the left-hand sides of these four sequents. This is the place where we rely on the law of the excluded middle which states $r \lor \neg r$, for any r. We use LEM for all propositional atoms (here p and q) and then we separately assume all the four cases, by using $\lor$e. That way we can invoke all four proofs of the sequents above and use the rule $\lor$e repeatedly until we have got rid of all our premises. We spell out the combination of these four phases schematically:

1	$p \lor \neg p$							LEM
2	p			ass	$\neg p$			ass
3	$q \lor \neg q$			LEM	$q \lor \neg q$			LEM
4	q	ass	$\neg q$	ass	q	ass	$\neg q$	ass
5	$\vdots$		$\vdots$		$\vdots$		$\vdots$	
6								
7	$p \land q \to p$		$p \land q \to p$		$p \land q \to p$		$p \land q \to p$	
8	$p \land q \to p$			$\lor$e	$p \land q \to p$			$\lor$e
9	$p \land q \to p$							$\lor$e

As soon as you understand how this particular example works, you will also realise that it will work for an arbitrary tautology with n distinct atoms. Of course, it seems ridiculous to prove $p \land q \to p$ using a proof that is this long. But remember that this illustrates a *uniform* method that constructs a proof for every tautology η, no matter how complicated it is.

Step 3: Finally, we need to find a proof for $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$. Take the proof for $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\dots(\phi_n \to \psi)\dots)))$ given by step 2 and augment its proof by introducing $\phi_1, \phi_2, \dots, \phi_n$ as premises. Then apply $\to$e n times on each of these premises (starting with ϕ_1, continuing with ϕ_2 etc.). Thus, we arrive at the conclusion ψ which gives us a proof for the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$.

Corollary 1.39 (Soundness and Completeness) *Let $\phi_1, \phi_2, \dots, \phi_n, \psi$ be formulas of propositional logic. Then $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ is holds iff the sequent $\phi_1, \phi_2, \dots, \phi_n \vdash \psi$ is valid.*

1.5 Normal forms

In the last section, we showed that our proof system for propositional logic is sound and complete for the truth-table semantics of formulas in Figure 1.6.

Soundness means that whatever we prove is going to be a true fact, based on the truth-table semantics. In the exercises, we apply this to show that a sequent does not have a proof: simply show that $\phi_1, \phi_2, \dots, \phi_2$ does not semantically entail ψ; then soundness implies that the sequent $\phi_1, \phi_2, \dots, \phi_2 \vdash \psi$ does not have a proof. Completeness comprised a much more powerful statement: no matter what (semantically) valid sequents there are, they all have syntactic proofs in the proof system of natural deduction. This tight correspondence allows us to freely switch between working with the notion of proofs ($\vdash$) and that of semantic entailment ($\vDash$).

Using natural deduction to decide the validity of instances of $\vdash$ is only one of many possibilities. In Exercise 1.2.6 we sketch a non-linear, tree-like, notion of proofs for sequents. Likewise, checking an instance of $\vDash$ by applying Definition 1.34 literally is only one of many ways of deciding whether $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds. We now investigate various alternatives for deciding $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ which are based on transforming these formulas syntactically into 'equivalent' ones upon which we can then settle the matter by purely syntactic or algorithmic means. This requires that we first clarify what exactly we mean by equivalent formulas.

1.5.1 Semantic equivalence, satisfiability and validity

Two formulas ϕ and ψ are said to be equivalent if they have the same 'meaning.' This suggestion is vague and needs to be refined. For example, $p \to q$ and $\neg p \lor q$ have the same truth table; all four combinations of T and F for p and q return the same result. 'Coincidence of truth tables' is not good enough for what we have in mind, for what about the formulas $p \land q \to p$ and $r \lor \neg r$? At first glance, they have little in common, having different atomic formulas and different connectives. Moreover, the truth table for $p \land q \to p$ is four lines long, whereas the one for $r \lor \neg r$ consists of only two lines. However, both formulas are always true. This suggests that we define the equivalence of formulas ϕ and ψ via $\vDash$: if ϕ semantically entails ψ and vice versa, then these formulas should be the same as far as our truth-table semantics is concerned.

Definition 1.40 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *semantically equivalent* iff $\phi \vDash \psi$ and $\psi \vDash \phi$ hold. In that case we write $\phi \equiv \psi$. Further, we call ϕ *valid* if $\vDash \phi$ holds.

Note that we could also have defined $\phi \equiv \psi$ to mean that $\vDash (\phi \to \psi) \land (\psi \to \phi)$ holds; it amounts to the same concept. Indeed, because of soundness and completeness, semantic equivalence is identical to *provable equivalence*

(Definition 1.25). Examples of equivalent formulas are

$$
\begin{aligned}
p \to q &\equiv \neg q \to \neg p \\
p \to q &\equiv \neg p \lor q \\
p \land q \to p &\equiv r \lor \neg r \\
p \land q \to r &\equiv p \to (q \to r).
\end{aligned}
$$

Recall that a formula η is called a tautology if $\vDash \eta$ holds, so the tautologies are exactly the valid formulas. The following lemma says that any decision procedure for tautologies is in fact a decision procedure for the validity of sequents as well.

Lemma 1.41 *Given formulas $\phi_1, \phi_2, \dots, \phi_n$ and ψ of propositional logic, $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds iff $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ holds.*

PROOF: First, suppose that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ holds. If $\phi_1, \phi_2, \dots, \phi_n$ are all true under some valuation, then ψ has to be true as well for that same valuation. Otherwise, $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ would not hold (compare this with Figure 1.11). Second, if $\phi_1, \phi_2, \dots, \phi_n \vDash \psi$ holds, we have already shown that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to \cdots \to (\phi_n \to \psi)))$ follows in step 1 of our completeness proof. $\square$

For our current purposes, we want to transform formulas into ones which don't contain $\to$ at all and the occurrences of $\land$ and $\lor$ are confined to separate layers such that validity checks are easy. This is being done by

1. using the equivalence $\phi \to \psi \equiv \neg\phi \lor \psi$ to remove all occurrences of $\to$ from a formula and
2. by specifying an algorithm that takes a formula without any $\to$ into a *normal form* (still without $\to$) for which checking validity is easy.

Naturally, we have to specify which forms of formulas we think of as being 'normal.' Again, there are many such notions, but in this text we study only two important ones.

Definition 1.42 A *literal* L is either an atom p or the negation of an atom $\neg p$. A formula C is in *conjunctive normal form* (CNF) if it is a conjunction of clauses, where each clause D is a disjunction of literals:

$$
\begin{aligned}
L &::= p \mid \neg p \\
D &::= L \mid L \lor D \\
C &::= D \mid D \land C.
\end{aligned}
\tag{1.6}
$$

Examples of formulas in conjunctive normal form are

$$(i) \quad (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q \qquad (ii) \quad (p \vee r) \wedge (\neg p \vee r) \wedge (p \vee \neg r).$$

In the first case, there are three clauses of type D: $\neg q \vee p \vee r$, $\neg p \vee r$, and q – which is a literal promoted to a clause by the first rule of clauses in (1.6). Notice how we made implicit use of the associativity laws for $\wedge$ and $\vee$, saying that $\phi \vee (\psi \vee \eta) \equiv (\phi \vee \psi) \vee \eta$ and $\phi \wedge (\psi \wedge \eta) \equiv (\phi \wedge \psi) \wedge \eta$, since we omitted some parentheses. The formula $(\neg(q \vee p) \vee r) \wedge (q \vee r)$ is not in CNF since $q \vee p$ is not a literal.

Why do we care at all about formulas ϕ in CNF? One of the reasons for their usefulness is that they allow easy checks of validity which otherwise take times exponential in the number of atoms. For example, consider the formula in CNF from above: $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$. The semantic entailment $\vDash (\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ holds iff all three relations

$$\vDash \neg q \vee p \vee r \qquad \vDash \neg p \vee r \qquad \vDash q$$

hold, by the semantics of $\wedge$. But since all of these formulas are disjunctions of literals, or literals, we can settle the matter as follows.

Lemma 1.43 *A disjunction of literals $L_1 \vee L_2 \vee \cdots \vee L_m$ is valid iff there are $1 \leq i, j \leq m$ such that L_i is $\neg L_j$.*

PROOF: If L_i equals $\neg L_j$, then $L_1 \vee L_2 \vee \cdots \vee L_m$ evaluates to T for all valuations. For example, the disjunct $p \vee q \vee r \vee \neg q$ can never be made false.

To see that the converse holds as well, assume that no literal L_k has a matching negation in $L_1 \vee L_2 \vee \cdots \vee L_m$. Then, for each k with $1 \leq k \leq n$, we assign F to L_k, if L_k is an atom; or T, if L_k is the negation of an atom. For example, the disjunct $\neg q \vee p \vee r$ can be made false by assigning F to p and r and T to q. $\square$

Hence, we have an easy and fast check for the validity of $\vDash \phi$, provided that ϕ is in CNF; inspect all conjuncts ψ_k of ϕ and search for atoms in ψ_k such that ψ_k also contains their negation. If such a match is found for all conjuncts, we have $\vDash \phi$. Otherwise (= some conjunct contains no pair L_i and $\neg L_i$), ϕ is not valid by the lemma above. Thus, the formula $(\neg q \vee p \vee r) \wedge (\neg p \vee r) \wedge q$ above is not valid. Note that the matching literal has to be found in the same conjunct ψ_k. Since there is no free lunch in this universe, we can expect that the computation of a formula ϕ' in CNF, which is equivalent to a given formula ϕ, is a costly worst-case operation.

Before we study how to compute equivalent conjunctive normal forms, we introduce another semantic concept closely related to that of validity.

Definition 1.44 Given a formula ϕ in propositional logic, we say that ϕ is *satisfiable* if it has a valuation in which is evaluates to T.

For example, the formula $p \lor q \to p$ is satisfiable since it computes T if we assign T to p. Clearly, $p \lor q \to p$ is not valid. Thus, satisfiability is a weaker concept since every valid formula is by definition also satisfiable but not vice versa. However, these two notions are just mirror images of each other, the mirror being negation.

Proposition 1.45 *Let ϕ be a formula of propositional logic. Then ϕ is satisfiable iff $\neg\phi$ is not valid.*

PROOF: First, assume that ϕ is satisfiable. By definition, there exists a valuation of ϕ in which ϕ evaluates to T; but that means that $\neg\phi$ evaluates to F for that same valuation. Thus, $\neg\phi$ cannot be valid.

Second, assume that $\neg\phi$ is not valid. Then there must be a valuation of $\neg\phi$ in which $\neg\phi$ evaluates to F. Thus, ϕ evaluates to T and is therefore satisfiable. (Note that the valuations of ϕ are exactly the valuations of $\neg\phi$.) □

This result is extremely useful since it essentially says that we need provide a decision procedure for only one of these concepts. For example, let's say that we have a procedure P for deciding whether any ϕ is valid. We obtain a decision procedure for satisfiability simply by asking P whether $\neg\phi$ is valid. If it is, ϕ is not satisfiable; otherwise ϕ is satisfiable. Similarly, we may transform any decision procedure for satisfiability into one for validity. We will encounter both kinds of procedures in this text.

There is one scenario in which computing an equivalent formula in CNF is really easy; namely, when someone else has already done the work of writing down a full truth table for ϕ. For example, take the truth table of $(p \to \neg q) \to (q \lor \neg p)$ in Figure 1.8 (page 40). For each line where $(p \to \neg q) \to (q \lor \neg p)$ computes F we now construct a disjunction of literals. Since there is only one such line, we have only one conjunct ψ_1. That conjunct is now obtained by a disjunction of literals, where we include literals $\neg p$ and q. Note that the literals are just the syntactic opposites of the truth values in that line: here p is T and q is F. The resulting formula in CNF is thus $\neg p \lor q$ which is readily seen to be in CNF and to be equivalent to $(p \to \neg q) \to (q \lor \neg p)$.

Why does this always work for any formula ϕ? Well, the constructed formula will be false iff at least one of its conjuncts ψ_i will be false. This means that all the disjuncts in such a ψ_i must be F. Using the de Morgan rule $\neg\phi_1 \vee \neg\phi_2 \vee \cdots \vee \neg\phi_n \equiv \neg(\phi_1 \wedge \phi_2 \wedge \cdots \wedge \phi_n)$, we infer that the conjunction of the syntactic opposites of those literals must be true. Thus, ϕ and the constructed formula have the same truth table.

Consider another example, in which ϕ is given by the truth table:

p	q	r	ϕ
T	T	T	T
T	T	F	F
T	F	T	T
T	F	F	T
F	T	T	F
F	T	F	F
F	F	T	F
F	F	F	T

Note that this table is really just a specification of ϕ; it does not tell us what ϕ looks like syntactically, but it does tells us how it ought to 'behave.' Since this truth table has four entries which compute F, we construct four conjuncts ψ_i $(1 \leq i \leq 4)$. We read the ψ_i off that table by listing the disjunction of all atoms, where we negate those atoms which are true in those lines:

$$\psi_1 \stackrel{\text{def}}{=} \neg p \vee \neg q \vee r \quad \text{(line 2)} \qquad \psi_2 \stackrel{\text{def}}{=} p \vee \neg q \vee \neg r \quad \text{(line 5)}$$
$$\psi_3 \stackrel{\text{def}}{=} p \vee \neg q \vee r \quad \text{etc} \qquad \psi_4 \stackrel{\text{def}}{=} p \vee q \vee \neg r.$$

The resulting ϕ in CNF is therefore

$$(\neg p \vee \neg q \vee r) \wedge (p \vee \neg q \vee \neg r) \wedge (p \vee \neg q \vee r) \wedge (p \vee q \vee \neg r).$$

If we don't have a full truth table at our disposal, but do know the structure of ϕ, then we would like to compute a version of ϕ in CNF. It should be clear by now that a full truth table of ϕ and an equivalent formula in CNF are pretty much the same thing as far as questions about validity are concerned – although the formula in CNF may be much more compact.

1.5.2 Conjunctive normal forms and validity

We have already seen the benefits of conjunctive normal forms in that they allow for a fast and easy syntactic test of validity. Therefore, one wonders whether any formula can be transformed into an *equivalent* formula in CNF. We now develop an algorithm achieving just that. Note that, by Definition 1.40, a formula is valid iff any of its equivalent formulas is valid. We reduce the problem of determining whether *any* ϕ is valid to the problem of computing an equivalent $\psi \equiv \phi$ such that ψ is in CNF and checking, via Lemma 1.43, whether ψ is valid.

Before we sketch such a procedure, we make some general remarks about its possibilities and its realisability constraints. First of all, there could be more or less efficient ways of computing such normal forms. But even more so, there could be many possible correct outputs, for $\psi_1 \equiv \phi$ and $\psi_2 \equiv \phi$ do not generally imply that ψ_1 is the same as ψ_2, even if ψ_1 and ψ_2 are in CNF. For example, take $\phi \stackrel{\text{def}}{=} p$, $\psi_1 \stackrel{\text{def}}{=} p$ and $\psi_2 \stackrel{\text{def}}{=} p \wedge (p \vee q)$; then convince yourself that $\phi \equiv \psi_2$ holds. Having this ambiguity of equivalent conjunctive normal forms, the computation of a CNF for ϕ with minimal 'cost' (where 'cost' could for example be the number of conjuncts, or the height of ϕ's parse tree) becomes a very important practical problem, an issue persued in Chapter 6. Right now, we are content with stating a *deterministic* algorithm which always computes the same output CNF for a given input ϕ.

This algorithm, called `CNF`, should satisfy the following requirements:

(1) `CNF` terminates for all formulas of propositional logic as input;
(2) for each such input, `CNF` outputs an equivalent formula; and
(3) all output computed by `CNF` is in CNF.

If a call of `CNF` with a formula ϕ of propositional logic as input terminates, which is enforced by (1), then (2) ensures that $\psi \equiv \phi$ holds for the output ψ. Thus, (3) guarantees that ψ is an equivalent CNF of ϕ. So ϕ is valid iff ψ is valid; and checking the latter is easy relative to the length of ψ.

What kind of strategy should `CNF` employ? It will have to function correctly for all, i.e. infinitely many, formulas of propositional logic. This strongly suggests to write a procedure that computes a CNF by structural induction on the formula ϕ. For example, if ϕ is of the form $\phi_1 \wedge \phi_2$, we may simply compute conjunctive normal forms η_i for ϕ_i $(i = 1, 2)$, whereupon $\eta_1 \wedge \eta_2$ is a conjunctive normal form which is equivalent to ϕ *provided that* $\eta_i \equiv \phi_i$ $(i = 1, 2)$. This strategy also suggests to use proof by structural induction on ϕ to prove that `CNF` meets the requirements (1–3) stated above.

Given a formula ϕ as input, we first do some *preprocessing.* Initially, we translate away all implications in ϕ by replacing all subformulas of the form $\psi \to \eta$ by $\neg\psi \vee \eta$. This is done by a procedure called `IMPL_FREE`. Note that this procedure has to be recursive, for there might be implications in ψ or η as well.

The application of `IMPL_FREE` might introduce double negations into the output formula. More importantly, negations whose scopes are non-atomic formulas might still be present. For example, the formula $p \wedge \neg(p \wedge q)$ has such a negation with $p \wedge q$ as its scope. Essentially, the question is whether one can efficiently compute a CNF for $\neg\phi$ from a CNF for ϕ. Since *nobody* seems to know the answer, we circumvent the question by translating $\neg\phi$

into an equivalent formula that contains only negations of atoms. Formulas which only negate atoms are said to be in *negation normal form* (NNF). We spell out such a procedure, `NNF`, in detail later on. The key to its specification for implication-free formulas lies in the de Morgan rules. The second phase of the preprocessing, therefore, calls `NNF` with the implication-free output of `IMPL_FREE` to obtain an equivalent formula in NNF.

After all this preprocessing, we obtain a formula ϕ' which is the result of the call $\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi))$. Note that $\phi' \equiv \phi$ since both algorithms only transform formulas into equivalent ones. Since ϕ' contains no occurrences of $\to$ and since only atoms in ϕ' are negated, we may program `CNF` by an analysis of only *three* cases: literals, conjunctions and disjunctions.

- If ϕ is a literal, it is by definition in CNF and so `CNF` outputs ϕ.
- If ϕ equals $\phi_1 \wedge \phi_2$, we call `CNF` recursively on each ϕ_i to get the respective output η_i and return the CNF $\eta_1 \wedge \eta_2$ as output for input ϕ.
- If ϕ equals $\phi_1 \vee \phi_2$, we again call `CNF` recursively on each ϕ_i to get the respective output η_i; but this time we must not simply return $\eta_1 \vee \eta_2$ since that formula is certainly *not* in CNF, unless η_1 and η_2 happen to be literals.

So how can we complete the program in the last case? Well, we may resort to the distributivity laws, which entitle us to translate any disjunction of conjunctions into a conjunction of disjunctions. However, for this to result in a CNF, we need to make certain that those disjunctions generated contain only literals. We apply a strategy for using distributivity based on matching patterns in $\phi_1 \vee \phi_2$. This results in an independent algorithm called `DISTR` which will do all that work for us. Thus, we simply call `DISTR` with the pair (η_1, η_2) as input and pass along its result.

Assuming that we already have written code for `IMPL_FREE`, `NNF` and `DISTR`, we may now write pseudo code for `CNF`:

```
function CNF ($\phi$):
/* precondition: $\phi$ implication free and in NNF */
/* postcondition: CNF ($\phi$) computes an equivalent CNF for $\phi$ */
begin function
  case
    $\phi$ is a literal:  return $\phi$
    $\phi$ is $\phi_1 \wedge \phi_2$:  return CNF ($\phi_1$) $\wedge$ CNF ($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$:  return DISTR (CNF ($\phi_1$), CNF ($\phi_2$))
  end case
end function
```

Notice how the calling of DISTR is done with the computed conjunctive normal forms of ϕ_1 and ϕ_2. The routine DISTR has η_1 and η_2 as input parameters and does a case analysis on whether these inputs are conjunctions. What should DISTR do if none of its input formulas is such a conjunction? Well, since we are calling DISTR for inputs η_1 and η_2 which are in CNF, this can only mean that η_1 and η_2 are literals, or disjunctions of literals. Thus, $\eta_1 \vee \eta_2$ is in CNF.

Otherwise, at least one of the formulas η_1 and η_2 is a conjunction. Since one conjunction suffices for simplifying the problem, we have to decide which conjunct we want to transform if *both* formulas are conjunctions. That way we maintain that our algorithm CNF is deterministic. So let us suppose that η_1 is of the form $\eta_{11} \wedge \eta_{12}$. Then the distributive law says that $\eta_1 \vee \eta_2 \equiv (\eta_{11} \vee \eta_2) \wedge (\eta_{12} \vee \eta_2)$. Since all participating formulas η_{11}, η_{12} and η_2 are in CNF, we may call DISTR again for the pairs (η_{11}, η_2) and (η_{12}, η_2), and then simply form their conjunction. This is the key insight for writing the function DISTR.

The case when η_2 is a conjunction is symmetric and the structure of the recursive call of DISTR is then dictated by the equivalence $\eta_1 \vee \eta_2 \equiv (\eta_1 \vee \eta_{21}) \wedge (\eta_1 \vee \eta_{22})$, where $\eta_2 = \eta_{21} \wedge \eta_{22}$:

```
function DISTR ($\eta_1, \eta_2$):
/* precondition: $\eta_1$ and $\eta_2$ are in CNF */
/* postcondition: DISTR ($\eta_1, \eta_2$) computes a CNF for $\eta_1 \vee \eta_2$ */
begin function
  case
    $\eta_1$ is $\eta_{11} \wedge \eta_{12}$:  return DISTR ($\eta_{11}, \eta_2$) $\wedge$ DISTR ($\eta_{12}, \eta_2$)
    $\eta_2$ is $\eta_{21} \wedge \eta_{22}$:  return DISTR ($\eta_1, \eta_{21}$) $\wedge$ DISTR ($\eta_1, \eta_{22}$)
    otherwise (= no conjunctions):  return $\eta_1 \vee \eta_2$
  end case
end function
```

Notice how the three clauses are exhausting all possibilities. Furthermore, the first and second cases overlap if η_1 and η_2 are both conjunctions. It is then our understanding that this code will inspect the clauses of a case statement from the top to the bottom clause. Thus, the first clause would apply.

Having specified the routines CNF and DISTR, this leaves us with the task of writing the functions IMPL_FREE and NNF. We delegate the design of IMPL_FREE to the exercises. The function NNF has to transform any implication-free formula into an equivalent one in negation normal form. Four examples of formulas in NNF are

$$\begin{array}{ll} p & \neg p \\ \neg p \wedge (p \wedge q) & \neg p \wedge (p \rightarrow q), \end{array}$$

although we won't have to deal with a formula of the last kind since $\rightarrow$ won't occur. Examples of formulas which are not in NNF are $\neg\neg p$ and $\neg(p \wedge q)$.

Again, we program NNF recursively by a case analysis over the structure of the input formula ϕ. The last two examples already suggest a solution for two of these clauses. In order to compute a NNF of $\neg\neg\phi$, we simply compute a NNF of ϕ. This is a sound strategy since ϕ and $\neg\neg\phi$ are semantically equivalent. If ϕ equals $\neg(\phi_1 \wedge \phi_2)$, we use the de Morgan rule $\neg(\phi_1 \wedge \phi_2) \equiv \neg\phi_1 \vee \neg\phi_2$ as a recipe for how NNF should call itself recursively in that case. Dually, the case of ϕ being $\neg(\phi_1 \vee \phi_2)$ appeals to the other de Morgan rule $\neg(\phi_1 \vee \phi_2) \equiv \neg\phi_1 \wedge \neg\phi_2$ and, if ϕ is a conjunction or disjunction, we simply let NNF pass control to those subformulas. Clearly, all literals are in NNF. The resulting code for NNF is thus

```
function NNF ($\phi$):
/* precondition: $\phi$ is implication free */
/* postcondition: NNF ($\phi$) computes a NNF for $\phi$ */
begin function
  case
    $\phi$ is a literal:  return $\phi$
    $\phi$ is $\neg\neg\phi_1$:  return NNF ($\phi_1$)
    $\phi$ is $\phi_1 \wedge \phi_2$:  return NNF ($\phi_1$) $\wedge$ NNF ($\phi_2$)
    $\phi$ is $\phi_1 \vee \phi_2$:  return NNF ($\phi_1$) $\vee$ NNF ($\phi_2$)
    $\phi$ is $\neg(\phi_1 \wedge \phi_2)$:  return NNF ($\neg\phi_1$) $\vee$ NNF ($\neg\phi_2$)
    $\phi$ is $\neg(\phi_1 \vee \phi_2)$:  return NNF ($\neg\phi_1$) $\wedge$ NNF ($\neg\phi_2$)
  end case
end function
```

Notice that these cases are exhaustive due to the algorithm's precondition. Given any formula ϕ of propositional logic, we may now convert it into an equivalent CNF by calling $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$. In the exercises, you are asked to show that

- all four algorithms terminate on input meeting their preconditions,
- the result of $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\phi)))$ is in CNF and
- that result is semantically equivalent to ϕ.

We will return to the important issue of formally proving the correctness of programs in Chapter 4.

Let us now illustrate the programs coded above on some concrete examples. We begin by computing $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\neg p \land q \to p \land (r \to q))))$. We show almost all details of this computation and you should compare this with how you would expect the code above to behave. First, we compute $\texttt{IMPL\_FREE}\ (\phi)$:

$$\begin{aligned}
\texttt{IMPL\_FREE}\,(\phi) &= \neg\texttt{IMPL\_FREE}\,(\neg p \land q) \lor \texttt{IMPL\_FREE}\,(p \land (r \to q))\\
&= \neg((\texttt{IMPL\_FREE}\,\neg p) \land (\texttt{IMPL\_FREE}\,q)) \lor \texttt{IMPL\_FREE}\,(p \land (r \to q))\\
&= \neg((\neg p) \land \texttt{IMPL\_FREE}\,q) \lor \texttt{IMPL\_FREE}\,(p \land (r \to q))\\
&= \neg(\neg p \land q) \lor \texttt{IMPL\_FREE}\,(p \land (r \to q))\\
&= \neg(\neg p \land q) \lor ((\texttt{IMPL\_FREE}\,p) \land \texttt{IMPL\_FREE}\,(r \to q))\\
&= \neg(\neg p \land q) \lor (p \land \texttt{IMPL\_FREE}\,(r \to q))\\
&= \neg(\neg p \land q) \lor (p \land (\neg(\texttt{IMPL\_FREE}\,r) \lor (\texttt{IMPL\_FREE}\,q)))\\
&= \neg(\neg p \land q) \lor (p \land (\neg r \lor (\texttt{IMPL\_FREE}\,q)))\\
&= \neg(\neg p \land q) \lor (p \land (\neg r \lor q)).
\end{aligned}$$

Second, we compute $\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi)$:

$$\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg(\neg p \land q)) \lor \texttt{NNF}\ (p \land (\neg r \lor q))\\
&= \texttt{NNF}\ (\neg(\neg p) \lor \neg q) \lor \texttt{NNF}\ (p \land (\neg r \lor q))\\
&= (\texttt{NNF}\ (\neg\neg p)) \lor (\texttt{NNF}\ (\neg q)) \lor \texttt{NNF}\ (p \land (\neg r \lor q))\\
&= (p \lor (\texttt{NNF}\ (\neg q))) \lor \texttt{NNF}\ (p \land (\neg r \lor q))\\
&= (p \lor \neg q) \lor \texttt{NNF}\ (p \land (\neg r \lor q))\\
&= (p \lor \neg q) \lor ((\texttt{NNF}\ p) \land (\texttt{NNF}\ (\neg r \lor q)))\\
&= (p \lor \neg q) \lor (p \land (\texttt{NNF}\ (\neg r \lor q)))\\
&= (p \lor \neg q) \lor (p \land ((\texttt{NNF}\ (\neg r)) \lor (\texttt{NNF}\ q)))\\
&= (p \lor \neg q) \lor (p \land (\neg r \lor (\texttt{NNF}\ q)))\\
&= (p \lor \neg q) \lor (p \land (\neg r \lor q)).
\end{aligned}$$

Third, we finish it off with

$$\begin{aligned}
\texttt{CNF}\ (\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi)) &= \texttt{CNF}\ ((p \lor \neg q) \lor (p \land (\neg r \lor q)))\\
&= \texttt{DISTR}\ (\texttt{CNF}\ (p \lor \neg q), \texttt{CNF}\ (p \land (\neg r \lor q)))\\
&= \texttt{DISTR}\ (p \lor \neg q, \texttt{CNF}\ (p \land (\neg r \lor q)))\\
&= \texttt{DISTR}\ (p \lor \neg q, p \land (\neg r \lor q))\\
&= \texttt{DISTR}\ (p \lor \neg q, p) \land \texttt{DISTR}\ (p \lor \neg q, \neg r \lor q)\\
&= (p \lor \neg q \lor p) \land \texttt{DISTR}\ (p \lor \neg q, \neg r \lor q)\\
&= (p \lor \neg q \lor p) \land (p \lor \neg q \lor \neg r \lor q)\ .
\end{aligned}$$

The formula $(p \lor \neg q \lor p) \land (p \lor \neg q \lor \neg r \lor q)$ is thus the result of the call $\texttt{CNF}\ (\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi))$ and is in conjunctive normal form and equivalent to ϕ. Note that it is satisfiable (choose p to be true) but not valid (choose p to be false and q to be true); it is also equivalent to the simpler conjunctive normal form $p \lor \neg q$. Observe that our algorithm does not do such optimisations so one would need a separate optimiser running on the output. Alternatively, one might change the code of our functions to allow for such optimisations 'on the fly,' a computational overhead which could prove to be counter-productive.

You should realise that we omitted several computation steps in the sub-calls $\texttt{CNF}\ (p \lor \neg q)$ and $\texttt{CNF}\ (p \land (\neg r \lor q))$. They return their input as a result since the input is already in conjunctive normal form.

As a second example, consider $\phi \stackrel{\text{def}}{=} r \to (s \to (t \land s \to r))$. We compute

$$\begin{aligned}
\texttt{IMPL\_FREE}\,(\phi) &= \neg(\texttt{IMPL\_FREE}\ r) \lor \texttt{IMPL\_FREE}\,(s \to (t \land s \to r))\\
&= \neg r \lor \texttt{IMPL\_FREE}\,(s \to (t \land s \to r))\\
&= \neg r \lor (\neg(\texttt{IMPL\_FREE}\,s) \lor \texttt{IMPL\_FREE}\,(t \land s \to r))\\
&= \neg r \lor (\neg s \lor \texttt{IMPL\_FREE}\,(t \land s \to r))\\
&= \neg r \lor (\neg s \lor (\neg(\texttt{IMPL\_FREE}\,(t \land s)) \lor \texttt{IMPL\_FREE}\,r))\\
&= \neg r \lor (\neg s \lor (\neg((\texttt{IMPL\_FREE}\,t) \land (\texttt{IMPL\_FREE}\,s)) \lor \texttt{IMPL\_FREE}\,r))\\
&= \neg r \lor (\neg s \lor (\neg(t \land (\texttt{IMPL\_FREE}\,s)) \lor (\texttt{IMPL\_FREE}\,r)))\\
&= \neg r \lor (\neg s \lor (\neg(t \land s)) \lor (\texttt{IMPL\_FREE}\,r))\\
&= \neg r \lor (\neg s \lor (\neg(t \land s)) \lor r)
\end{aligned}$$

$$\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL\_FREE}\ \phi) &= \texttt{NNF}\ (\neg r \vee (\neg s \vee \neg(t \wedge s) \vee r)) \\
&= (\texttt{NNF}\ \neg r) \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r) \\
&= \neg r \vee \texttt{NNF}\ (\neg s \vee \neg(t \wedge s) \vee r) \\
&= \neg r \vee (\texttt{NNF}\ (\neg s) \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r)) \\
&= \neg r \vee (\neg s \vee \texttt{NNF}\ (\neg(t \wedge s) \vee r)) \\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg(t \wedge s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee (\texttt{NNF}\ (\neg t \vee \neg s)) \vee \texttt{NNF}\ r) \\
&= \neg r \vee (\neg s \vee ((\texttt{NNF}\ (\neg t) \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \texttt{NNF}\ (\neg s)) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee \texttt{NNF}\ r)) \\
&= \neg r \vee (\neg s \vee ((\neg t \vee \neg s) \vee r))
\end{aligned}$$

where the latter is already in CNF and valid as r has a matching $\neg r$.

1.5.3 Horn clauses and satisfiability

We have already commented on the computational price we pay for transforming a propositional logic formula into an equivalent CNF. The latter class of formulas has an easy syntactic check for validity, but its test for satisfiability is very hard in general. Fortunately, there are practically important subclasses of formulas which have much more efficient ways of deciding their satisfiability. One such example is the class of *Horn formulas*; the name 'Horn' is derived from the logician A. Horn's last name. We shortly define them and give an algorithm for checking their satisfiability.

Recall that the logical constants $\perp$ ('bottom') and $\top$ ('top') denote an unsatisfiable formula, respectively, a tautology.

Definition 1.46 A *Horn formula* is a formula ϕ of propositional logic if it can be generated as an instance of H in this grammar:

$$\begin{aligned}
P &::= \perp \ \mid\ \top \ \mid\ p \\
A &::= P \ \mid\ P \wedge A \\
C &::= A \rightarrow P \\
H &::= C \ \mid\ C \wedge H.
\end{aligned} \tag{1.7}$$

We call each instance of C a *Horn clause*.

Horn formulas are conjunctions of Horn clauses. A Horn clause is an implication whose assumption A is a conjunction of propositions of type P and whose conclusion is also of type P. Examples of Horn formulas are

$$(p \land q \land s \to p) \land (q \land r \to p) \land (p \land s \to s)$$
$$(p \land q \land s \to \bot) \land (q \land r \to p) \land (\top \to s)$$
$$(p_2 \land p_3 \land p_5 \to p_{13}) \land (\top \to p_5) \land (p_5 \land p_{11} \to \bot).$$

Examples of formulas which are *not* Horn formulas are

$$(p \land q \land s \to \neg p) \land (q \land r \to p) \land (p \land s \to s)$$
$$(p \land q \land s \to \bot) \land (\neg q \land r \to p) \land (\top \to s)$$
$$(p_2 \land p_3 \land p_5 \to p_{13} \land p_{27}) \land (\top \to p_5) \land (p_5 \land p_{11} \to \bot)$$
$$(p_2 \land p_3 \land p_5 \to p_{13} \land p_{27}) \land (\top \to p_5) \land (p_5 \land p_{11} \lor \bot).$$

The first formula is not a Horn formula since $\neg p$, the conclusion of the implication of the first conjunct, is not of type P. The second formula does not qualify since the premise of the implication of the second conjunct, $\neg q \land r$, is not a conjunction of atoms, $\bot$, or $\top$. The third formula is not a Horn formula since the conclusion of the implication of the first conjunct, $p_{13} \land p_{27}$, is not of type P. The fourth formula clearly is not a Horn formula since it is not a conjunction of implications.

The algorithm we propose for deciding the satisfiability of a Horn formula ϕ maintains a list of all occurrences of type P in ϕ and proceeds like this:

1. It marks $\top$ if it occurs in that list.
2. If there is a conjunct $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ of ϕ such that all P_j with $1 \leq j \leq k_i$ are marked, mark P' as well and go to 2. Otherwise (= there is no conjunct $P_1 \land P_2 \land \cdots \land P_{k_i} \to P'$ such that all P_j are marked) go to 3.
3. If $\bot$ is marked, print out 'The Horn formula ϕ is unsatisfiable.' and stop. Otherwise, go to 4.
4. Print out 'The Horn formula ϕ is satisfiable.' and stop.

In these instructions, the markings of formulas are *shared* by all other occurrences of these formulas in the Horn formula. For example, once we mark p_2 because of one of the criteria above, then all other occurrences of p_2 are marked as well. We use pseudo code to specify this algorithm formally:

```
function HORN ($\phi$):
/* precondition: $\phi$ is a Horn formula */
/* postcondition: HORN ($\phi$) decides the satisfiability for $\phi$ */
begin function
        mark all occurrences of $\top$ in $\phi$;
        while there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of $\phi$
                such that all $P_j$ are marked but $P'$ isn't do
                mark $P'$
        end while
        if $\perp$ is marked then return 'unsatisfiable' else return 'satisfiable'
end function
```

We need to make sure that this algorithm terminates on all Horn formulas ϕ as input and that its output (= its decision) is always correct.

Theorem 1.47 *The algorithm* `HORN` *is correct for the satisfiability decision problem of Horn formulas and has no more than $n + 1$ cycles in its while-statement if n is the number of atoms in ϕ. In particular,* `HORN` *always terminates on correct input.*

PROOF: Let us first consider the question of program termination. Notice that entering the body of the while-statement has the effect of marking an unmarked P which is not $\top$. Since this marking applies to all occurrences of P in ϕ, the while-statement can have at most one more cycle than there are atoms in ϕ.

Since we guaranteed termination, it suffices to show that the answers given by the algorithm `HORN` are always correct. To that end, it helps to reveal the functional role of those markings. Essentially, marking a P means that that P has got to be true if the formula ϕ is ever going to be satisfiable. We use mathematical induction to show that

'All marked P are true for all valuations in which ϕ evaluates to T.' (1.8)

holds after any number of executions of the body of the while-statement above. The base case, zero executions, is when the while-statement has not yet been entered but we already and only marked all occurrences of $\top$. Since $\top$ must be true in all valuations, (1.8) follows.

In the inductive step, we assume that (1.8) holds after k cycles of the while-statement. Then we need to show that same assertion for all marked P after $k + 1$ cycles. If we enter the $(k + 1)$th cycle, the condition of the while-statement is certainly true. Thus, there exists a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ such that all P_j are marked. Let v be any valuation in which ϕ is true. By our induction hypothesis, we know that all P_j and therefore $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i}$ have to be true in v as well. The conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ has be to true in v, too, from which we infer that P' has to be true in v.

By mathematical induction, we therefore secured that (1.8) holds no matter how many cycles that while-statement went through.

Finally, we need to make sure that the if-statement above always renders correct replies. First, if $\perp$ is marked, then there has to be some conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to \perp$ of ϕ such that all P_i are marked as well. By (1.8) that conjunct of ϕ evaluates to $\mathtt{T} \to \mathtt{F} = \mathtt{F}$ whenever ϕ is true. As this is impossible the reply 'unsatisfiable' is correct. Second, if $\perp$ is not marked, we simply assign T to all marked atoms and F to all unmarked atoms and use proof by contradiction to show that ϕ has to be true with respect to that valuation.

If ϕ is *not* true under that valuation, it must make one of its principal conjuncts $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ false. By the semantics of implication this can only mean that all P_j are true and P' is false. By the definition of our valuation, we then infer that all P_j are marked, so $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ is a conjunct of ϕ that would have been dealt with in one of the cycles of the while-statement and so P' is marked, too. Since $\perp$ is not marked, P' has to be $\top$ or some atom q. In any event, the conjunct is then true by (1.8), a contradiction □

Note that the proof by contradiction employed in the last proof was not really needed. It just made the argument seem more natural to us. The literature is full of such examples where one uses proof by contradiction more out of psychological than proof-theoretical necessity.

1.6 SAT solvers

The marking algorithm for Horn formulas computes marks as constraints on all valuations that can make a formule true. By (1.8), all marked atoms have to be true for any such valuation. We can extend this idea to general formulas ϕ by computing constraints saying which subformulas of ϕ require a certain truth value for all valuations that make ϕ true:

$$\text{'All marked subformulas evaluate to their mark value for all valuations in which } \phi \text{ evaluates to } \mathtt{T}.\text{'} \tag{1.9}$$

In that way, marking atomic formulas generalizes to marking subformulas; and 'true' marks generalize into 'true' and 'false' marks. At the same