$$

In particular, for $p = b - a$, we have:

$$\log_e n < \frac{1}{b-a}n^{b-a}.$$

If $g(n) \in \mathcal{O}(n^a \log_c n)$, then there must exist k_1 and c_1 such that:

$$\forall n \ge k_1 \ (g(n) \le c_1 n^a \log_c n).$$

So, for all $n \ge max(1, k_1)$, we have that:

$$g(n) \le cn^a \log_c n = cn^a \log_c e \log_e n < \frac{c \log_c e}{b-a} n^a n^{b-a} \le \frac{c \log_c e}{b-a} n^b.$$

So let $k = max(1, k_1)$ and $c_0 = \frac{c \log_c e}{b-a}$. Then:

$$\forall n \ge k \ \ (g(n) \le c_0 n^b).$$

Fact 6: Exponentials (including the fact that exponentials dominate polynomials):

1. **If $1 < a \le b$ then $\mathcal{O}(a^n) \subseteq \mathcal{O}(b^n)$.**

   If $g(n) \in \mathcal{O}(a^n)$, then there must exist k_1 and c_1 such that:

   $$\forall n \ge k_1 \ (g(n) \le c_1 a^n \le c_1 b^n) \qquad (\text{since } (a \le b) \rightarrow (a^n \le b^n)).$$

   So let $k = k_1$ and $c = c_1$. Then:

   $$\forall n \ge k \ \ (g(n) \le cb^n).$$

2. **If $a \ge 0$ and $b > 1$ then $\mathcal{O}(n^a) \subseteq \mathcal{O}(b^n)$.**

   If $a = 0$, then we have that $\mathcal{O}(1) \subseteq \mathcal{O}(b^n)$, which is trivially true. We now consider the case in which $a > 0$. First notice that, if $p \ge 1$ and $n \ge p$, then:

   $$\log_e n = \int_1^n \frac{1}{x}dx = \int_1^p \frac{1}{x}dx + \int_p^n \frac{1}{x}dx$$

   $$\le \log_e p + \int_p^n \frac{1}{p}dx$$

   $$\le \log_e p + \frac{n-p}{p}$$

   $$\le \log_e p + \frac{n}{p}.$$

If, in particular, $p = \max(\frac{a}{\log_e b}, 1)$, then $\frac{1}{p} \le \frac{\log_e}{a} b$, $p \ge 1$ and:

$$\log_e n \le \log_e p + \frac{\log_e b}{a} n$$

$$a \log_e n \le a \log_e p + \log_e b \cdot n .$$

And: $$n^a = e^{a \cdot \log_e n} \le e^{a \cdot \log_e p + \log_e b \cdot n}$$

$$\le p^a \cdot b^n .$$

If $g(n) \in \mathcal{O}(n^a)$, then there must exist k_1 and c_1 such that:

$$\forall n \ge k_1 \ (g(n) \le c_1 n^a).$$

So, again letting $p = \max(\frac{a}{\log_e b}, 1)$, let $k = max(k_1, p)$ and $c = c_1 p^a$. Then:

$$\forall n \ge k \ \ (g(n) \le c b^n).$$

3. If $f(n) = c_{j+1}2^n + c_j n^j + c_{j-1}n^{j-1} + \ldots c_1 n + c_0$, then $f(n) \in \mathcal{O}(2^n)$.

From 2, we have that $c_j n^j + c_{j-1}n^{j-1} + \ldots c_1 n + c_0 \in \mathcal{O}(n^j)$. From 2, we have that $n^j \in \mathcal{O}(2^n)$. So, using the transitivity property that we prove in 0, we have that $c_j n^j + c_{j-1}n^{j-1} + \ldots c_1 n + c_0 \in \mathcal{O}(2^n)$. So there must exist k_1 and c_1 such that:

$$\begin{aligned} \forall n \ge k_1 \ (c_j n^j + c_{j-1}n^{j-1} + \ldots c_1 n + c_0 &\le c_1 2^n \\ c_{j+1}2^n + c_j n^j + c_{j-1}n^{j-1} + \ldots c_1 n + c_0 &\le c_{j+1}2^n + c_1 2^n \\ &\le (c_{j+1} + c_1)2^n). \end{aligned}$$

So let $k = k_1$ and $c = c_{j+1} + c_1$. Then:

$$\forall n \ge k \ \ (f(n) \le c 2^n).$$

Fact 7: Factorial dominates exponentials: If $a \ge 1$ then $\mathcal{O}(a^n) \subseteq \mathcal{O}(n!)$.

First notice that, if $a \ge 1$, then:

$$a^n = \prod_{k=1}^{n} a = \prod_{k=1}^{\lceil a \rceil - 1} a \cdot \prod_{k=\lceil a \rceil}^{n} a \le \prod_{k=1}^{\lceil a \rceil - 1} a \cdot \prod_{k=\lceil a \rceil}^{n} k$$

$$\le \prod_{k=1}^{\lceil a \rceil - 1} a \cdot \frac{\prod_{k=1}^{\lceil a \rceil - 1} k}{\prod_{k=1}^{\lceil a \rceil - 1} k} \cdot \prod_{k=\lceil a \rceil}^{n} k$$

$$\le \prod_{k=1}^{\lceil a \rceil - 1} \frac{a}{k} \cdot \prod_{k=1}^{n} k$$

$$\leq \prod_{k=1}^{\lceil a \rceil - 1} \frac{a}{k} \cdot n! \cdot$$

If $g(n) \in \mathcal{O}(a^n)$, then there must exist k_1 and c_1 such that:

$$\forall n \geq k_1 \ (g(n) \leq c_1 a^n).$$

So let $k = k_1$ and $c = c_1 \prod_{k=1}^{\lceil a \rceil - 1} \frac{a}{k}$. Then:

$$\forall n \geq k \ \ (g(n) \leq cn!).$$

Fact 8: Transitivity: If $f(n) \in \mathcal{O}(f_1(n))$ and $f_1(n) \in \mathcal{O}(f_2(n))$ then $f(n) \in \mathcal{O}(f_2(n))$.

If $f(n) \in \mathcal{O}(f_1(n))$ and $f_1(n) \in \mathcal{O}(f_2(n))$, then there must exist k_1, c_1, k_2 and c_2 such that:

$$\forall n \geq k_1 \ (f(n) \leq c_1 f_1(n)).$$
$$\forall n \geq k_2 \ (f_1(n) \leq c_2 f_2(n)).$$

So let $k = max(k_1, k_2)$ and $c = c_1 c_2$. Then:

$$\forall n \geq k \ \ (f(n) \leq c f_2(n)).$$

■

Theorem 37.2 Facts about o

We will prove separately the two claims made in the theorem. The basis for these proofs is the definition of the relation o: $f(n) \in o(g(n))$ iff, for every positive c, there exists a positive integer k such that:

$$\forall n \geq k \ (f(n) < c \ g(n)).$$

Let f and g be functions from the natural numbers to the positive reals. Then:

1. $f(n) \notin o(f(n))$:

Let $c = 1$. Then there exists no k such that $\forall n \geq k \ (f(n) < c \ f(n)$, and $f(n) < f(n))$.

2. $o(f(n)) \subset \mathcal{O}(f(n))$:

If $g(n) \in o(f(n))$ then, for every positive c_1, there exists a k_1 such that: $\forall n \geq k \ (g(n) < c \ f(n))$. To show that $g(n) \in \mathcal{O}(f(n))$, it suffices to find a single c and k that satisfy the definition of $\mathcal{O}$. Let $c = 1$ and let k be the k_1 that must exist if $c_1 = 1$.

■

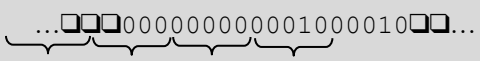
37.2 The Linear Speedup Theorem

In Section 27.5 we introduced the theory of asymptotic dominance so that we could describe the time and space requirements of Turing machines by the rate at which those requirements grow, rather than by some more exact measure of them. One consequence of this approach is that constant factors get ignored. This makes sense for two reasons. The first is that, in most of the problems we want to consider, such factors are dominated by much faster growing ones, so they have little impact on the size of the problems that we can reasonably solve.

But a second reason is the one that we will focus on here: All but the most efficient Turing machines can be sped up by any constant factor. The idea behind this claim is simple. At each step of its operation, a Turing machine visits one tape square. If we can compress the contents of the tape so that they fit on fewer squares, we can reduce the number of steps that a Turing machine must execute to process the tape. Compressing a tape is easy. We simply increase the size of the tape alphabet. That enables us to encode a chunk of squares from the original tape into a single square of the new one.

Example 37.1 Encoding Multiple Tape Squares as a Single Square

Let M be a Turing machine whose tape alphabet is {0, 1, ❑}. We can build a new Turing machine M' that uses the tape alphabet {A-Z, a-z, 0-9, α-ω}. Using the new alphabet, we can encode five squares of M's tape as a single square on the tape M' will use. Initially, we'll include in the encoding at least one blank on either side of the input (plus more on the right for padding if necessary). If M ever moves off its input, new squares can be encoded as necessary. So, for example:

If the tape of M is: 

Then the tape of M' might be: C 8 f π

If we design the transitions of M' appropriately, it will be able to do in one step what it takes M six steps to do.

The compression idea that we have just described is the basis of the Linear Speedup Theorem that we are about to present. Before we go into the details of the theorem and its proof, one caveat is in order. While this theorem is of some theoretical interest, its application to real computers is limited. Real computers have a fixed size alphabet (generally consisting of two binary symbols). So the sort of compression that we are using here cannot be applied. However, it is worth noting that other compression algorithms (that exploit patterns in particular kinds of input strings and are thus able to reduce the number of bits required) are routinely used in many kinds of real applications.

Theorem 37.3 The Linear Speedup Theorem

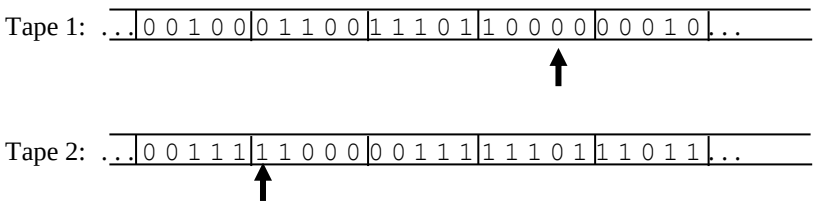
Theorem: Let M be a k-tape Turing machine where $k > 1$ and $timereq(M) = f(n)$. Given any constant $c > 0$, there exists another k-tape Turing machine M' such that $L(M) = L(M')$ and:

$$timereq(M') \le \left\lceil \frac{f(n)}{c} \right\rceil + 2n + 2 \cdot \lceil 6c \rceil \cdot$$

Notice that c is a factor by which we reduce the time a computation requires. So, if we want to say that the new program takes half as long, we set c to 2. In some statements of the theorem, c is a multiplicative factor. Using those versions, we'd set c to ½ in that case. The two formulations are equivalent.

Proof: We prove the claim by describing the operation of M'. M' will begin by making one pass across its input tape to encode it as described in Example 37.1. It will store the encoded string on tape 2 and blank out tape 1. During the rest of its operation, it will use tape 1 as M would have used tape 2 and vice versa. The number of symbols to be collapsed into one at this step is determined by c, the speedup that is desired. We'll call this collapsing factor m and set it to $\lceil 6c \rceil$.

Next M' simulates the execution of M. The idea here is that, since M' encodes m symbols as one, it can process m symbols as one. Unfortunately, on any one tape, the m symbols that M' needs may be spread across two of the new encoded tape squares: They may fall on the current square plus the one to its right or they may fall on the current square plus the one to its left. So M' must make one move to the left, then one back, and then one more to the right and then back before it can be sure that it has all the information it needs to make the move that simulates m moves of M. We illustrate this with the following example. Let $k = 2$ and $m = 5$. Each tape square actually contains a single symbol that encodes five original symbols, but we've shown them here with the original sequences so that it is possible to see how the simulation works. Suppose that, at some point in its computation, M has entered some state q and its tapes contain the following fragments:



If the next five moves of *M* move the read/write head on tape 1 to the right five times and they move the read/write head on tape 2 to the left five times, *M′* will need to examine both one encoded square to the left and one encoded square to the right before it will have enough information to simulate all five of those moves.

So *M′* simulates *M* by doing the following:

1. Move one square to the left on each of the tapes and record in the state the encoded symbol it finds on each of the *k* tapes.
2. Move one square back to the right on each tape and record in the state the encoded symbol it finds on each of the *k* tapes.
3. Move one more square to the right on each tape and record in the state the encoded symbol it finds on each of the *k* tapes.
4. Move one square back to the left on each tape.

   At this point, the read/write heads of *M′* are back where they started and the state of *M′* includes the following vector of information that captures the current state of *M*:

(q,	*M*'s state.
L_1, C_1, R_1, t_1,	The relevant contents of tape 1: L_1 is the encoded square to the left of the one that contains *M*'s simulated read/write head. C_1 is the encoded square that contains *M*'s simulated read/write head. R_1 is the encoded square to the right of the one that contains *M*'s simulated read/write head. t_1 (an integer between 1 and *m*) is the position within C_1 of *M*'s simulated read/write head.
L_2, C_2, R_2, t_2,	Tape 2: similarly.
…	
L_k, C_k, R_k, t_k)	Tape *k*: similarly

5. Using this information, make one move that alters the *C* squares as necessary and moves each read/write head as required to simulate *M*. Also update *M*'s state.
6. Make one more move if necessary. If, on some tape, *M*'s simulated read/write head moved off the *C* square, it will be necessary to make this second move in order to alter the contents of either the *L* or *R* square to match what *M* would have done. But note that, on any given tape, it will only be necessary to work with the current square plus one to the left *or* the current square plus one to the right. So two moves suffice to make all necessary changes to the tapes.

The first phase, encoding the input tape, requires that *M* make one complete pass through the input and then move back to the left. It may have to use up to *m* padding blanks. So, in the worst case, on an input of length *n* this phase requires $2(n+m)$ steps. The second phase, simulating *M*, requires at most six steps for every *m* steps that *M* would have executed. So, if $timereq(M) = f(n)$, then:

$$timereq(M') \leq \left\lceil \frac{6f(n)}{m} \right\rceil + 2(n+m).$$

Since $m = \lceil 6c \rceil$, we then have:

$$timereq(M') \leq \left\lceil \frac{f(n)}{c} \right\rceil + 2n + 2 \cdot \lceil 6c \rceil \cdot$$

■