Figure 34.10: Reducing circuit satisfiability to formula satisfiability. The formula produced by the reduction algorithm has a variable for each wire in the circuit.

The formula \( \varphi \) produced by the reduction algorithm is the AND of the circuit-output variable with the conjunction of clauses describing the operation of each gate. For the circuit in the figure, the formula is

\[
\varphi = x_{10} \land (x_4 \land x_3) \\
\land (x_5 \land (x_1 \lor x_2)) \\
\land (x_6 \land x_4) \\
\land (x_7 \land (x_1 \land x_2 \land x_4)) \\
\land (x_8 \land (x_5 \lor x_6)) \\
\land (x_9 \land (x_6 \lor x_7)) \\
\land (x_{10} \land (x_7 \land x_8 \land x_9)).
\]

Given a circuit \( C \), it is straightforward to produce such a formula \( \varphi \) in polynomial time.

Why is the circuit \( C \) satisfiable exactly when the formula \( \varphi \) is satisfiable? If \( C \) has a satisfying assignment, each wire of the circuit has a well-defined value, and the output of the circuit is 1. Therefore, the assignment of wire values to variables in \( \varphi \) makes each clause of \( \varphi \) evaluate to 1, and thus the conjunction of all evaluates to 1. Conversely, if there is an assignment that causes \( \varphi \) to evaluate to 1, the circuit \( C \) is satisfiable by an analogous argument. Thus, we have shown that \textsc{Circuit-SAT} \( \leq_p \) \textsc{Sat}, which completes the proof.

### 3-CNF satisfiability

Many problems can be proved NP-complete by reduction from formula satisfiability. The reduction algorithm must handle any input formula, though, and this requirement can lead to a huge number of cases that must be considered. It is often desirable, therefore, to reduce from a restricted language of boolean formulas, so that fewer cases need be considered. Of course, we must not restrict the language so much that it becomes polynomial-time solvable. One convenient language is 3-CNF satisfiability, or 3-CNF-SAT.

We define 3-CNF satisfiability using the following terms. A \textit{literal} in a boolean formula is an occurrence of a variable or its negation. A boolean formula is in \textit{conjunctive normal form}, or \textit{CNF}, if it is expressed as an AND of \textit{clauses}, each of which is the OR of one or more literals. A boolean formula is in \textit{3-conjunctive normal form}, or 3-CNF, if each clause has exactly three distinct literals.
For example, the boolean formula

\((x_1 \land \neg x_1 \land \neg x_2) \land (x_3 \land x_2 \land x_4) \land (\neg x_1 \land \neg x_3 \land \neg x_4)\)

is in 3-CNF. The first of its three clauses is \((x_1 \land \neg x_1 \land \neg x_2)\), which contains the three literals \(x_1\), \(x_2\), and \(\neg x_2\).

In 3-CNF-SAT, we are asked whether a given boolean formula \(\phi\) in 3-CNF is satisfiable. The following theorem shows that a polynomial-time algorithm that can determine the satisfiability of boolean formulas is unlikely to exist, even when they are expressed in this simple normal form.

**Theorem 34.10**

Satisfiability of boolean formulas in 3-conjunctive normal form is NP-complete.

**Proof** The argument we used in the proof of Theorem 34.9 to show that \(\text{SAT} \leq_p \text{NP}\) applies equally well here to show that \(\text{3-CNF-SAT} \leq_p \text{NP}\). Thus, by Lemma 34.8, we need only show that \(\text{SAT} \leq_p \text{3-CNF-SAT}\).

The reduction algorithm can be broken into three basic steps. Each step progressively transforms the input formula \(\phi\) closer to the desired 3-conjunctive normal form.

The first step is similar to the one used to prove \(\text{CIRCUIT-SAT} \leq_p \text{SAT}\) in Theorem 34.9. First, we construct a binary "parse" tree for the input formula \(\phi\), with literals as leaves and connectives as internal nodes. Figure 34.11 shows such a parse tree for the formula

\[(34.3) \phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2\]

Figure 34.11: The tree corresponding to the formula \(\phi = ((x_1 \rightarrow x_2) \lor \neg((\neg x_1 \leftrightarrow x_3) \lor x_4)) \land \neg x_2\).

Should the input formula contain a clause such as the OR of several literals, associativity can be used to parenthesize the expression fully so that every internal node in the resulting tree has 1 or 2 children. The binary parse tree can now be viewed as a circuit for computing the function.

Mimicking the reduction in the proof of Theorem 34.9, we introduce a variable \(y_i\) for the output of each internal node. Then, we rewrite the original formula \(\phi\) as the AND of the root
variable and a conjunction of clauses describing the operation of each node. For the formula (34.3), the resulting expression is

\[ \varphi' = y_1 \land (y_1 \leftrightarrow (y_2 \land \neg x_2)) \land (y_2 \leftrightarrow (y_3 \land y_4)) \land (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \land (y_4 \leftrightarrow y_5) \land (y_5 \leftrightarrow (y_6 \land x_4)) \land (y_6 \leftrightarrow (x_1 \rightarrow x_3)) \]

Observe that the formula \( \varphi' \) thus obtained is a conjunction of clauses \( \phi_i \), each of which has at most 3 literals. The only additional requirement is that each clause be an OR of literals.

The second step of the reduction converts each clause \( \phi_i \) into conjunctive normal form. We construct a truth table for \( \phi_i \) by evaluating all possible assignments to its variables. Each row of the truth table consists of a possible assignment of the variables of the clause, together with the value of the clause under that assignment. Using the truth-table entries that evaluate to 0, we build a formula in disjunctive normal form (or DNF)-an OR of AND's-that is equivalent to \( \neg \phi_i \). We then convert this formula into a CNF formula \( \phi_i \) by using DeMorgan's laws (equations (B.2)) to complement all literals and change OR's into AND's and AND's into OR's.

In our example, we convert the clause \( \phi'_i = (y_1 \leftrightarrow (y_2 \land \neg x_2)) \) into CNF as follows. The truth table for \( \phi_i \) is given in Figure 34.12. The DNF formula equivalent to \( \neg \phi_i \) is

\[
(y_1 \land y_2 \land \neg x_2) \lor (y_1 \land \neg y_2 \land x_2) \lor (y_1 \land \neg y_2 \land \neg x_2) \lor \neg y_1 \land y_2 \land \neg x_2.
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
 y_1 & y_2 & x_2 & (y_1 \leftrightarrow (y_2 \land \neg x_2)) \\
1 & 1 & 1 & 0 \\
1 & 1 & 0 & 1 \\
1 & 0 & 1 & 0 \\
1 & 0 & 0 & 0 \\
0 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 1 \\
0 & 0 & 0 & 1 \\
\end{array}
\]

Figure 34.12: The truth table for the clause \( y_1 \leftrightarrow (y_2 \land \neg x_2) \).

Applying DeMorgan's laws, we get the CNF formula
\[ \phi'_i = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \land (\neg y_1 \vee y_2 \vee \neg x_2) \land (\neg y_1 \vee y_2 \vee x_2) \land (y_1 \vee \neg y_2 \vee x_2) , \]

which is equivalent to the original clause \( \phi_i \).

Each clause \( \phi \) of the formula \( \varphi \) has now been converted into a CNF formula \( \phi'_v \), and thus \( \varphi \) is equivalent to the CNF formula \( \varphi'' \) consisting of the conjunction of the \( \phi'_v \). Moreover, each clause of \( \varphi'' \) has at most 3 literals.

The third and final step of the reduction further transforms the formula so that each clause has exactly 3 distinct literals. The final 3-CNF formula \( \varphi''' \) is constructed from the clauses of the CNF formula \( \varphi'' \). It also uses two auxiliary variables that we shall call \( p \) and \( q \). For each clause \( C_i \) of \( \varphi'' \), we include the following clauses in \( \varphi''' \):

- If \( C_i \) has 3 distinct literals, then simply include \( C_i \) as a clause of \( \varphi''' \).
- If \( C_i \) has 2 distinct literals, that is, if \( C_i = (l_1 \ 1 \ l_2) \), where \( l_1 \) and \( l_2 \) are literals, then include \( (l_1 \ 1 \ l_2 \ 1 \ p) \land (l_1 \ 1 \ l_2 \ \neg p) \) as clauses of \( \varphi''' \). The literals \( p \) and \( \neg p \) merely fulfill the syntactic requirement that there be exactly 3 distinct literals per clause: \( (l_1 \ 1 \ l_2 \ 1 \ p) \land (l_1 \ 1 \ l_2 \ \neg p) \) is equivalent to \( (l_1 \ 1 \ l_2) \) whether \( p = 0 \) or \( p = 1 \).
- If \( C_i \) has just 1 distinct literal \( l \), then include \( (l \ 1 \ p) \land (l \ 1 \ q) \land (l \ 1 \ \neg q) \land (l \ 1 \ \neg p) \) as clauses of \( \varphi''' \). Note that every setting of \( p \) and \( q \) causes the conjunction of these four clauses to evaluate to \( l \).

We can see that the 3-CNF formula \( \varphi''' \) is satisfiable if and only if \( \varphi \) is satisfiable by inspecting each of the three steps. Like the reduction from CIRCUIT-SAT to SAT, the construction of \( \varphi' \) from \( \varphi \) in the first step preserves satisfiability. The second step produces a CNF formula \( \varphi'' \) that is algebraically equivalent to \( \varphi' \). The third step produces a 3-CNF formula \( \varphi''' \) that is effectively equivalent to \( \varphi'' \), since any assignment to the variables \( p \) and \( q \) produces a formula that is algebraically equivalent to \( \varphi'' \).

We must also show that the reduction can be computed in polynomial time. Constructing \( \varphi' \) from \( \varphi \) introduces at most 1 variable and 1 clause per connective in \( \varphi \). Constructing \( \varphi'' \) from \( \varphi' \) can introduce at most 8 clauses into \( \varphi'' \) for each clause from \( \varphi' \), since each clause of \( \varphi' \) has at most 3 variables, and the truth table for each clause has at most \( 2^3 = 8 \) rows. The construction of \( \varphi''' \) from \( \varphi'' \) introduces at most 4 clauses into \( \varphi''' \) for each clause of \( \varphi'' \). Thus, the size of the resulting formula \( \varphi''' \) is polynomial in the length of the original formula. Each of the constructions can easily be accomplished in polynomial time.

Exercises 34.4-1

Consider the straightforward (nonpolynomial-time) reduction in the proof of Theorem 34.9. Describe a circuit of size \( n \) that, when converted to a formula by this method, yields a formula whose size is exponential in \( n \).

Exercises 34.4-2