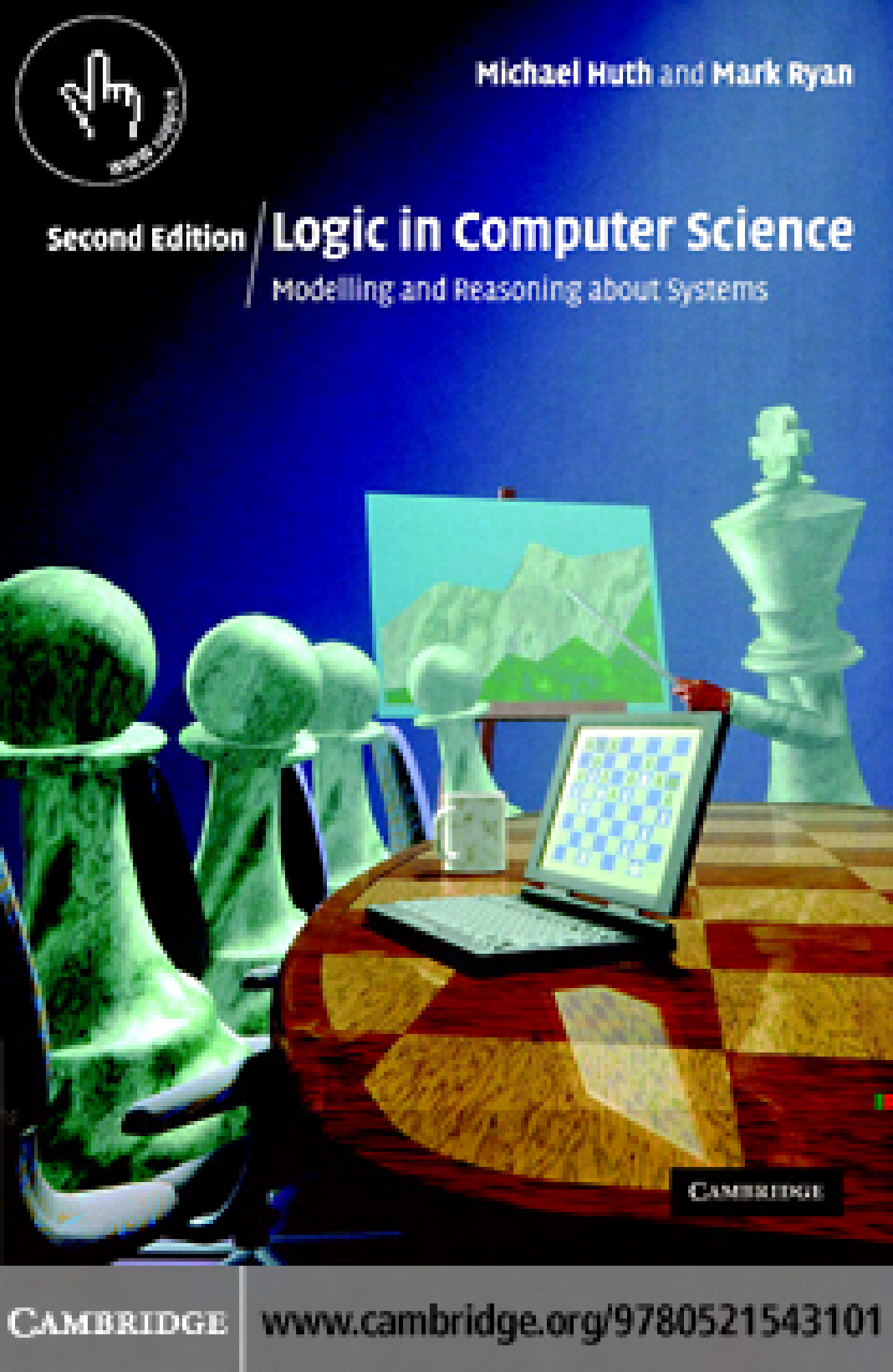

Michael Huth and Mark Ryan
Second Edition
Logic in Computer Science
Modelling and Reasoning about Systems
CAMBRIDGE
CAMBRIDGE
www.cambridge.org/9780521543101

This page intentionally left blank

LOGIC IN COMPUTER SCIENCE

Modelling and Reasoning about Systems

LOGIC IN COMPUTER SCIENCE

Modelling and Reasoning about Systems

MICHAEL HUTH
Department of Computing
Imperial College London, United Kingdom

MARK RYAN
School of Computer Science
University of Birmingham, United Kingdom

CAMBRIDGE
UNIVERSITY PRESS

CAMBRIDGE UNIVERSITY PRESS
Cambridge, New York, Melbourne, Madrid, Cape Town, Singapore, São Paulo

Cambridge University Press
The Edinburgh Building, Cambridge CB2 8RU, UK

Published in the United States of America by Cambridge University Press, New York

www.cambridge.org
Information on this title: www.cambridge.org/9780521543101

First published in print format 2004

ISBN-13 978-0-511-26401-6 eBook (EBL)
ISBN-10 0-511-26401-1 eBook (EBL)

ISBN-13 978-0-521-54310-1 paperback
ISBN-10 0-521-54310-X paperback

Contents

Foreword to the first edition *page* ix
Preface to the second edition xi
Acknowledgements xiii
1 Propositional logic 1
1.1 Declarative sentences 2
1.2 Natural deduction 5
1.2.1 Rules for natural deduction 6
1.2.2 Derived rules 23
1.2.3 Natural deduction in summary 26
1.2.4 Provable equivalence 29
1.2.5 An aside: proof by contradiction 29
1.3 Propositional logic as a formal language 31
1.4 Semantics of propositional logic 36
1.4.1 The meaning of logical connectives 36
1.4.2 Mathematical induction 40
1.4.3 Soundness of propositional logic 45
1.4.4 Completeness of propositional logic 49
1.5 Normal forms 53
1.5.1 Semantic equivalence, satisfiability and validity 54
1.5.2 Conjunctive normal forms and validity 58
1.5.3 Horn clauses and satisfiability 65
1.6 SAT solvers 68
1.6.1 A linear solver 69
1.6.2 A cubic solver 72
1.7 Exercises 78
1.8 Bibliographic notes 91
2 Predicate logic 93
2.1 The need for a richer language 93

2.2 Predicate logic as a formal language 98
2.2.1 Terms 99
2.2.2 Formulas 100
2.2.3 Free and bound variables 102
2.2.4 Substitution 104
2.3 Proof theory of predicate logic 107
2.3.1 Natural deduction rules 107
2.3.2 Quantifier equivalences 117
2.4 Semantics of predicate logic 122
2.4.1 Models 123
2.4.2 Semantic entailment 129
2.4.3 The semantics of equality 130
2.5 Undecidability of predicate logic 131
2.6 Expressiveness of predicate logic 136
2.6.1 Existential second-order logic 139
2.6.2 Universal second-order logic 140
2.7 Micromodels of software 141
2.7.1 State machines 142
2.7.2 Alma – re-visited 146
2.7.3 A software micromodel 148
2.8 Exercises 157
2.9 Bibliographic notes 170
3 Verification by model checking 172
3.1 Motivation for verification 172
3.2 Linear-time temporal logic 175
3.2.1 Syntax of LTL 175
3.2.2 Semantics of LTL 178
3.2.3 Practical patterns of specifications 183
3.2.4 Important equivalences between LTL formulas 184
3.2.5 Adequate sets of connectives for LTL 186
3.3 Model checking: systems, tools, properties 187
3.3.1 Example: mutual exclusion 187
3.3.2 The NuSMV model checker 191
3.3.3 Running NuSMV 194
3.3.4 Mutual exclusion revisited 195
3.3.5 The ferryman 199
3.3.6 The alternating bit protocol 203
3.4 Branching-time logic 207
3.4.1 Syntax of CTL 208

3.4.2 Semantics of CTL 211
3.4.3 Practical patterns of specifications 215
3.4.4 Important equivalences between CTL formulas 215
3.4.5 Adequate sets of CTL connectives 216
3.5 CTL* and the expressive powers of LTL and CTL 217
3.5.1 Boolean combinations of temporal formulas in CTL 220
3.5.2 Past operators in LTL 221
3.6 Model-checking algorithms 221
3.6.1 The CTL model-checking algorithm 222
3.6.2 CTL model checking with fairness 230
3.6.3 The LTL model-checking algorithm 232
3.7 The fixed-point characterisation of CTL 238
3.7.1 Monotone functions 240
3.7.2 The correctness of $\mathtt{SAT}_{\mathtt{EG}}$ 242
3.7.3 The correctness of $\mathtt{SAT}_{\mathtt{EU}}$ 243
3.8 Exercises 245
3.9 Bibliographic notes 254
4 Program verification 256
4.1 Why should we specify and verify code? 257
4.2 A framework for software verification 258
4.2.1 A core programming language 259
4.2.2 Hoare triples 262
4.2.3 Partial and total correctness 265
4.2.4 Program variables and logical variables 268
4.3 Proof calculus for partial correctness 269
4.3.1 Proof rules 269
4.3.2 Proof tableaux 273
4.3.3 A case study: minimal-sum section 287
4.4 Proof calculus for total correctness 292
4.5 Programming by contract 296
4.6 Exercises 299
4.7 Bibliographic notes 304
5 Modal logics and agents 306
5.1 Modes of truth 306
5.2 Basic modal logic 307
5.2.1 Syntax 307
5.2.2 Semantics 308
5.3 Logic engineering 316
5.3.1 The stock of valid formulas 317

5.3.2 Important properties of the accessibility relation 320
5.3.3 Correspondence theory 322
5.3.4 Some modal logics 326
5.4 Natural deduction 328
5.5 Reasoning about knowledge in a multi-agent system 331
5.5.1 Some examples 332
5.5.2 The modal logic KT45^n 335
5.5.3 Natural deduction for KT45^n 339
5.5.4 Formalising the examples 342
5.6 Exercises 350
5.7 Bibliographic notes 356
6 Binary decision diagrams 358
6.1 Representing boolean functions 358
6.1.1 Propositional formulas and truth tables 359
6.1.2 Binary decision diagrams 361
6.1.3 Ordered BDDs 366
6.2 Algorithms for reduced OBDDs 372
6.2.1 The algorithm `reduce` 372
6.2.2 The algorithm `apply` 373
6.2.3 The algorithm `restrict` 377
6.2.4 The algorithm `exists` 377
6.2.5 Assessment of OBDDs 380
6.3 Symbolic model checking 382
6.3.1 Representing subsets of the set of states 383
6.3.2 Representing the transition relation 385
6.3.3 Implementing the functions $\mathrm{pre}_\exists$ and $\mathrm{pre}_\forall$ 387
6.3.4 Synthesising OBDDs 387
6.4 A relational mu-calculus 390
6.4.1 Syntax and semantics 390
6.4.2 Coding CTL models and specifications 393
6.5 Exercises 398
6.6 Bibliographic notes 413
Bibliography 414
Index 418

Foreword to the first edition

by

Edmund M. Clarke
FORE Systems Professor of Computer Science
Carnegie Mellon University
Pittsburgh, PA

Formal methods have finally come of age! Specification languages, theorem provers, and model checkers are beginning to be used routinely in industry. Mathematical logic is basic to all of these techniques. Until now textbooks on logic for computer scientists have not kept pace with the development of tools for hardware and software specification and verification. For example, in spite of the success of model checking in verifying sequential circuit designs and communication protocols, until now I did not know of a single text, suitable for undergraduate and beginning graduate students, that attempts to explain how this technique works. As a result, this material is rarely taught to computer scientists and electrical engineers who will need to use it as part of their jobs in the near future. Instead, engineers avoid using formal methods in situations where the methods would be of genuine benefit or complain that the concepts and notation used by the tools are complicated and unnatural. This is unfortunate since the underlying mathematics is generally quite simple, certainly no more difficult than the concepts from mathematical analysis that every calculus student is expected to learn.

Logic in Computer Science by Huth and Ryan is an exceptional book. I was amazed when I looked through it for the first time. In addition to propositional and predicate logic, it has a particularly thorough treatment of temporal logic and model checking. In fact, the book is quite remarkable in how much of this material it is able to cover: linear and branching time temporal logic, explicit state model checking, fairness, the basic fixpoint

theorems for computation tree logic (CTL), even binary decision diagrams and symbolic model checking. Moreover, this material is presented at a level that is accessible to undergraduate and beginning graduate students. Numerous problems and examples are provided to help students master the material in the book. Since both Huth and Ryan are active researchers in logics of programs and program verification, they write with considerable authority.

In summary, the material in this book is up-to-date, practical, and elegantly presented. The book is a wonderful example of what a modern text on logic for computer science should be like. I recommend it to the reader with greatest enthusiasm and predict that the book will be an enormous success.

(This foreword is re-printed in the second edition with its author's permission.)

Preface to the second edition

Our motivation for (re)writing this book

One of the leitmotifs of writing the first edition of our book was the observation that most logics used in the design, specification and verification of computer systems fundamentally deal with a *satisfaction relation*

$$\mathcal{M} \models \phi$$

where $\mathcal{M}$ is some sort of *situation* or *model* of a system, and ϕ is a specification, a formula of that logic, expressing what should be true in situation $\mathcal{M}$. At the heart of this set-up is that one can often specify and implement algorithms for computing $\models$. We developed this theme for propositional, first-order, temporal, modal, and program logics. Based on the encouraging feedback received from five continents we are pleased to hereby present the second edition of this text which means to preserve and improve on the original intent of the first edition.

What's new and what's gone

Chapter 1 now discusses the design, correctness, and complexity of a SAT solver (a marking algorithm similar to Stålmarck's method [SS90]) for full propositional logic.

Chapter 2 now contains basic results from model theory (Compactness Theorem and Löwenheim–Skolem Theorem); a section on the transitive closure and the expressiveness of existential and universal second-order logic; and a section on the use of the object modelling language Alloy and its analyser for specifying and exploring under-specified first-order logic models with respect to properties written in first-order logic with transitive closure. The Alloy language is executable which makes such exploration interactive and formal.

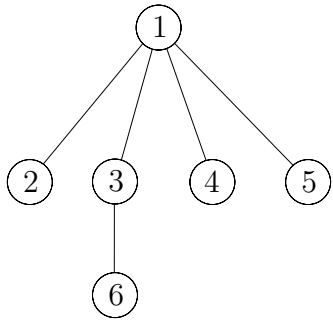
Chapter 3 has been completely restructured. It now begins with a discussion of linear-time temporal logic; features the open-source NuSMV model-checking tool throughout; and includes a discussion on planning problems, more material on the expressiveness of temporal logics, and new modelling examples.

Chapter 4 contains more material on total correctness proofs and a new section on the programming-by-contract paradigm of verifying program correctness.

Chapters 5 and 6 have also been revised, with many small alterations and corrections.

The interdependence of chapters and prerequisites

The book requires that students know the basics of elementary arithmetic and naive set theoretic concepts and notation. The core material of Chapter 1 (everything except Sections 1.4.3 to 1.6.2) is essential for all of the chapters that follow. Other than that, only Chapter 6 depends on Chapter 3 and a basic understanding of the static scoping rules covered in Chapter 2 – although one may easily cover Sections 6.1 and 6.2 without having done Chapter 3 at all. Roughly, the interdependence diagram of chapters is



WWW page

This book is supported by a Web page, which contains a list of errata; text files for all the program code; ancillary technical material and links; all the figures; an interactive tutor based on multiple-choice questions; and details of how instructors can obtain the solutions to exercises in this book which are marked with a $*$. The URL for the book's page is `www.cs.bham.ac.uk/research/lics/`. See also `www.cambridge.org/052154310x`

Acknowledgements

Many people have, directly or indirectly, assisted us in writing this book. David Schmidt kindly provided serveral exercises for Chapter 4. Krysia Broda has pointed out some typographical errors and she and the other authors of [BEKV94] have allowed us to use some exercises from that book. We have also borrowed exercises or examples from [Hod77] and [FHMV95]. Susan Eisenbach provided a first description of the Package Dependency System that we model in Alloy in Chapter 2. Daniel Jackson make very helpful comments on versions of that section. Zena Matilde Ariola, Josh Hodas, Jan Komorowski, Sergey Kotov, Scott A. Smolka and Steve Vickers have corresponded with us about this text; their comments are appreciated. Matt Dwyer and John Hatcliff made useful comments on drafts of Chapter 3. Kevin Lucas provided insightful comments on the content of Chapter 6, and notified us of numerous typographical errors in several drafts of the book. Achim Jung read several chapters and gave useful feedback.

Additionally, a number of people read and provided useful comments on several chapters, including Moti Ben-Ari, Graham Clark, Christian Haack, Anthony Hook, Roberto Segala, Alan Sexton and Allen Stoughton. Numerous students at Kansas State University and the University of Birmingham have given us feedback of various kinds, which has influenced our choice and presentation of the topics. We acknowledge Paul Taylor's LaTeX package for proof boxes. About half a dozen anonymous referees made critical, but constructive, comments which helped to improve this text in various ways. In spite of these contributions, there may still be errors in the book, and we alone must take responsibility for those.

Added for second edition

Many people have helped improve this text by pointing out typos and making other useful comments after the publication date. Among them,

we mention Wolfgang Ahrendt, Yasuhiro Ajiro, Torben Amtoft, Stephan Andrei, Bernhard Beckert, Jonathan Brown, James Caldwell, Ruchira Datta, Amy Felty, Dimitar Guelev, Hirotsugu Kakugawa, Kamran Kashef, Markus Krötzsch, Jagun Kwon, Ranko Lazic, David Makinson, Alexander Miczo, Aart Middeldorp, Robert Morelli, Prakash Panangaden, Aileen Paraguya, Frank Pfenning, Shekhar Pradhan, Koichi Takahashi, Kazunori Ueda, Hiroshi Watanabe, Fuzhi Wang and Reinhard Wilhelm.

1
Propositional logic

The aim of logic in computer science is to develop languages to model the situations we encounter as computer science professionals, in such a way that we can reason about them formally. Reasoning about situations means constructing arguments about them; we want to do this formally, so that the arguments are valid and can be defended rigorously, or executed on a machine.

Consider the following argument:

Example 1.1 If the train arrives late and there are no taxis at the station, then John is late for his meeting. John is not late for his meeting. The train did arrive late. *Therefore*, there were taxis at the station.

Intuitively, the argument is valid, since if we put the *first* sentence and the *third* sentence together, they tell us that if there are no taxis, then John will be late. The second sentence tells us that he was not late, so it must be the case that there were taxis.

Much of this book will be concerned with arguments that have this structure, namely, that consist of a number of sentences followed by the word 'therefore' and then another sentence. The argument is valid if the sentence after the 'therefore' logically follows from the sentences before it. Exactly what we mean by 'follows from' is the subject of this chapter and the next one.

Consider another example:

Example 1.2 If it is raining and Jane does not have her umbrella with her, then she will get wet. Jane is not wet. It is raining. *Therefore*, Jane has her umbrella with her.

This is also a valid argument. Closer examination reveals that it actually has the same structure as the argument of the previous example! All we have

done is substituted some sentence fragments for others:

Example 1.1	Example 1.2
the train is late	it is raining
there are taxis at the station	Jane has her umbrella with her
John is late for his meeting	Jane gets wet.

The argument in each example could be stated without talking about trains and rain, as follows:

If p and not q, then r. Not r. p. *Therefore*, q.

In developing logics, we are not concerned with what the sentences really mean, but only in their logical structure. Of course, when we *apply* such reasoning, as done above, such meaning will be of great interest.

1.1 Declarative sentences

In order to make arguments rigorous, we need to develop a language in which we can express sentences in such a way that brings out their logical structure. The language we begin with is the language of propositional logic. It is based on *propositions*, or *declarative sentences* which one can, in principle, argue as being true or false. Examples of declarative sentences are:

(1) The sum of the numbers 3 and 5 equals 8.
(2) Jane reacted violently to Jack's accusations.
(3) Every even natural number >2 is the sum of two prime numbers.
(4) All Martians like pepperoni on their pizza.
(5) Albert Camus était un écrivain français.
(6) Die Würde des Menschen ist unantastbar.

These sentences are all declarative, because they are in principle capable of being declared 'true', or 'false'. Sentence (1) can be tested by appealing to basic facts about arithmetic (and by tacitly assuming an Arabic, decimal representation of natural numbers). Sentence (2) is a bit more problematic. In order to give it a truth value, we need to know who Jane and Jack are and perhaps to have a reliable account from someone who witnessed the situation described. In principle, e.g., if we had been at the scene, we feel that we would have been able to detect Jane's *violent* reaction, provided that it indeed occurred in that way. Sentence (3), known as Goldbach's conjecture, seems straightforward on the face of it. Clearly, a fact about *all* even numbers >2 is either true or false. But to this day nobody knows whether sentence (3) expresses a truth or not. It is even not clear whether this could be shown by some finite means, even if it were true. However, in

this text we will be content with sentences as soon as they can, in principle, attain some truth value regardless of whether this truth value reflects the actual state of affairs suggested by the sentence in question. Sentence (4) seems a bit silly, although we could say that *if* Martians exist and eat pizza, then all of them will either like pepperoni on it or not. (We have to introduce predicate logic in Chapter 2 to see that this sentence is also declarative if *no* Martians exist; it is then true.) Again, for the purposes of this text sentence (4) will do. Et alors, qu'est-ce qu'on pense des phrases (5) et (6)? Sentences (5) and (6) are fine if you happen to read French and German a bit. Thus, declarative statements can be made in any natural, or artificial, language.

The kind of sentences we *won't* consider here are non-declarative ones, like

- Could you please pass me the salt?
- Ready, steady, go!
- May fortune come your way.

Primarily, we are interested in precise declarative sentences, or *statements* about the behaviour of computer systems, or programs. Not only do we want to specify such statements but we also want to *check* whether a given program, or system, fulfils a specification at hand. Thus, we need to develop a calculus of reasoning which allows us to draw conclusions from given assumptions, like initialised variables, which are reliable in the sense that they preserve truth: if all our assumptions are true, then our conclusion ought to be true as well. A much more difficult question is whether, given any true property of a computer program, we can find an argument in our calculus that has this property as its conclusion. The declarative sentence (3) above might illuminate the problematic aspect of such questions in the context of number theory.

The logics we intend to design are *symbolic* in nature. We translate a certain sufficiently large subset of all English declarative sentences into strings of symbols. This gives us a compressed but still complete encoding of declarative sentences and allows us to concentrate on the mere mechanics of our argumentation. This is important since specifications of systems or software are sequences of such declarative sentences. It further opens up the possibility of automatic manipulation of such specifications, a job that computers just love to do[1]. Our strategy is to consider certain declarative sentences as

[1] There is a certain, slightly bitter, circularity in such endeavours: in proving that a certain computer program `P` satisfies a given property, we might let some other computer program `Q` try to find a proof that `P` satisfies the property; but who guarantees us that `Q` satisfies the property of producing only correct proofs? We seem to run into an infinite regress.

being *atomic*, or *indecomposable*, like the sentence

'The number 5 is even.'

We assign certain distinct symbols $p, q, r, \ldots$, or sometimes $p_1, p_2, p_3, \ldots$ to each of these atomic sentences and we can then code up more complex sentences in a *compositional* way. For example, given the atomic sentences

p: 'I won the lottery last week.'
q: 'I purchased a lottery ticket.'
r: 'I won last week's sweepstakes.'

we can form more complex sentences according to the rules below:

$\neg$: The *negation* of p is denoted by $\neg p$ and expresses 'I did **not** win the lottery last week,' or equivalently 'It is **not** true that I won the lottery last week.'

$\vee$: Given p and r we may wish to state that *at least one of them* is true: 'I won the lottery last week, **or** I won last week's sweepstakes;' we denote this declarative sentence by $p \vee r$ and call it the *disjunction* of p and r[2].

$\wedge$: Dually, the formula $p \wedge r$ denotes the rather fortunate *conjunction* of p and r: 'Last week I won the lottery **and** the sweepstakes.'

$\rightarrow$: Last, but definitely not least, the sentence '**If** I won the lottery last week, **then** I purchased a lottery ticket.' expresses an *implication* between p and q, suggesting that q is a logical consequence of p. We write $p \rightarrow q$ for that[3]. We call p the *assumption* of $p \rightarrow q$ and q its *conclusion*.

Of course, we are entitled to use these rules of constructing propositions repeatedly. For example, we are now in a position to form the proposition

$$p \wedge q \rightarrow \neg r \vee q$$

which means that '**if** p **and** q **then not** r **or** q'. You might have noticed a potential ambiguity in this reading. One could have argued that this sentence has the structure 'p is the case **and if** q **then** ...' A computer would require the insertion of brackets, as in

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

[2] Its meaning should not be confused with the often implicit meaning of **or** in natural language discourse as **either** ... **or**. In this text **or** always means *at least one of them* and should not be confounded with *exclusive or* which states that *exactly one* of the two statements holds.

[3] The natural language meaning of '**if** ... **then** ...' often implicitly assumes a *causal role* of the assumption somehow enabling its conclusion. The logical meaning of implication is a bit different, though, in the sense that it states the *preservation of truth* which might happen without any causal relationship. For example, 'If all birds can fly, then Bob Dole was never president of the United States of America.' is a true statement, but there is no known causal connection between the flying skills of penguins and effective campaigning.

to disambiguate this assertion. However, we humans get annoyed by a proliferation of such brackets which is why we adopt certain conventions about the *binding priorities* of these symbols.

Convention 1.3 $\neg$ binds more tightly than $\vee$ and $\wedge$, and the latter two bind more tightly than $\rightarrow$. Implication $\rightarrow$ is *right-associative*: expressions of the form $p \rightarrow q \rightarrow r$ denote $p \rightarrow (q \rightarrow r)$.

1.2 Natural deduction

How do we go about constructing a calculus for reasoning about propositions, so that we can establish the validity of Examples 1.1 and 1.2? Clearly, we would like to have a set of rules each of which allows us to draw a conclusion given a certain arrangement of premises.

In natural deduction, we have such a collection of *proof rules*. They allow us to *infer* formulas from other formulas. By applying these rules in succession, we may infer a conclusion from a set of premises.

Let's see how this works. Suppose we have a set of formulas[4] ϕ_1, ϕ_2, ϕ_3, $\ldots$, ϕ_n, which we will call *premises*, and another formula, ψ, which we will call a *conclusion*. By applying proof rules to the premises, we hope to get some more formulas, and by applying more proof rules to those, to eventually obtain the conclusion. This intention we denote by

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi.$$

This expression is called a *sequent*; it is *valid* if a proof for it can be found. The sequent for Examples 1.1 and 1.2 is $p \wedge \neg q \rightarrow r, \neg r, p \vdash q$. Constructing such a proof is a creative exercise, a bit like programming. It is not necessarily obvious which rules to apply, and in what order, to obtain the desired conclusion. Additionally, our proof rules should be carefully chosen; otherwise, we might be able to 'prove' invalid patterns of argumentation. For

[4] It is traditional in logic to use Greek letters. Lower-case letters are used to stand for formulas and upper-case letters are used for sets of formulas. Here are some of the more commonly used Greek letters, together with their pronunciation:

Lower-case		Upper-case	
ϕ	phi	Φ	Phi
ψ	psi	Ψ	Psi
χ	chi	Γ	Gamma
η	eta	Δ	Delta
α	alpha		
β	beta		
γ	gamma		

example, we expect that we won't be able to show the sequent $p, q \vdash p \wedge \neg q$. For example, if p stands for 'Gold is a metal.' and q for 'Silver is a metal,' then knowing these two facts should not allow us to infer that 'Gold is a metal whereas silver isn't.'

Let's now look at our proof rules. We present about fifteen of them in total; we will go through them in turn and then summarise at the end of this section.

1.2.1 Rules for natural deduction

The rules for conjunction Our first rule is called the rule for conjunction ($\wedge$): and-introduction. It allows us to conclude $\phi \wedge \psi$, given that we have already concluded ϕ and ψ separately. We write this rule as

$$\frac{\phi \qquad \psi}{\phi \wedge \psi} \wedge\text{i}.$$

Above the line are the two premises of the rule. Below the line goes the conclusion. (It might not yet be the final conclusion of our argument; we might have to apply more rules to get there.) To the right of the line, we write the name of the rule; $\wedge$i is read 'and-introduction'. Notice that we have introduced a $\wedge$ (in the conclusion) where there was none before (in the premises).

For each of the connectives, there is one or more rules to introduce it and one or more rules to eliminate it. The rules for and-elimination are these two:

$$\frac{\phi \wedge \psi}{\phi} \wedge\text{e}_1 \qquad \frac{\phi \wedge \psi}{\psi} \wedge\text{e}_2. \tag{1.1}$$

The rule $\wedge\text{e}_1$ says: if you have a proof of $\phi \wedge \psi$, then by applying this rule you can get a proof of ϕ. The rule $\wedge\text{e}_2$ says the same thing, but allows you to conclude ψ instead. Observe the dependences of these rules: in the first rule of (1.1), the conclusion ϕ has to match the first conjunct of the premise, whereas the exact nature of the second conjunct ψ is irrelevant. In the second rule it is just the other way around: the conclusion ψ has to match the second conjunct ψ and ϕ can be any formula. It is important to engage in this kind of *pattern matching* before the application of proof rules.

Example 1.4 Let's use these rules to prove that $p \wedge q, r \vdash q \wedge r$ is valid. We start by writing down the premises; then we leave a gap and write the

conclusion:

$$p \wedge q$$
$$r$$

$$q \wedge r$$

The task of constructing the proof is to fill the gap between the premises and the conclusion by applying a suitable sequence of proof rules. In this case, we apply $\wedge e_2$ to the first premise, giving us q. Then we apply $\wedge i$ to this q and to the second premise, r, giving us $q \wedge r$. That's it! We also usually number all the lines, and write in the justification for each line, producing this:

1	$p \wedge q$	premise
2	r	premise
3	q	$\wedge e_2$ 1
4	$q \wedge r$	$\wedge i$ 3, 2

Demonstrate to yourself that you've understood this by trying to show on your own that $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ is valid. Notice that the ϕ and ψ can be instantiated not just to atomic sentences, like p and q in the example we just gave, but also to compound sentences. Thus, from $(p \wedge q) \wedge r$ we can deduce $p \wedge q$ by applying $\wedge e_1$, instantiating ϕ to $p \wedge q$ and ψ to r.

If we applied these proof rules literally, then the proof above would actually be a tree with root $q \wedge r$ and leaves $p \wedge q$ and r, like this:

$$\dfrac{\dfrac{p \wedge q}{q}\,\wedge e_2 \qquad r}{q \wedge r}\,\wedge i$$

However, we flattened this tree into a linear presentation which necessitates the use of pointers as seen in lines 3 and 4 above. These pointers allow us to recreate the actual proof tree. Throughout this text, we will use the flattened version of presenting proofs. That way you have to concentrate only on finding a proof, not on how to fit a growing tree onto a sheet of paper.

If a sequent is valid, there may be many different ways of proving it. So if you compare your solution to these exercises with those of others, they need not coincide. The important thing to realise, though, is that any putative proof can be *checked* for correctness by checking each individual line, starting at the top, for the valid application of its proof rule.

The rules of double negation Intuitively, there is no difference between a formula ϕ and its *double negation* $\neg\neg\phi$, which expresses no more and nothing less than ϕ itself. The sentence

'It is **not** true that it does **not** rain.'

is just a more contrived way of saying

'It rains.'

Conversely, knowing 'It rains,' we are free to state this fact in this more complicated manner if we wish. Thus, we obtain rules of elimination and introduction for double negation:

$$\frac{\neg\neg\phi}{\phi}\,\neg\neg\text{e} \qquad \frac{\phi}{\neg\neg\phi}\,\neg\neg\text{i}.$$

(There are rules for single negation on its own, too, which we will see later.)

Example 1.5 The proof of the sequent $p, \neg\neg(q \wedge r) \vdash \neg\neg p \wedge r$ below uses most of the proof rules discussed so far:

1	p	premise
2	$\neg\neg(q \wedge r)$	premise
3	$\neg\neg p$	$\neg\neg$i 1
4	$q \wedge r$	$\neg\neg$e 2
5	r	$\wedge\text{e}_2$ 4
6	$\neg\neg p \wedge r$	$\wedge$i 3, 5

Example 1.6 We now prove the sequent $(p \wedge q) \wedge r,\ s \wedge t \vdash q \wedge s$ which you were invited to prove by yourself in the last section. Please compare the proof below with your solution:

1	$(p \wedge q) \wedge r$	premise
2	$s \wedge t$	premise
3	$p \wedge q$	$\wedge\text{e}_1$ 1
4	q	$\wedge\text{e}_2$ 3
5	s	$\wedge\text{e}_1$ 2
6	$q \wedge s$	$\wedge$i 4, 5

The rule for eliminating implication There is one rule to introduce $\rightarrow$ and one to eliminate it. The latter is one of the best known rules of propositional logic and is often referred to by its Latin name *modus ponens*. We will usually call it by its modern name, implies-elimination (sometimes also referred to as arrow-elimination). This rule states that, given ϕ and knowing that ϕ implies ψ, we may rightfully conclude ψ. In our calculus, we write this as

$$\frac{\phi \quad \phi \rightarrow \psi}{\psi} \rightarrow\text{e}.$$

Let us justify this rule by spelling out instances of some declarative sentences p and q. Suppose that

$$\begin{aligned} p &: \text{It rained.} \\ p \rightarrow q &: \text{If it rained, then the street is wet.} \end{aligned}$$

so q is just 'The street is wet.' Now, *if* we know that it rained and *if* we know that the street is wet in the case that it rained, then we may combine these two pieces of information to conclude that the street is indeed wet. Thus, the justification of the $\rightarrow$e rule is a mere application of common sense. Another example from programming is:

$$\begin{aligned} p &: \text{The value of the program's input is an integer.} \\ p \rightarrow q &: \text{If the program's input is an integer, then the program outputs a boolean.} \end{aligned}$$

Again, we may put all this together to conclude that our program outputs a boolean value if supplied with an integer input. However, it is important to realise that the presence of p is absolutely essential for the inference to happen. For example, our program might well satisfy $p \rightarrow q$, but if it doesn't satisfy p – e.g. if its input is a surname – then we will not be able to derive q.

As we saw before, the formal parameters ϕ and the ψ for $\rightarrow$e can be instantiated to any sentence, including compound ones:

1	$\neg p \wedge q$	premise
2	$\neg p \wedge q \rightarrow r \vee \neg p$	premise
3	$r \vee \neg p$	$\rightarrow$e 2, 1

Of course, we may use any of these rules as often as we wish. For example, given p, $p \to q$ and $p \to (q \to r)$, we may infer r:

1	$p \to (q \to r)$	premise
2	$p \to q$	premise
3	p	premise
4	$q \to r$	$\to$e 1, 3
5	q	$\to$e 2, 3
6	r	$\to$e 4, 5

Before turning to implies-introduction, let's look at a hybrid rule which has the Latin name *modus tollens*. It is like the $\to$e rule in that it eliminates an implication. Suppose that $p \to q$ *and* $\neg q$ are the case. Then, *if* p holds we can use $\to$e to conclude that q holds. Thus, we then have that q *and* $\neg q$ hold, which is impossible. Therefore, we may infer that p must be false. But this can only mean that $\neg p$ is true. We summarise this reasoning into the rule *modus tollens*, or MT for short:[5]

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ \text{MT}.$$

Again, let us see an example of this rule in the natural language setting:

> *'If Abraham Lincoln was Ethiopian, then he was African. Abraham Lincoln was not African; therefore he was not Ethiopian.'*

Example 1.7 In the following proof of

$$p \to (q \to r),\ p,\ \neg r \vdash \neg q$$

we use several of the rules introduced so far:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	$q \to r$	$\to$e 1, 2
5	$\neg q$	MT 4, 3

[5] We will be able to *derive* this rule from other ones later on, but we introduce it here because it allows us already to do some pretty slick proofs. You may think of this rule as one on a higher level insofar as it does not mention the lower-level rules upon which it depends.

Examples 1.8 Here are two example proofs which combine the rule MT with either ¬¬e or ¬¬i:

1	$\neg p \to q$	premise
2	$\neg q$	premise
3	$\neg\neg p$	MT 1, 2
4	p	¬¬e 3

proves that the sequent $\neg p \to q,\ \neg q \vdash p$ is valid; and

1	$p \to \neg q$	premise
2	q	premise
3	$\neg\neg q$	¬¬i 2
4	$\neg p$	MT 1, 3

shows the validity of the sequent $p \to \neg q,\ q \vdash \neg p$.

Note that the order of applying double negation rules and MT is different in these examples; this order is driven by the structure of the particular sequent whose validity one is trying to show.

The rule implies introduction The rule MT made it possible for us to show that $p \to q,\ \neg q \vdash \neg p$ is valid. But the validity of the sequent $p \to q \vdash \neg q \to \neg p$ seems just as plausible. That sequent is, in a certain sense, saying the same thing. Yet, so far we have no rule which *builds* implications that do not already occur as premises in our proofs. The mechanics of such a rule are more involved than what we have seen so far. So let us proceed with care. Let us suppose that $p \to q$ is the case. If we *temporarily* assume that $\neg q$ holds, we can use MT to infer $\neg p$. Thus, assuming $p \to q$ we can show that $\neg q$ **implies** $\neg p$; but the latter we express *symbolically* as $\neg q \to \neg p$. To summarise, we have found an argumentation for $p \to q \vdash \neg q \to \neg p$:

1	$p \to q$	premise
2	$\neg q$	assumption
3	$\neg p$	MT 1, 2
4	$\neg q \to \neg p$	→i 2–3

The box in this proof serves to demarcate the scope of the temporary assumption $\neg q$. What we are saying is: let's make the assumption of $\neg q$. To

do this, we open a box and put $\neg q$ at the top. Then we continue applying other rules as normal, for example to obtain $\neg p$. But this still depends on the assumption of $\neg q$, so it goes inside the box. Finally, we are ready to apply $\rightarrow$i. It allows us to conclude $\neg q \rightarrow \neg p$, but that conclusion no longer *depends* on the assumption $\neg q$. Compare this with saying that 'If you are French, then you are European.' The truth of this sentence does not depend on whether anybody is French or not. Therefore, we write the conclusion $\neg q \rightarrow \neg p$ outside the box.

This works also as one would expect if we think of $p \rightarrow q$ as a *type* of a procedure. For example, p could say that the procedure expects an integer value x as input and q might say that the procedure returns a boolean value y as output. The validity of $p \rightarrow q$ amounts now to an assume-guarantee assertion: if the input is an integer, then the output is a boolean. This assertion can be true about a procedure while that same procedure could compute strange things or crash in the case that the input is not an integer. Showing $p \rightarrow q$ using the rule $\rightarrow$i is now called *type checking*, an important topic in the construction of compilers for typed programming languages.

We thus formulate the rule $\rightarrow$i as follows:

$$\frac{\boxed{\begin{array}{c}\phi \\ \vdots \\ \psi\end{array}}}{\phi \rightarrow \psi}\ \rightarrow\text{i}.$$

It says: in order to prove $\phi \rightarrow \psi$, make a temporary assumption of ϕ and then prove ψ. In your proof of ψ, you can use ϕ and any of the other formulas such as premises and provisional conclusions that you have made so far. Proofs may nest boxes or open new boxes after old ones have been closed. There are rules about which formulas can be used at which points in the proof. Generally, we can only use a formula ϕ in a proof at a given point if that formula occurs *prior* to that point and if no box which encloses that occurrence of ϕ has been closed already.

The line immediately following a closed box has to match the pattern of the conclusion of the rule that uses the box. For implies-introduction, this means that we have to continue after the box with $\phi \rightarrow \psi$, where ϕ was the first and ψ the last formula of that box. We will encounter two more proof rules involving proof boxes and they will require similar pattern matching.

Example 1.9 Here is another example of a proof using →i:

1	$\neg q \to \neg p$	premise
2	p	assumption
3	$\neg\neg p$	¬¬i 2
4	$\neg\neg q$	MT 1, 3
5	$p \to \neg\neg q$	→i 2–4

which verifies the validity of the sequent $\neg q \to \neg p \vdash p \to \neg\neg q$. Notice that we could apply the rule MT to formulas occurring in or above the box: at line 4, no box has been closed that would enclose line 1 or 3.

At this point it is instructive to consider the one-line argument

1	p	premise

which demonstrates $p \vdash p$. The rule →i (with conclusion $\phi \to \psi$) does not prohibit the possibility that ϕ and ψ coincide. They could both be instantiated to p. Therefore we may extend the proof above to

1	p	assumption
2	$p \to p$	→i 1 – 1

We write $\vdash p \to p$ to express that the argumentation for $p \to p$ does not depend on any premises at all.

Definition 1.10 Logical formulas ϕ with valid sequent $\vdash \phi$ are *theorems*.

Example 1.11 Here is an example of a theorem whose proof utilises most of the rules introduced so far:

1	$q \to r$	assumption
2	$\neg q \to \neg p$	assumption
3	p	assumption
4	$\neg\neg p$	¬¬i 3
5	$\neg\neg q$	MT 2, 4
6	q	¬¬e 5
7	r	→e 1, 6
8	$p \to r$	→i 3–7
9	$(\neg q \to \neg p) \to (p \to r)$	→i 2–8
10	$(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$	→i 1–9

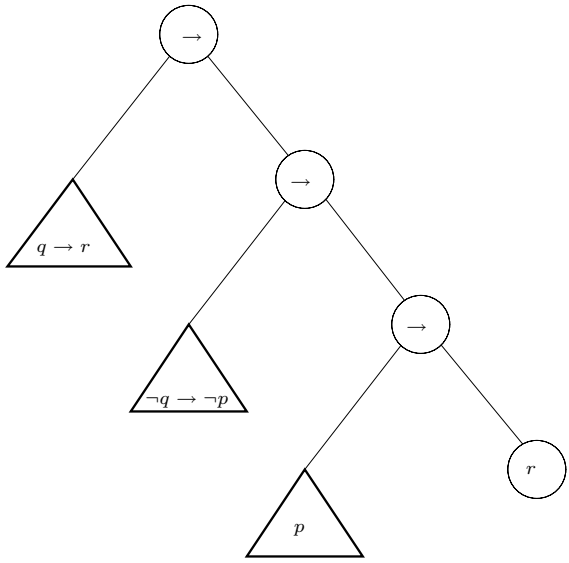


Figure 1.1. Part of the structure of the formula $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ to show how it determines the proof structure.

Therefore the sequent $\vdash (q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is valid, showing that $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ is another theorem.

Remark 1.12 Indeed, this example indicates that we may transform any proof of $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ in such a way into a proof of the theorem

$$\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\cdots \to (\phi_n \to \psi) \ldots)))$$

by 'augmenting' the previous proof with n lines of the rule $\to$i applied to ϕ_n, ϕ_{n-1},..., ϕ_1 in that order.

The nested boxes in the proof of Example 1.11 reveal a pattern of using elimination rules first, to deconstruct assumptions we have made, and then introduction rules to construct our final conclusion. More difficult proofs may involve several such phases.

Let us dwell on this important topic for a while. How did we come up with the proof above? Parts of it are *determined* by the structure of the formulas we have, while other parts require us to be *creative*. Consider the logical structure of $(q \to r) \to ((\neg q \to \neg p) \to (p \to r))$ schematically depicted in Figure 1.1. The formula is overall an implication since $\to$ is the root of the tree in Figure 1.1. But the only way to build an implication is by means

of the rule →i. Thus, we need to state the assumption of that implication as such (line 1) and have to show its conclusion (line 9). If we managed to do that, then we know how to end the proof in line 10. In fact, as we already remarked, this is the only way we could have ended it. So essentially lines 1, 9 and 10 are completely determined by the structure of the formula; further, we have reduced the problem to filling the gaps in between lines 1 and 9. But again, the formula in line 9 is an implication, so we have only one way of showing it: assuming its premise in line 2 and trying to show its conclusion in line 8; as before, line 9 is obtained by →i. The formula $p \to r$ in line 8 is yet another implication. Therefore, we have to assume p in line 3 and hope to show r in line 7, then →i produces the desired result in line 8.

The remaining question now is this: how can we show r, using the three assumptions in lines 1–3? This, and only this, is the creative part of this proof. We see the implication $q \to r$ in line 1 and know how to get r (using →e) if only we had q. So how could we get q? Well, lines 2 and 3 almost look like a pattern for the MT rule, which would give us $\neg\neg q$ in line 5; the latter is quickly changed to q in line 6 via ¬¬e. However, the pattern for MT does not match right away, since it requires $\neg\neg p$ instead of p. But this is easily accomplished via ¬¬i in line 4.

The moral of this discussion is that the logical structure of the formula to be shown tells you a lot about the structure of a possible proof and it is definitely worth your while to exploit that information in trying to prove sequents. Before ending this section on the rules for implication, let's look at some more examples (this time also involving the rules for conjunction).

Example 1.13 Using the rule ∧i, we can prove the validity of the sequent $p \wedge q \to r \vdash p \to (q \to r)$:

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	∧i 2, 3
5	r	→e 1, 4
6	$q \to r$	→i 3–5
7	$p \to (q \to r)$	→i 2–6

Example 1.14 Using the two elimination rules $\wedge e_1$ and $\wedge e_2$, we can show that the 'converse' of the sequent above is valid, too:

1	$p \to (q \to r)$	premise
2	$p \wedge q$	assumption
3	p	$\wedge e_1$ 2
4	q	$\wedge e_2$ 2
5	$q \to r$	$\to e$ 1, 3
6	r	$\to e$ 5, 4
7	$p \wedge q \to r$	$\to i$ 2–6

The validity of $p \to (q \to r) \vdash p \wedge q \to r$ and $p \wedge q \to r \vdash p \to (q \to r)$ means that these two formulas are equivalent in the sense that we can prove one from the other. We denote this by

$$p \wedge q \to r \dashv\vdash p \to (q \to r).$$

Since there can be only one formula to the right of $\vdash$, we observe that each instance of $\dashv\vdash$ can only relate *two* formulas to each other.

Example 1.15 Here is an example of a proof that uses introduction *and* elimination rules for conjunction; it shows the validity of the sequent $p \to q \vdash p \wedge r \to q \wedge r$:

1	$p \to q$	premise
2	$p \wedge r$	assumption
3	p	$\wedge e_1$ 2
4	r	$\wedge e_2$ 2
5	q	$\to e$ 1, 3
6	$q \wedge r$	$\wedge i$ 5, 4
7	$p \wedge r \to q \wedge r$	$\to i$ 2–6

The rules for disjunction The rules for disjunction are different in spirit from those for conjunction. The case for conjunction was concise and clear: proofs of $\phi \wedge \psi$ are essentially nothing but a concatenation of a proof of ϕ and a proof of ψ, plus an additional line invoking $\wedge i$. In the case of disjunctions, however, it turns out that the *introduction* of disjunctions is by far easier to grasp than their elimination. So we begin with the rules $\vee i_1$ and $\vee i_2$. From the premise ϕ we can infer that 'ϕ **or** ψ' holds, for we already know

that ϕ holds. Note that this inference is valid for any choice of ψ. By the same token, we may conclude 'ϕ **or** ψ' if we already have ψ. Similarly, that inference works for any choice of ϕ. Thus, we arrive at the proof rules

$$\frac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \frac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2.$$

So if p stands for 'Agassi won a gold medal in 1996.' and q denotes the sentence 'Agassi won Wimbledon in 1996.' then $p \vee q$ is the case because p is true, regardless of the fact that q is false. Naturally, the constructed disjunction depends upon the assumptions needed in establishing its respective disjunct p or q.

Now let's consider or-elimination. How can we use a formula of the form $\phi \vee \psi$ in a proof? Again, our guiding principle is to *disassemble* assumptions into their basic constituents so that the latter may be used in our argumentation such that they render our desired conclusion. Let us imagine that we want to show some proposition χ by assuming $\phi \vee \psi$. Since we don't know which of ϕ and ψ is true, we have to give *two* separate proofs which we need to combine into one argument:

1. First, we assume ϕ is true and have to come up with a proof of χ.
2. Next, we assume ψ is true and need to give a proof of χ as well.
3. Given these two proofs, we can infer χ from the truth of $\phi \vee \psi$, since our case analysis above is exhaustive.

Therefore, we write the rule $\vee$e as follows:

$$\frac{\phi \vee \psi \qquad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline\end{array} \qquad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline\end{array}}{\chi}\,\vee\mathrm{e}.$$

It is saying that: if $\phi \vee \psi$ is true and – no matter whether we assume ϕ or we assume ψ – we can get a proof of χ, then we are entitled to deduce χ anyway. Let's look at a proof that $p \vee q \vdash q \vee p$ is valid:

1		$p \vee q$	premise
2	┌	p	assumption
3	└	$q \vee p$	$\vee\mathrm{i}_2$ 2
4	┌	q	assumption
5	└	$q \vee p$	$\vee\mathrm{i}_1$ 4
6		$q \vee p$	$\vee$e 1, 2−3, 4−5

Here are some points you need to remember about applying the $\vee$e rule.

- For it to be a sound argument we have to make sure that the conclusions in each of the two cases (the χ in the rule) are actually the same formula.
- The work done by the rule $\vee$e is the combining of the arguments of the two cases into one.
- In each case you may not use the temporary assumption of the other case, unless it is something that has already been shown before those case boxes began.
- The invocation of rule $\vee$e in line 6 lists three things: the line in which the disjunction appears (1), and the location of the two boxes for the two cases (2–3 and 4–5).

If we use $\phi \vee \psi$ in an argument where it occurs only as an assumption or a premise, then we are missing a certain amount of information: we know ϕ, or ψ, but we don't know which one of the two it is. Thus, we have to make a solid case for each of the two possibilities ϕ or ψ; this resembles the behaviour of a `CASE` or `IF` statement found in most programming languages.

Example 1.16 Here is a more complex example illustrating these points. We prove that the sequent $q \to r \vdash p \vee q \to p \vee r$ is valid:

1	$q \to r$	premise
2	$p \vee q$	assumption
3	p	assumption
4	$p \vee r$	$\vee\text{i}_1$ 3
5	q	assumption
6	r	$\to$e 1, 5
7	$p \vee r$	$\vee\text{i}_2$ 6
8	$p \vee r$	$\vee$e 2, 3–4, 5–7
9	$p \vee q \to p \vee r$	$\to$i 2–8

Note that the propositions in lines 4, 7 and 8 coincide, so the application of $\vee$e is legitimate.

We give some more example proofs which use the rules $\vee$e, $\vee\text{i}_1$ and $\vee\text{i}_2$.

Example 1.17 Proving the validity of the sequent $(p \vee q) \vee r \vdash p \vee (q \vee r)$ is surprisingly long and seemingly complex. But this is to be expected, since

the elimination rules break $(p \vee q) \vee r$ up into its atomic constituents p, q and r, whereas the introduction rules then built up the formula $p \vee (q \vee r)$.

1	$(p \vee q) \vee r$	premise
2	$(p \vee q)$	assumption
3	p	assumption
4	$p \vee (q \vee r)$	$\vee$i$_1$ 3
5	q	assumption
6	$q \vee r$	$\vee$i$_1$ 5
7	$p \vee (q \vee r)$	$\vee$i$_2$ 6
8	$p \vee (q \vee r)$	$\vee$e $2, 3{-}4, 5{-}7$
9	r	assumption
10	$q \vee r$	$\vee$i$_2$ 9
11	$p \vee (q \vee r)$	$\vee$i$_2$ 10
12	$p \vee (q \vee r)$	$\vee$e $1, 2{-}8, 9{-}11$

Example 1.18 From boolean algebra, or circuit theory, you may know that disjunctions distribute over conjunctions. We are now able to prove this in natural deduction. The following proof:

1	$p \wedge (q \vee r)$	premise
2	p	$\wedge$e$_1$ 1
3	$q \vee r$	$\wedge$e$_2$ 1
4	q	assumption
5	$p \wedge q$	$\wedge$i $2, 4$
6	$(p \wedge q) \vee (p \wedge r)$	$\vee$i$_1$ 5
7	r	assumption
8	$p \wedge r$	$\wedge$i $2, 7$
9	$(p \wedge q) \vee (p \wedge r)$	$\vee$i$_2$ 8
10	$(p \wedge q) \vee (p \wedge r)$	$\vee$e $3, 4{-}6, 7{-}9$

verifies the validity of the sequent $p \wedge (q \vee r) \vdash (p \wedge q) \vee (p \wedge r)$ and you are encouraged to show the validity of the 'converse' $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$ yourself.

A final rule is required in order to allow us to conclude a box with a formula which has already appeared earlier in the proof. Consider the sequent $\vdash p \to (q \to p)$, whose validity may be proved as follows:

1	p	assumption
2	q	assumption
3	p	copy 1
4	$q \to p$	$\to$i 2–3
5	$p \to (q \to p)$	$\to$i 1–4

The rule 'copy' allows us to repeat something that we know already. We need to do this in this example, because the rule $\to$i requires that we end the inner box with p. The copy rule entitles us to copy formulas that appeared before, unless they depend on temporary assumptions whose box has already been closed. Though a little inelegant, this additional rule is a small price to pay for the freedom of being able to use premises, or any other 'visible' formulas, more than once.

The rules for negation We have seen the rules $\neg\neg$i and $\neg\neg$e, but we haven't seen any rules that introduce or eliminate single negations. These rules involve the notion of *contradiction*. This detour is to be expected since our reasoning is concerned about the inference, and therefore the preservation, of truth. Hence, there cannot be a direct way of inferring $\neg\phi$, given ϕ.

Definition 1.19 Contradictions are expressions of the form $\phi \wedge \neg\phi$ or $\neg\phi \wedge \phi$, where ϕ is any formula.

Examples of such contradictions are $r \wedge \neg r$, $(p \to q) \wedge \neg(p \to q)$ and $\neg(r \vee s \to q) \wedge (r \vee s \to q)$. Contradictions are a very important notion in logic. As far as truth is concerned, they are all equivalent; that means we should be able to prove the validity of

$$\neg(r \vee s \to q) \wedge (r \vee s \to q) \dashv\vdash (p \to q) \wedge \neg(p \to q) \tag{1.2}$$

since both sides are contradictions. We'll be able to prove this later, when we have introduced the rules for negation.

Indeed, it's not just that contradictions can be derived from contradictions; actually, *any* formula can be derived from a contradiction. This can be

confusing when you first encounter it; why should we endorse the argument $p \wedge \neg p \vdash q$, where

$$p\colon \text{The moon is made of green cheese.}$$
$$q\colon \text{I like pepperoni on my pizza.}$$

considering that our taste in pizza doesn't have anything to do with the constitution of the moon? On the face of it, such an endorsement may seem absurd. Nevertheless, natural deduction does have this feature that any formula can be derived from a contradiction and therefore it makes this argument valid. The reason it takes this stance is that $\vdash$ tells us all the things we may infer, provided that we can assume the formulas to the left of it. This process does not care whether such premises make any sense. This has at least the advantage that we can match $\vdash$ to checks based on semantic intuitions which we formalise later by using truth tables: if all the premises compute to 'true', then the conclusion must compute 'true' as well. In particular, this is not a constraint in the case that one of the premises is (always) false.

The fact that $\bot$ can prove anything is encoded in our calculus by the proof rule bottom-elimination:

$$\frac{\bot}{\phi}\ \bot\text{e}.$$

The fact that $\bot$ itself represents a contradiction is encoded by the proof rule not-elimination:

$$\frac{\phi \quad \neg\phi}{\bot}\ \neg\text{e}.$$

Example 1.20 We apply these rules to show that $\neg p \vee q \vdash p \to q$ is valid:

1	$\neg p \vee q$				
2	$\neg p$	premise	q	premise	
3	p	assumption	p	assumption	
4	$\bot$	$\neg$e 3, 2	q	copy 2	
5	q	$\bot$e 4	$p \to q$	$\to$i 3–4	
6	$p \to q$	$\to$i 3–5			
7	$p \to q$				$\vee$e 1, 2–6

Notice how, in this example, the proof boxes for $\vee$e are drawn side by side instead of on top of each other. It doesn't matter which way you do it.

What about introducing negations? Well, suppose we make an assumption which gets us into a contradictory state of affairs, i.e. gets us $\bot$. Then our assumption cannot be true; so it must be false. This intuition is the basis for the proof rule $\neg$i:

$$\frac{\boxed{\begin{array}{c}\phi\\ \vdots\\ \bot\end{array}}}{\neg\phi}\ \neg\text{i}.$$

Example 1.21 We put these rules in action, demonstrating that the sequent $p \to q,\ p \to \neg q \vdash \neg p$ is valid:

1	$p \to q$	premise
2	$p \to \neg q$	premise
3	p	assumption
4	q	$\to$e 1, 3
5	$\neg q$	$\to$e 2, 3
6	$\bot$	$\neg$e 4, 5
7	$\neg p$	$\neg$i 3–6

Lines 3–6 contain all the work of the $\neg$i rule. Here is a second example, showing the validity of a sequent, $p \to \neg p \vdash \neg p$, with a contradictory formula as sole premise:

1	$p \to \neg p$	premise
2	p	assumption
3	$\neg p$	$\to$e 1, 2
4	$\bot$	$\neg$e 2, 3
5	$\neg p$	$\neg$i 2–4

Example 1.22 We prove that the sequent $p \to (q \to r),\ p,\ \neg r \vdash \neg q$ is valid,

without using the MT rule:

1	$p \to (q \to r)$	premise
2	p	premise
3	$\neg r$	premise
4	q	assumption
5	$q \to r$	$\to$e 1, 2
6	r	$\to$e 5, 4
7	$\bot$	$\neg$e 6, 3
8	$\neg q$	$\neg$i 4–7

Example 1.23 Finally, we return to the argument of Examples 1.1 and 1.2, which can be coded up by the sequent $p \wedge \neg q \to r$, $\neg r$, $p \vdash q$ whose validity we now prove:

1	$p \wedge \neg q \to r$	premise
2	$\neg r$	premise
3	p	premise
4	$\neg q$	assumption
5	$p \wedge \neg q$	$\wedge$i 3, 4
6	r	$\to$e 1, 5
7	$\bot$	$\neg$e 6, 2
8	$\neg\neg q$	$\neg$i 4–7
9	q	$\neg\neg$e 8

1.2.2 Derived rules

When describing the proof rule *modus tollens* (MT), we mentioned that it is not a primitive rule of natural deduction, but can be derived from some of the other rules. Here is the derivation of

$$\frac{\phi \to \psi \quad \neg\psi}{\neg\phi}\ \text{MT}$$

from $\rightarrow$e, $\neg$e and $\neg$i:

1	$\phi \rightarrow \psi$	premise
2	$\neg\psi$	premise
3	ϕ	assumption
4	ψ	$\rightarrow$e 1, 3
5	$\bot$	$\neg$e 4, 2
6	$\neg\phi$	$\neg$i 3–5

We could now go back through the proofs in this chapter and replace applications of MT by this combination of $\rightarrow$e, $\neg$e and $\neg$i. However, it is convenient to think of MT as a shorthand (or a macro).

The same holds for the rule

$$\frac{\phi}{\neg\neg\phi}\ \neg\neg\text{i}.$$

It can be derived from the rules $\neg$i and $\neg$e, as follows:

1	ϕ	premise
2	$\neg\phi$	assumption
3	$\bot$	$\neg$e 1, 2
4	$\neg\neg\phi$	$\neg$i 2–3

There are (unboundedly) many such derived rules which we could write down. However, there is no point in making our calculus fat and unwieldy; and some purists would say that we should stick to a minimum set of rules, all of which are independent of each other. We don't take such a purist view. Indeed, the two derived rules we now introduce are extremely useful. You will find that they crop up frequently when doing exercises in natural deduction, so it is worth giving them names as derived rules. In the case of the second one, its derivation from the primitive proof rules is not very obvious.

The first one has the Latin name *reductio ad absurdum*. It means 'reduction to absurdity' and we will simply call it *proof by contradiction* (PBC for short). The rule says: if from $\neg\phi$ we obtain a contradiction, then we are entitled to deduce ϕ:

$$\frac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline\end{array}}{\phi}\ \text{PBC}.$$

This rule looks rather similar to ¬i, except that the negation is in a different place. This is the clue to how to derive PBC from our basic proof rules. Suppose we have a proof of $\bot$ from $\neg\phi$. By $\to$i, we can transform this into a proof of $\neg\phi \to \bot$ and proceed as follows:

1	$\neg\phi \to \bot$	given
2	$\neg\phi$	assumption
3	$\bot$	$\to$e 1, 2
4	$\neg\neg\phi$	¬i 2–3
5	ϕ	¬¬e 4

This shows that PBC can be derived from $\to$i, ¬i, $\to$e and ¬¬e.

The final derived rule we consider in this section is arguably the most useful to use in proofs, because its derivation is rather long and complicated, so its usage often saves time and effort. It also has a Latin name, *tertium non datur*; the English name is the law of the excluded middle, or LEM for short. It simply says that $\phi \vee \neg\phi$ is true: whatever ϕ is, it must be either true or false; in the latter case, $\neg\phi$ is true. There is no third possibility (hence *excluded middle*): the sequent $\vdash \phi \vee \neg\phi$ is valid. Its validity is implicit, for example, whenever you write an if-statement in a programming language: '`if` B $\{C_1\}$ `else` $\{C_2\}$' relies on the fact that $B \vee \neg B$ is always true (and that B and $\neg B$ can never be true at the same time). Here is a proof in natural deduction that derives the law of the excluded middle from basic proof rules:

1	$\neg(\phi \vee \neg\phi)$	assumption
2	ϕ	assumption
3	$\phi \vee \neg\phi$	$\vee i_1$ 2
4	$\bot$	¬e 3, 1
5	$\neg\phi$	¬i 2–4
6	$\phi \vee \neg\phi$	$\vee i_2$ 5
7	$\bot$	¬e 6, 1
8	$\neg\neg(\phi \vee \neg\phi)$	¬i 1–7
9	$\phi \vee \neg\phi$	¬¬e 8

Example 1.24 Using LEM, we show that $p \to q \vdash \neg p \vee q$ is valid:

1	$p \to q$	premise
2	$\neg p \vee p$	LEM
3	$\neg p$	assumption
4	$\neg p \vee q$	$\vee\text{i}_1$ 3
5	p	assumption
6	q	$\to$e 1, 5
7	$\neg p \vee q$	$\vee\text{i}_2$ 6
8	$\neg p \vee q$	$\vee$e 2, 3−4, 5−7

It can be difficult to decide which instance of LEM would benefit the progress of a proof. Can you re-do the example above with $q \vee \neg q$ as LEM?

1.2.3 Natural deduction in summary

The proof rules for natural deduction are summarised in Figure 1.2. The explanation of the rules we have given so far in this chapter is *declarative*; we have presented each rule and justified it in terms of our intuition about the logical connectives. However, when you try to use the rules yourself, you'll find yourself looking for a more *procedural* interpretation; what does a rule do and how do you use it? For example,

- $\wedge$i says: to prove $\phi \wedge \psi$, you must first prove ϕ and ψ separately and then use the rule $\wedge$i.
- $\wedge\text{e}_1$ says: to prove ϕ, try proving $\phi \wedge \psi$ and then use the rule $\wedge\text{e}_1$. Actually, this doesn't sound like very good advice because probably proving $\phi \wedge \psi$ will be harder than proving ϕ alone. However, you might find that you *already have* $\phi \wedge \psi$ lying around, so that's when this rule is useful. Compare this with the example sequent in Example 1.15.
- $\vee\text{i}_1$ says: to prove $\phi \vee \psi$, try proving ϕ. Again, in general it is harder to prove ϕ than it is to prove $\phi \vee \psi$, so this will usually be useful only if you've already managed to prove ϕ. For example, if you want to prove $q \vdash p \vee q$, you certainly won't be able simply to use the rule $\vee\text{i}_1$, but $\vee\text{i}_2$ will work.
- $\vee$e has an excellent procedural interpretation. It says: if you have $\phi \vee \psi$, and you want to prove some χ, then try to prove χ from ϕ and from ψ in turn. (In those subproofs, of course you can use the other prevailing premises as well.)
- Similarly, $\to$i says, if you want to prove $\phi \to \psi$, try proving ψ from ϕ (and the other prevailing premises).
- $\neg$i says: to prove $\neg\phi$, prove $\bot$ from ϕ (and the other prevailing premises).

The basic rules of natural deduction:

	introduction	*elimination*
$\wedge$	$\dfrac{\phi \quad \psi}{\phi \wedge \psi}\,\wedge\mathrm{i}$	$\dfrac{\phi \wedge \psi}{\phi}\,\wedge\mathrm{e}_1 \qquad \dfrac{\phi \wedge \psi}{\psi}\,\wedge\mathrm{e}_2$
$\vee$	$\dfrac{\phi}{\phi \vee \psi}\,\vee\mathrm{i}_1 \qquad \dfrac{\psi}{\phi \vee \psi}\,\vee\mathrm{i}_2$	$\dfrac{\phi \vee \psi \quad \begin{array}{|c|}\hline \phi \\ \vdots \\ \chi \\ \hline\end{array} \quad \begin{array}{|c|}\hline \psi \\ \vdots \\ \chi \\ \hline\end{array}}{\chi}\,\vee\mathrm{e}$
$\rightarrow$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \psi \\ \hline\end{array}}{\phi \rightarrow \psi}\,\rightarrow\mathrm{i}$	$\dfrac{\phi \quad \phi \rightarrow \psi}{\psi}\,\rightarrow\mathrm{e}$
$\neg$	$\dfrac{\begin{array}{|c|}\hline \phi \\ \vdots \\ \bot \\ \hline\end{array}}{\neg\phi}\,\neg\mathrm{i}$	$\dfrac{\phi \quad \neg\phi}{\bot}\,\neg\mathrm{e}$
$\bot$	(no introduction rule for $\bot$)	$\dfrac{\bot}{\phi}\,\bot\mathrm{e}$
$\neg\neg$		$\dfrac{\neg\neg\phi}{\phi}\,\neg\neg\mathrm{e}$

Some useful derived rules:

$$\frac{\phi \rightarrow \psi \quad \neg\psi}{\neg\phi}\,\mathrm{MT} \qquad \frac{\phi}{\neg\neg\phi}\,\neg\neg\mathrm{i}$$

$$\frac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline\end{array}}{\phi}\,\mathrm{PBC} \qquad \frac{}{\phi \vee \neg\phi}\,\mathrm{LEM}$$

Figure 1.2. Natural deduction rules for propositional logic.

At any stage of a proof, it is permitted to introduce any formula as assumption, by choosing a proof rule that opens a box. As we saw, natural deduction employs boxes to control the scope of assumptions. When an assumption is introduced, a box is opened. Discharging assumptions is achieved by closing a box according to the pattern of its particular proof rule. It's useful to make assumptions by opening boxes. *But don't forget you have to close them in the manner prescribed by their proof rule.*

OK, but how do we actually go about constructing a proof? Given a sequent, you write its premises at the top of your page and its conclusion at the bottom. Now, you're trying to fill in the gap, which involves working simultaneously on the premises (to bring them towards the conclusion) and on the conclusion (to massage it towards the premises).

Look first at the conclusion. If it is of the form $\phi \to \psi$, then apply[6] the rule $\to$i. This means drawing a box with ϕ at the top and ψ at the bottom. So your proof, which started out like this:

$$\begin{array}{c} \vdots \\ \text{premises} \\ \vdots \\ \phi \to \psi \end{array}$$

now looks like this:

$$\begin{array}{l} \vdots \\ \text{premises} \\ \vdots \\ \boxed{\begin{array}{ll} \phi & \text{assumption} \\ \\ \\ \psi & \end{array}} \\ \phi \to \psi \qquad \to\text{i} \end{array}$$

You still have to find a way of filling in the gap between the ϕ and the ψ. But you now have an extra formula to work with and you have simplified the conclusion you are trying to reach.

[6] Except in situations such as $p \to (q \to \neg r), p \vdash q \to \neg r$ where $\to$e produces a simpler proof.

The proof rule ¬i is very similar to →i and has the same beneficial effect on your proof attempt. It gives you an extra premise to work with and simplifies your conclusion.

At any stage of a proof, several rules are likely to be applicable. Before applying any of them, list the applicable ones and think about which one is likely to improve the situation for your proof. You'll find that →i and ¬i most often improve it, so always use them whenever you can. There is no easy recipe for when to use the other rules; often you have to make judicious choices.

1.2.4 Provable equivalence

Definition 1.25 Let ϕ and ψ be formulas of propositional logic. We say that ϕ and ψ are *provably equivalent* iff (we write 'iff' for 'if, and only if' in the sequel) the sequents $\phi \vdash \psi$ and $\psi \vdash \phi$ are valid; that is, there is a proof of ψ from ϕ and another one going the other way around. As seen earlier, we denote that ϕ and ψ are provably equivalent by $\phi \dashv\vdash \psi$.

Note that, by Remark 1.12, we could just as well have defined $\phi \dashv\vdash \psi$ to mean that the sequent $\vdash (\phi \to \psi) \wedge (\psi \to \phi)$ is valid; it defines the same concept. Examples of provably equivalent formulas are

$$\begin{array}{ll} \neg(p \wedge q) \dashv\vdash \neg q \vee \neg p & \neg(p \vee q) \dashv\vdash \neg q \wedge \neg p \\ p \to q \dashv\vdash \neg q \to \neg p & p \to q \dashv\vdash \neg p \vee q \\ p \wedge q \to p \dashv\vdash r \vee \neg r & p \wedge q \to r \dashv\vdash p \to (q \to r). \end{array}$$

The reader should prove all of these six equivalences in natural deduction.

1.2.5 An aside: proof by contradiction

Sometimes we can't prove something *directly* in the sense of taking apart given assumptions and reasoning with their constituents in a constructive way. Indeed, the proof system of natural deduction, summarised in Figure 1.2, specifically allows for *indirect* proofs that lack a constructive quality: for example, the rule

$$\dfrac{\begin{array}{|c|}\hline \neg\phi \\ \vdots \\ \bot \\ \hline \end{array}}{\phi}\ \text{PBC}$$

allows us to prove ϕ by showing that $\neg\phi$ leads to a contradiction. Although 'classical logicians' argue that this is valid, logicians of another kind, called 'intuitionistic logicians,' argue that to prove ϕ you should do it directly, rather than by arguing merely that $\neg\phi$ is impossible. The two other rules on which classical and intuitionistic logicians disagree are

$$\frac{}{\phi \vee \neg\phi}\ \text{LEM} \qquad \frac{\neg\neg\phi}{\phi}\ \neg\neg\text{e}.$$

Intuitionistic logicians argue that, to show $\phi \vee \neg\phi$, you have to show ϕ, or $\neg\phi$. If neither of these can be shown, then the putative truth of the disjunction has no justification. Intuitionists reject $\neg\neg$e since we have already used this rule to prove LEM and PBC from rules which the intuitionists do accept. In the exercises, you are asked to show why the intuitionists also reject PBC.

Let us look at a proof that shows up this difference, involving real numbers. Real numbers are floating point numbers like 23.54721, only some of them might actually be infinitely long such as 23.138592748500123950734..., with no periodic behaviour after the decimal point.

Given a positive real number a and a *natural* (whole) number b, we can calculate a^b: it is just a times itself, b times, so $2^2 = 2 \cdot 2 = 4$, $2^3 = 2 \cdot 2 \cdot 2 = 8$ and so on. When b is a *real* number, we can also define a^b, as follows. We say that $a^0 \stackrel{\text{def}}{=} 1$ and, for a non-zero rational number k/n, where $n \neq 0$, we let $a^{k/n} \stackrel{\text{def}}{=} \sqrt[n]{a^k}$ where $\sqrt[n]{x}$ is the real number y such that $y^n = x$. From real analysis one knows that any real number b can be approximated by a sequence of rational numbers k_0/n_0, k_1/n_1, ... Then we define a^b to be the real number approximated by the sequence a^{k_0/n_0}, a^{k_1/n_1}, ... (In calculus, one can show that this 'limit' a^b is unique and independent of the choice of approximating sequence.) Also, one calls a real number *irrational* if it can't be written in the form k/n for some integers k and $n \neq 0$. In the exercises you will be asked to find a semi-formal proof showing that $\sqrt{2}$ is irrational.

We now present a proof of a fact about real numbers in the informal style used by mathematicians (this proof can be formalised as a natural deduction proof in the logic presented in Chapter 2). The fact we prove is:

Theorem 1.26 *There exist irrational numbers a and b such that a^b is rational.*

PROOF: We choose b to be $\sqrt{2}$ and proceed by a case analysis. Either b^b is irrational, or it is not. (Thus, our proof uses $\vee$e on an instance of LEM.)

(i) Assume that b^b is rational. Then this proof is easy since we can choose irrational numbers a and b to be $\sqrt{2}$ and see that a^b is just b^b which was assumed to be rational.

(ii) Assume that b^b is *ir*rational. Then we change our strategy slightly and choose a to be $\sqrt{2}^{\sqrt{2}}$. Clearly, a is irrational by the assumption of case (ii). But we know that b is irrational (this was known by the ancient Greeks; see the proof outline in the exercises). So a and b are both irrational numbers and

$$a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = \sqrt{2}^{(\sqrt{2}\cdot\sqrt{2})} = (\sqrt{2})^2 = 2$$

is rational, where we used the law $(x^y)^z = x^{(y\cdot z)}$.

Since the two cases above are exhaustive (*either* b^b is irrational, *or* it isn't) we have proven the theorem. □

This proof is perfectly legitimate and mathematicians use arguments like that all the time. The exhaustive nature of the case analysis above rests on the use of the rule LEM, which we use to prove that either b is rational or it is not. Yet, there is something puzzling about it. Surely, we have secured the fact that there are irrational numbers a and b such that a^b is rational, but are we in a position to specify an actual pair of such numbers satisfying this theorem? More precisely, which of the pairs (a, b) above fulfils the assertion of the theorem, the pair $(\sqrt{2}, \sqrt{2})$, or the pair $(\sqrt{2}^{\sqrt{2}}, \sqrt{2})$? Our proof tells us nothing about *which* of them is the right choice; it just says that at least one of them works.

Thus, the intuitionists favour a calculus containing the introduction and elimination rules shown in Figure 1.2 and excluding the rule ¬¬e and the derived rules. Intuitionistic logic turns out to have some specialised applications in computer science, such as modelling type-inference systems used in compilers or the staged execution of program code; but in this text we stick to the full so-called classical logic which includes all the rules.

1.3 Propositional logic as a formal language

In the previous section we learned about propositional atoms and how they can be used to build more complex logical formulas. We were deliberately informal about that, for our main focus was on trying to understand the precise mechanics of the natural deduction rules. However, it should have been clear that the rules we stated are valid for *any* formulas we can form, as long as they match the pattern required by the respective rule. For example,

the application of the proof rule →e in

1	$p \to q$	premise
2	p	premise
3	q	→e 1, 2

is equally valid if we substitute p with $p \vee \neg r$ and q with $r \to p$:

1	$p \vee \neg r \to (r \to p)$	premise
2	$p \vee \neg r$	premise
3	$r \to p$	→e 1, 2

This is why we expressed such rules as schemes with Greek symbols standing for generic formulas. Yet, it is time that we make precise the notion of 'any formula we may form.' Because this text concerns various logics, we will introduce in (1.3) an easy formalism for specifying well-formed formulas. In general, we need an *unbounded* supply of propositional atoms $p, q, r, \ldots$, or $p_1, p_2, p_3, \ldots$ You should not be too worried about the need for infinitely many such symbols. Although we may only need *finitely many* of these propositions to describe a property of a computer program successfully, we cannot specify how many such atomic propositions we will need in any concrete situation, so having infinitely many symbols at our disposal is a cheap way out. This can be compared with the potentially infinite nature of English: the number of grammatically correct English sentences is infinite, but finitely many such sentences will do in whatever situation you might be in (writing a book, attending a lecture, listening to the radio, having a dinner date, ...).

Formulas in our propositional logic should certainly be strings over the alphabet $\{p, q, r, \ldots\} \cup \{p_1, p_2, p_3, \ldots\} \cup \{\neg, \wedge, \vee, \to, (,)\}$. This is a trivial observation and as such is not good enough for what we are trying to capture. For example, the string $(\neg)() \vee pq \to$ is a word over that alphabet, yet, it does not seem to make a lot of sense as far as propositional logic is concerned. So what we have to define are those strings which we want to call formulas. We call such formulas *well-formed*.

Definition 1.27 The well-formed formulas of propositional logic are those which we obtain by using the construction rules below, and only those, finitely many times:

atom: Every propositional atom $p, q, r, \ldots$ and $p_1, p_2, p_3, \ldots$ is a well-formed formula.
¬: If ϕ is a well-formed formula, then so is $(\neg\phi)$.
∧: If ϕ and ψ are well-formed formulas, then so is $(\phi \wedge \psi)$.
∨: If ϕ and ψ are well-formed formulas, then so is $(\phi \vee \psi)$.
→: If ϕ and ψ are well-formed formulas, then so is $(\phi \to \psi)$.

It is most crucial to realize that this definition is the one a computer would expect and that we did not make use of the binding priorities agreed upon in the previous section.

Convention. In this section we act as if we are a rigorous computer and we call formulas well-formed iff they can be deduced to be so using the definition above.

Further, note that the condition 'and only those' in the definition above rules out the possibility of any other means of establishing that formulas are well-formed. Inductive definitions, like the one of well-formed propositional logic formulas above, are so frequent that they are often given by a defining grammar in Backus Naur form (BNF). In that form, the above definition reads more compactly as

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \mid (\phi \vee \phi) \mid (\phi \to \phi) \tag{1.3}$$

where p stands for any atomic proposition and each occurrence of ϕ to the right of $::=$ stands for any already constructed formula.

So how can we show that a string is a well-formed formula? For example, how do we answer this for ϕ being

$$(((\neg p) \wedge q) \to (p \wedge (q \vee (\neg r)))) \ ? \tag{1.4}$$

Such reasoning is greatly facilitated by the fact that the grammar in (1.3) satisfies the *inversion principle*, which means that we can invert the process of building formulas: although the grammar rules allow for five different ways of constructing more complex formulas – the five clauses in (1.3) – there is always a unique clause which was used last. For the formula above, this last operation was an application of the fifth clause, for ϕ is an implication with the assumption $((\neg p) \wedge q)$ and conclusion $(p \wedge (q \vee (\neg r)))$. By applying the inversion principle to the assumption, we see that it is a conjunction of $(\neg p)$ and q. The former has been constructed using the second clause and is well-formed since p is well-formed by the first clause in (1.3). The latter is well-formed for the same reason. Similarly, we can apply the inversion

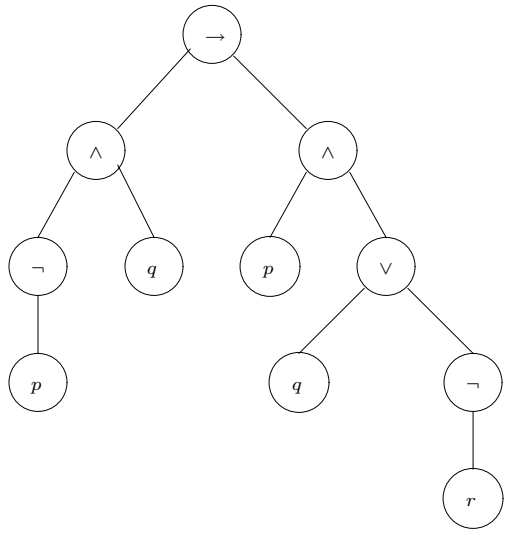


Figure 1.3. A parse tree representing a well-formed formula.

principle to the conclusion $(p \wedge (q \vee (\neg r)))$, inferring that it is indeed well-formed. In summary, the formula in (1.4) is well-formed.

For us humans, dealing with brackets is a tedious task. The reason we need them is that formulas really have a tree-like structure, although we prefer to represent them in a linear way. In Figure 1.3 you can see the parse tree[7] of the well-formed formula ϕ in (1.4). Note how brackets become unnecessary in this parse tree since the paths and the branching structure of this tree remove any possible ambiguity in interpreting ϕ. In representing ϕ as a linear string, the branching structure of the tree is retained by the insertion of brackets as done in the definition of well-formed formulas.

So how would you go about showing that a string of symbols ψ is *not* well-formed? At first sight, this is a bit trickier since we somehow have to make sure that ψ could not have been obtained by *any* sequence of construction rules. Let us look at the formula $(\neg)() \vee pq \rightarrow$ from above. We can decide this matter by being very observant. The string $(\neg)() \vee pq \rightarrow$ contains $\neg)$ and $\neg$ cannot be the rightmost symbol of a well-formed formula (check all the rules to verify this claim!); but the only time we can put a ')' to the right of something is if that something is a well-formed formula (again, check all the rules to see that this is so). Thus, $(\neg)() \vee pq \rightarrow$ is *not* well-formed.

Probably the easiest way to verify whether some formula ϕ is well-formed is by trying to draw its parse tree. In this way, you can verify that the

[7] We will use this name without explaining it any further and are confident that you will understand its meaning through the examples.

formula in (1.4) is well-formed. In Figure 1.3 we see that its parse tree has $\rightarrow$ as its root, expressing that the formula is, at its top level, an implication. Using the grammar clause for implication, it suffices to show that the left and right subtrees of this root node are well-formed. That is, we proceed in a top-down fashion and, in this case, successfully. Note that the parse trees of well-formed formulas have either an atom as root (and then this is all there is in the tree), or the root contains $\neg$, $\vee$, $\wedge$ or $\rightarrow$. In the case of $\neg$ there is only *one* subtree coming out of the root. In the cases $\wedge$, $\vee$ or $\rightarrow$ we must have *two* subtrees, each of which must behave as just described; this is another example of an *inductive* definition.

Thinking in terms of trees will help you understand standard notions in logic, for example, the concept of a *subformula*. Given the well-formed formula ϕ above, its subformulas are just the ones that correspond to the subtrees of its parse tree in Figure 1.3. So we can list all its leaves p, q (occurring twice), and r, then $(\neg p)$ and $((\neg p) \wedge q)$ on the left subtree of $\rightarrow$ and $(\neg r)$, $(q \vee (\neg r))$ and $((p \wedge (q \vee (\neg p))))$ on the right subtree of $\rightarrow$. The whole tree is a subtree of itself as well. So we can list all nine subformulas of ϕ as

$$
\begin{array}{l}
p \\
q \\
r \\
(\neg p) \\
((\neg p) \wedge q) \\
(\neg r) \\
(q \vee (\neg r)) \\
((p \wedge (q \vee (\neg r)))) \\
(((\neg p) \wedge q) \rightarrow (p \wedge (q \vee (\neg r)))).
\end{array}
$$

Let us consider the tree in Figure 1.4. Why does it represent a well-formed formula? All its leaves are propositional atoms (p twice, q and r), all branching nodes are logical connectives ($\neg$ twice, $\wedge$, $\vee$ and $\rightarrow$) and the numbers of subtrees are correct in all those cases (one subtree for a $\neg$ node and two subtrees for all other non-leaf nodes). How do we obtain the linear representation of this formula? If we ignore brackets, then we are seeking nothing but the *in-order* representation of this tree as a list[8]. The resulting well-formed formula is $((\neg(p \vee (q \rightarrow (\neg p)))) \wedge r)$.

[8] The other common ways of flattening trees to lists are *preordering* and *postordering*. See any text on binary trees as data structures for further details.

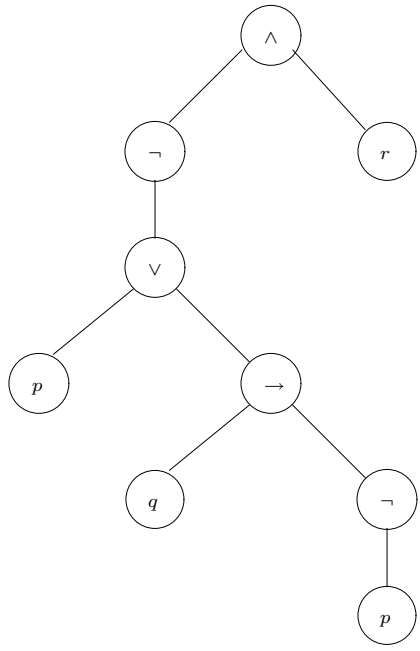


Figure 1.4. Given: a tree; wanted: its linear representation as a logical formula.

The tree in Figure 1.21 on page 82, however, does *not* represent a well-formed formula for two reasons. First, the leaf $\wedge$ (and a similar argument applies to the leaf $\neg$), the left subtree of the node $\rightarrow$, is not a propositional atom. This could be fixed by saying that we decided to leave the left and right subtree of that node unspecified and that we are willing to provide those now. However, the second reason is fatal. The p node is not a leaf since it has a subtree, the node $\neg$. This cannot make sense if we think of the entire tree as some logical formula. So this tree does not represent a well-formed logical formula.

1.4 Semantics of propositional logic

1.4.1 The meaning of logical connectives

In the second section of this chapter, we developed a calculus of reasoning which could verify that sequents of the form $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ are valid, which means: from the premises ϕ_1, ϕ_2, …, ϕ_n, we may conclude ψ.

In this section we give another account of this relationship between the premises ϕ_1, ϕ_2, …, ϕ_n and the conclusion ψ. To contrast with the sequent

above, we define a new relationship, written

$$\phi_1, \phi_2, \ldots, \phi_n \vDash \psi.$$

This account is based on looking at the 'truth values' of the atomic formulas in the premises and the conclusion; and at how the logical connectives manipulate these truth values. What is the truth value of a declarative sentence, like sentence (3) 'Every even natural number > 2 is the sum of two prime numbers'? Well, declarative sentences express a fact about the real world, the physical world we live in, or more abstract ones such as computer models, or our thoughts and feelings. Such factual statements either match reality (they are *true*), or they don't (they are *false*).

If we combine declarative sentences p and q with a logical connective, say $\wedge$, then the truth value of $p \wedge q$ is determined by three things: the truth value of p, the truth value of q and the meaning of $\wedge$. The meaning of $\wedge$ is captured by the observation that $p \wedge q$ is true iff p *and* q are both true; otherwise $p \wedge q$ is false. Thus, as far as $\wedge$ is concerned, it needs only to know whether p and q are true, it does *not* need to know what p and q are actually saying about the world out there. This is also the case for all the other logical connectives and is the reason why we can compute the truth value of a formula just by knowing the truth values of the atomic propositions occurring in it.

Definition 1.28 1. The set of truth values contains two elements `T` and `F`, where `T` represents 'true' and `F` represents 'false'.
2. A *valuation* or *model* of a formula ϕ is an assignment of each propositional atom in ϕ to a truth value.

Example 1.29 The map which assigns `T` to q and `F` to p is a valuation for $p \vee \neg q$. Please list the remaining three valuations for this formula.

We can think of the meaning of $\wedge$ as a function of two arguments; each argument is a truth value and the result is again such a truth value. We specify this function in a table, called the *truth table for conjunction*, which you can see in Figure 1.5. In the first column, labelled ϕ, we list all possible

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

Figure 1.5. The truth table for conjunction, the logical connective $\wedge$.

ϕ	ψ	$\phi \wedge \psi$
T	T	T
T	F	F
F	T	F
F	F	F

ϕ	ψ	$\phi \vee \psi$
T	T	T
T	F	T
F	T	T
F	F	F

ϕ	ψ	$\phi \to \psi$
T	T	T
T	F	F
F	T	T
F	F	T

ϕ	$\neg\phi$
T	F
F	T

$\top$
T

$\bot$
F

Figure 1.6. The truth tables for all the logical connectives discussed so far.

truth values of ϕ. Actually we list them *twice* since we also have to deal with another formula ψ, so the possible number of combinations of truth values for ϕ and ψ equals $2 \cdot 2 = 4$. Notice that the four pairs of ϕ and ψ values in the first two columns really exhaust all those possibilities (TT, TF, FT and FF). In the third column, we list the result of $\phi \wedge \psi$ according to the truth values of ϕ and ψ. So in the first line, where ϕ and ψ have value T, the result is T again. In all other lines, the result is F since at least one of the propositions ϕ or ψ has value F.

In Figure 1.6 you find the truth tables for all logical connectives of propositional logic. Note that $\neg$ turns T into F and vice versa. Disjunction is the mirror image of conjunction if we swap T and F, namely, a disjunction returns F iff both arguments are equal to F, otherwise (= at least one of the arguments equals T) it returns T. The behaviour of implication is not quite as intuitive. Think of the meaning of $\to$ as checking whether *truth is being preserved*. Clearly, this is not the case when we have $\mathtt{T} \to \mathtt{F}$, since we infer something that is false from something that is true. So the second entry in the column $\phi \to \psi$ equals F. On the other hand, $\mathtt{T} \to \mathtt{T}$ obviously preserves truth, but so do the cases $\mathtt{F} \to \mathtt{T}$ and $\mathtt{F} \to \mathtt{F}$, because there is no truth to be preserved in the first place as the assumption of the implication is false.

If you feel slightly uncomfortable with the semantics (= the meaning) of $\to$, then it might be good to think of $\phi \to \psi$ as an abbreviation of the formula $\neg\phi \vee \psi$ *as far as meaning is concerned*; these two formulas are very different syntactically and natural deduction treats them differently as well. But using the truth tables for $\neg$ and $\vee$ you can check that $\phi \to \psi$ evaluates

to T iff $\neg\phi \vee \psi$ does so. This means that $\phi \to \psi$ and $\neg\phi \vee \psi$ are *semantically equivalent*; more on that in Section 1.5.

Given a formula ϕ which contains the propositional atoms $p_1, p_2, \ldots, p_n$, we can construct a truth table for ϕ, at least in principle. The caveat is that this truth table has 2^n many lines, each line listing a possible combination of truth values for $p_1, p_2, \ldots, p_n$; and for large n this task is impossible to complete. Our aim is thus to compute the value of ϕ for each of these 2^n cases for moderately small values of n. Let us consider the example ϕ in Figure 1.3. It involves three propositional atoms ($n = 3$) so we have $2^3 = 8$ cases to consider.

We illustrate how things go for one particular case, namely for the valuation in which q evaluates to F; and p and r evaluate to T. What does $\neg p \wedge q \to p \wedge (q \vee \neg r)$ evaluate to? Well, the beauty of our semantics is that it is *compositional*. If we know the meaning of the subformulas $\neg p \wedge q$ and $p \wedge (q \vee \neg r)$, then we just have to look up the appropriate line of the $\to$ truth table to find the value of ϕ, for ϕ is an implication of these two subformulas. Therefore, we can do the calculation by traversing the parse tree of ϕ in a bottom-up fashion. We know what its leaves evaluate to since we stated what the atoms p, q and r evaluated to. Because the meaning of p is T, we see that $\neg p$ computes to F. Now q is assumed to represent F and the conjunction of F and F is F. Thus, the left subtree of the node $\to$ evaluates to F. As for the right subtree of $\to$, r stands for T so $\neg r$ computes to F and q means F, so the disjunction of F and F is still F. We have to take that result, F, and compute its conjunction with the meaning of p which is T. Since the conjunction of T and F is F, we get F as the meaning of the right subtree of $\to$. Finally, to evaluate the meaning of ϕ, we compute $\mathtt{F} \to \mathtt{F}$ which is T. Figure 1.7 shows how the truth values propagate upwards to reach the root whose associated truth value is the truth value of ϕ given the meanings of p, q and r above.

It should now be quite clear how to build a truth table for more complex formulas. Figure 1.8 contains a truth table for the formula $(p \to \neg q) \to (q \vee \neg p)$. To be more precise, the first two columns list all possible combinations of values for p and q. The next two columns compute the corresponding values for $\neg p$ and $\neg q$. Using these four columns, we may compute the column for $p \to \neg q$ and $q \vee \neg p$. To do so we think of the first and fourth columns as the data for the $\to$ truth table and compute the column of $p \to \neg q$ accordingly. For example, in the first line p is T and $\neg q$ is F so the entry for $p \to \neg q$ is $\mathtt{T} \to \mathtt{F} = \mathtt{F}$ by definition of the meaning of $\to$. In this fashion, we can fill out the rest of the fifth column. Column 6 works similarly, only we now need to look up the truth table for $\vee$ with columns 2 and 3 as input.

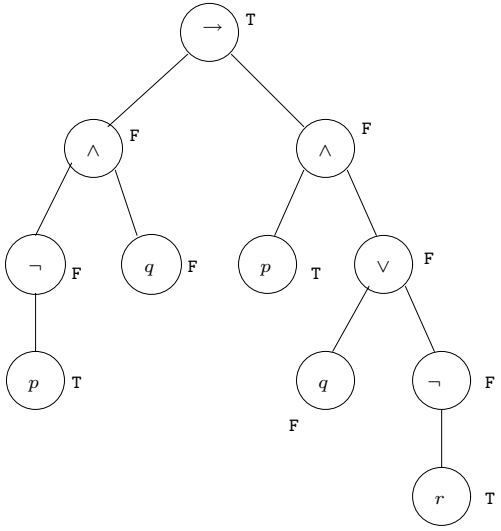


Figure 1.7. The evaluation of a logical formula under a given valuation.

p	q	$\neg p$	$\neg q$	$p \to \neg q$	$q \vee \neg p$	$(p \to \neg q) \to (q \vee \neg p)$
T	T	F	F	F	T	T
T	F	F	T	T	F	F
F	T	T	F	T	T	T
F	F	T	T	T	T	T

Figure 1.8. An example of a truth table for a more complex logical formula.

Finally, column 7 results from applying the truth table of $\to$ to columns 5 and 6.

1.4.2 Mathematical induction

Here is a little anecdote about the German mathematician Gauss who, as a pupil at age 8, did not pay attention in class (can you imagine?), with the result that his teacher made him sum up all natural numbers from 1 to 100. The story has it that Gauss came up with the correct answer 5050 within seconds, which infuriated his teacher. How did Gauss do it? Well, possibly he knew that

$$1 + 2 + 3 + 4 + \cdots + n = \frac{n \cdot (n+1)}{2} \tag{1.5}$$

for all natural numbers n.[9] Thus, taking $n = 100$, Gauss could easily calculate:

$$1 + 2 + 3 + 4 + \cdots + 100 = \frac{100 \cdot 101}{2} = 5050.$$

Mathematical induction allows us to prove equations, such as the one in (1.5), for arbitrary n. More generally, it allows us to show that *every* natural number satisfies a certain property. Suppose we have a property M which we think is true of all natural numbers. We write $M(5)$ to say that the property is true of 5, etc. Suppose that we know the following two things about the property M:

1. **Base case:** The natural number 1 has property M, i.e. we have a proof of $M(1)$.
2. **Inductive step:** If n is a natural number which *we assume* to have property $M(n)$, then *we can show* that $n + 1$ has property $M(n + 1)$; i.e. we have a proof of $M(n) \to M(n + 1)$.

Definition 1.30 The principle of mathematical induction says that, on the grounds of these two pieces of information above, every natural number n has property $M(n)$. The assumption of $M(n)$ in the inductive step is called the *induction hypothesis*.

Why does this principle make sense? Well, take *any* natural number k. If k equals 1, then k has property $M(1)$ using the base case and so we are done. Otherwise, we can use the inductive step, applied to $n = 1$, to infer that $2 = 1 + 1$ has property $M(2)$. We can do that using $\to$e, for we know that 1 has the property in question. Now we use that same inductive step on $n = 2$ to infer that 3 has property $M(3)$ and we repeat this until we reach $n = k$ (see Figure 1.9). Therefore, we should have no objections about using the principle of mathematical induction for natural numbers.

Returning to Gauss' example we claim that the sum $1 + 2 + 3 + 4 + \cdots + n$ equals $n \cdot (n + 1)/2$ for all natural numbers n.

Theorem 1.31 *The sum* $1 + 2 + 3 + 4 + \cdots + n$ *equals* $n \cdot (n + 1)/2$ *for all natural numbers* n.

[9] There is another way of finding the sum $1 + 2 + \cdots + 100$, which works like this: write the sum backwards, as $100 + 99 + \cdots + 1$. Now add the forwards and backwards versions, obtaining $101 + 101 + \cdots + 101$ (100 times), which is 10100. Since we added the sum to itself, we now divide by two to get the answer 5050. Gauss probably used this method; but the method of mathematical induction that we explore in this section is much more powerful and can be applied in a wide variety of situations.

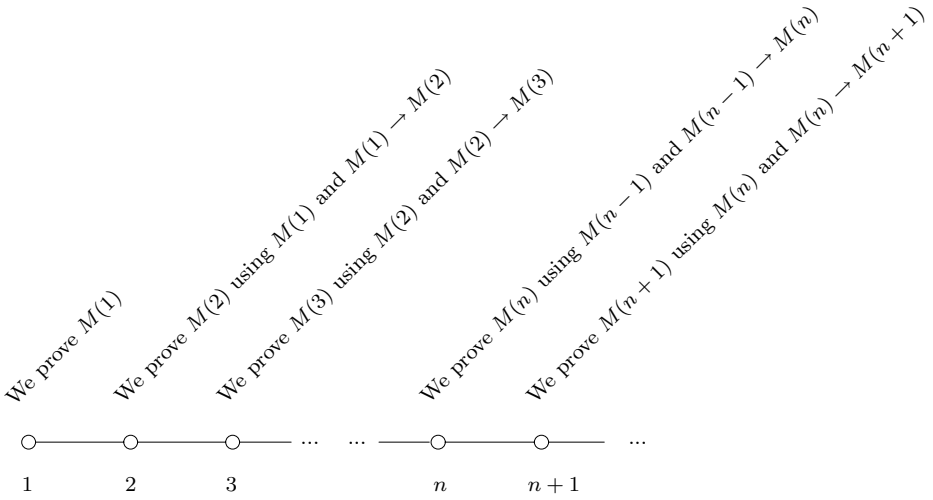


Figure 1.9. How the principle of mathematical induction works. By proving just two facts, $M(1)$ and $M(n) \to M(n+1)$ for a formal (and unconstrained) parameter n, we are able to deduce $M(k)$ for each natural number k.

Proof: We use mathematical induction. In order to reveal the fine structure of our proof we write LHS_n for the expression $1+2+3+4+\cdots+n$ and RHS_n for $n \cdot (n+1)/2$. Thus, we need to show $\mathrm{LHS}_n = \mathrm{RHS}_n$ for all $n \geq 1$.

Base case: If n equals 1, then LHS_1 is just 1 (there is only one summand), which happens to equal $\mathrm{RHS}_1 = 1 \cdot (1+1)/2$.

Inductive step: Let us assume that $\mathrm{LHS}_n = \mathrm{RHS}_n$. Recall that this assumption is called the induction hypothesis; it is the driving force of our argument. We need to show $\mathrm{LHS}_{n+1} = \mathrm{RHS}_{n+1}$, i.e. that the longer sum $1+2+3+4+\cdots+(n+1)$ equals $(n+1) \cdot ((n+1)+1)/2$. The key observation is that the sum $1+2+3+4+\cdots+(n+1)$ is nothing but the sum $(1+2+3+4+\cdots+n)+(n+1)$ of two summands, where the first one is the sum of our induction hypothesis. The latter says that $1+2+3+4+\cdots+n$ equals $n \cdot (n+1)/2$, and we are certainly entitled to substitute equals for equals in our reasoning. Thus, we compute

$$\begin{aligned} &\mathrm{LHS}_{n+1} \\ &\quad = 1+2+3+4+\cdots+(n+1) \\ &\quad = \mathrm{LHS}_n + (n+1) \quad \text{regrouping the sum} \end{aligned}$$

$$
\begin{aligned}
&= \mathrm{RHS}_n + (n+1) \quad \text{by our induction hypothesis} \\
&= \tfrac{n\cdot(n+1)}{2} + (n+1) \\
&= \tfrac{n\cdot(n+1)}{2} + \tfrac{2\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \tfrac{(n+2)\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \tfrac{((n+1)+1)\cdot(n+1)}{2} \quad \text{arithmetic} \\
&= \mathrm{RHS}_{n+1}.
\end{aligned}
$$

Since we successfully showed the base case and the inductive step, we can use mathematical induction to infer that all natural numbers n have the property stated in the theorem above. □

Actually, there are numerous variations of this principle. For example, we can think of a version in which the base case is $n = 0$, which would then cover all natural numbers including 0. Some statements hold only for all natural numbers, say, greater than 3. So you would have to deal with a base case 4, but keep the version of the inductive step (see the exercises for such an example). The use of mathematical induction typically suceeds on properties $M(n)$ that involve inductive definitions (e.g. the definition of k^l with $l \geq 0$). Sentence (3) on page 2 suggests there may be true properties $M(n)$ for which mathematical induction won't work.

Course-of-values induction. There is a variant of mathematical induction in which the induction hypothesis for proving $M(n+1)$ is not just $M(n)$, but the conjunction $M(1) \wedge M(2) \wedge \cdots \wedge M(n)$. In that variant, called *course-of-values* induction, there doesn't have to be an explicit base case at all – everything can be done in the inductive step.

How can this work without a base case? The answer is that the base case is implicitly included in the inductive step. Consider the case $n = 3$: the inductive-step instance is $M(1) \wedge M(2) \wedge M(3) \rightarrow M(4)$. Now consider $n = 1$: the inductive-step instance is $M(1) \rightarrow M(2)$. What about the case when n equals 0? In this case, there are zero formulas on the left of the $\rightarrow$, so we have to prove $M(1)$ from nothing at all. The inductive-step instance is simply the obligation to show $M(1)$. You might find it useful to modify Figure 1.9 for course-of-values induction.

Having said that the base case is implicit in course-of-values induction, it frequently turns out that it still demands special attention when you get inside trying to prove the inductive case. We will see precisely this in the two applications of course-of-values induction in the following pages.

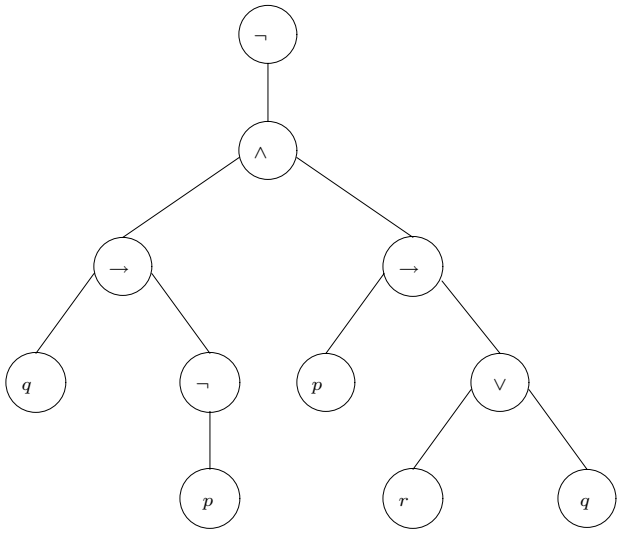


Figure 1.10. A parse tree with height 5.

In computer science, we often deal with finite structures of some kind, data structures, programs, files etc. Often we need to show that *every* instance of such a structure has a certain property. For example, the well-formed formulas of Definition 1.27 have the property that the number of '(' brackets in a particular formula equals its number of ')' brackets. We can use mathematical induction on the domain of natural numbers to prove this. In order to succeed, we somehow need to connect well-formed formulas to natural numbers.

Definition 1.32 Given a well-formed formula ϕ, we define its *height* to be 1 plus the length of the longest path of its parse tree.

For example, consider the well-formed formulas in Figures 1.3, 1.4 and 1.10. Their heights are 5, 6 and 5, respectively. In Figure 1.3, the longest path goes from $\rightarrow$ to $\wedge$ to $\vee$ to $\neg$ to r, a path of length 4, so the height is $4 + 1 = 5$. Note that the height of atoms is $1 + 0 = 1$. Since every well-formed formula has finite height, we can show statements about all well-formed formulas by mathematical induction on their height. This trick is most often called *structural induction*, an important reasoning technique in computer science. Using the notion of the height of a parse tree, we realise that structural induction is just a special case of course-of-values induction.

Theorem 1.33 *For every well-formed propositional logic formula, the number of left brackets is equal to the number of right brackets.*

PROOF: We proceed by course-of-values induction on the height of well-formed formulas ϕ. Let $M(n)$ mean 'All formulas of height n have the same number of left and right brackets.' We assume $M(k)$ for each $k < n$ and try to prove $M(n)$. Take a formula ϕ of height n.

- **Base case:** Then $n = 1$. This means that ϕ is just a propositional atom. So there are no left or right brackets, 0 equals 0.
- **Course-of-values inductive step:** Then $n > 1$ and so the root of the parse tree of ϕ must be $\neg$, $\rightarrow$, $\vee$ or $\wedge$, for ϕ is well-formed. We assume that it is $\rightarrow$, the other three cases are argued in a similar way. Then ϕ equals $(\phi_1 \rightarrow \phi_2)$ for some well-formed formulas ϕ_1 and ϕ_2 (of course, they are just the left, respectively right, linear representations of the root's two subtrees). It is clear that the heights of ϕ_1 and ϕ_2 are strictly smaller than n. Using the induction hypothesis, we therefore conclude that ϕ_1 has the same number of left and right brackets and that the same is true for ϕ_2. But in $(\phi_1 \rightarrow \phi_2)$ we added just two more brackets, one '(' and one ')'. Thus, the number of occurrences of '(' and ')' in ϕ is the same. □

The formula $(p \rightarrow (q \wedge \neg r))$ illustrates why we could not prove the above directly with mathematical induction on the height of formulas. While this formula has height 4, its two subtrees have heights 1 and 3, respectively. Thus, an induction hypothesis for height 3 would have worked for the right subtree but failed for the left subtree.

1.4.3 Soundness of propositional logic

The natural deduction rules make it possible for us to develop rigorous threads of argumentation, in the course of which we arrive at a conclusion ψ assuming certain other propositions $\phi_1, \phi_2, \ldots, \phi_n$. In that case, we said that the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. Do we have any evidence that these rules are all *correct* in the sense that valid sequents all 'preserve truth' computed by our truth-table semantics?

Given a proof of $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$, is it conceivable that there is a valuation in which ψ above is false although all propositions $\phi_1, \phi_2, \ldots, \phi_n$ are true? Fortunately, this is not the case and in this subsection we demonstrate why this is so. Let us suppose that some proof in our natural deduction calculus has established that the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. We need to show: for all valuations in which all propositions $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to `T`, ψ evaluates to `T` as well.

Definition 1.34 If, for all valuations in which all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to T, ψ evaluates to T as well, we say that

$$\phi_1, \phi_2, \ldots, \phi_n \models \psi$$

holds and call $\models$ the *semantic entailment* relation.

Let us look at some examples of this notion.

1. Does $p \wedge q \models p$ hold? Well, we have to inspect all assignments of truth values to p and q; there are four of these. Whenever such an assignment computes T for $p \wedge q$ we need to make sure that p is true as well. But $p \wedge q$ computes T only if p and q are true, so $p \wedge q \models p$ is indeed the case.
2. What about the relationship $p \vee q \models p$? There are three assignments for which $p \vee q$ computes T, so p would have to be true for all of these. However, if we assign T to q and F to p, then $p \vee q$ computes T, but p is false. Thus, $p \vee q \models p$ does not hold.
3. What if we modify the above to $\neg q, p \vee q \models p$? Notice that we have to be concerned only about valuations in which $\neg q$ *and* $p \vee q$ evaluate to T. This forces q to be false, which in turn forces p to be true. Hence $\neg q, p \vee q \models p$ is the case.
4. Note that $p \models q \vee \neg q$ holds, despite the fact that no atomic proposition on the right of $\models$ occurs on the left of $\models$.

From the discussion above we realize that a soundness argument has to show: if $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.

Theorem 1.35 (Soundness) *Let $\phi_1, \phi_2, \ldots, \phi_n$ and ψ be propositional logic formulas. If $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid, then $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.*

PROOF: Since $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid we know there is a proof of ψ from the premises $\phi_1, \phi_2, \ldots, \phi_n$. We now do a pretty slick thing, namely, we reason by *mathematical induction on the length of this proof!* The length of a proof is just the number of lines it involves. So let us be perfectly clear about what it is we mean to show. We intend to show the assertion $M(k)$:

> *'For all sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ ($n \geq 0$) which have a proof of length k, it is the case that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds.'*

by course-of-values induction on the natural number k. This idea requires

some work, though. The sequent $p \wedge q \to r \vdash p \to (q \to r)$ has a proof

1	$p \wedge q \to r$	premise
2	p	assumption
3	q	assumption
4	$p \wedge q$	$\wedge$i 2, 3
5	r	$\to$e 1, 4
6	$q \to r$	$\to$i 3–5
7	$p \to (q \to r)$	$\to$i 2–6

but if we remove the last line or several of the last lines, we no longer have a proof as the outermost box does not get closed. We get a complete proof, though, by removing the last line and re-writing the assumption of the outermost box as a premise:

1	$p \wedge q \to r$	premise
2	p	premise
3	q	assumption
4	$p \wedge q$	$\wedge$i 2, 3
5	r	$\to$e 1, 4
6	$q \to r$	$\to$i 3–5

This is a proof of the sequent $p \wedge q \to r,\ p \vdash p \to r$. The induction hypothesis then ensures that $p \wedge q \to r,\ p \vDash p \to r$ holds. But then we can also reason that $p \wedge q \to r \vDash p \to (q \to r)$ holds as well – why?

Let's proceed with our proof by induction. We assume $M(k')$ for each $k' < k$ and we try to prove $M(k)$.

Base case: a one-line proof. If the proof has length 1 ($k = 1$), then it must be of the form

1	ϕ	premise

since all other rules involve more than one line. This is the case when $n = 1$ and ϕ_1 and ψ equal ϕ, i.e. we are dealing with the sequent $\phi \vdash \phi$. Of course, since ϕ evaluates to `T` so does ϕ. Thus, $\phi \vDash \phi$ holds as claimed.

Course-of-values inductive step: Let us assume that the proof of the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ has length k and that the statement we want to prove is true for all numbers less than k. Our proof has the following structure:

$$\begin{array}{ll} 1 & \phi_1 \text{ premise} \\ 2 & \phi_2 \text{ premise} \\ & \vdots \\ n & \phi_n \text{ premise} \\ & \vdots \\ k & \psi \text{ justification} \end{array}$$

There are two things we don't know at this point. First, what is happening in between those dots? Second, what was the last rule applied, i.e. what is the justification of the last line? The first uncertainty is of no concern; this is where mathematical induction demonstrates its power. The second lack of knowledge is where all the work sits. In this generality, there is simply no way of knowing which rule was applied last, so we need to consider all such rules in turn.

1. Let us suppose that this last rule is $\wedge$i. Then we know that ψ is of the form $\psi_1 \wedge \psi_2$ and the justification in line k refers to two lines further up which have ψ_1, respectively ψ_2, as their conclusions. Suppose that these lines are k_1 and k_2. Since k_1 and k_2 are smaller than k, we see that there exist proofs of the sequents $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi_2$ with length *less than* k – just take the first k_1, respectively k_2, lines of our original proof. Using the induction hypothesis, we conclude that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1$ and $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_2$ holds. But these two relations imply that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi_1 \wedge \psi_2$ holds as well – why?
2. If ψ has been shown using the rule $\vee$e, then we must have proved, assumed or given as a premise some formula $\eta_1 \vee \eta_2$ in some line k' with $k' < k$, which was referred to via $\vee$e in the justification of line k. Thus, we have a shorter proof of the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \eta_1 \vee \eta_2$ within that proof, obtained by turning all assumptions of boxes that are open at line k' into premises. In a similar way we obtain proofs of the sequents $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vdash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vdash \psi$ from the case analysis of $\vee$e. By our induction hypothesis, we conclude that the relations $\phi_1, \phi_2, \ldots, \phi_n \vDash \eta_1 \vee \eta_2$, $\phi_1, \phi_2, \ldots, \phi_n, \eta_1 \vDash \psi$ and $\phi_1, \phi_2, \ldots, \phi_n, \eta_2 \vDash \psi$ hold. But together these three relations then force that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds as well – why?
3. You can guess by now that the rest of the argument checks each possible proof rule in turn and ultimately boils down to verifying that our natural deduction

rules behave semantically in the same way as their corresponding truth tables evaluate. We leave the details as an exercise. □

The soundness of propositional logic is useful in ensuring the *non-existence* of a proof for a given sequent. Let's say you try to prove that $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ is valid, but that your best efforts won't succeed. How could you be sure that no such proof can be found? After all, it might just be that you can't find a proof even though there is one. It suffices to find a valuation in which ϕ_i evaluate to `T` whereas ψ evaluates to `F`. Then, by definition of $\vDash$, we don't have $\phi_1, \phi_2, \ldots, \phi_2 \vDash \psi$. Using soundness, this means that $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ cannot be valid. Therefore, this sequent does not have a proof. You will practice this method in the exercises.

1.4.4 Completeness of propositional logic

In this subsection, we hope to convince you that the natural deduction rules of propositional logic are *complete*: whenever $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, then there exists a natural deduction proof for the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Combined with the soundness result of the previous subsection, we then obtain

$$\phi_1, \phi_2, \ldots, \phi_n \vdash \psi \text{ is valid} \quad \text{iff} \quad \phi_1, \phi_2, \ldots, \phi_n \vDash \psi \text{ holds.}$$

This gives you a certain freedom regarding which method you prefer to use. Often it is much easier to show one of these two relationships (although neither of the two is universally better, or easier, to establish). The first method involves a *proof search*, upon which the *logic programming* paradigm is based. The second method typically forces you to compute a truth table which is exponential in the size of occurring propositional atoms. Both methods are intractable in general but particular instances of formulas often respond differently to treatment under these two methods.

The remainder of this section is concerned with an argument saying that if $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, then $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid. Assuming that $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ holds, the argument proceeds in three steps:

Step 1: We show that $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ holds.
Step 2: We show that $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ is valid.
Step 3: Finally, we show that $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ is valid.

The first and third steps are quite easy; all the real work is done in the second one.

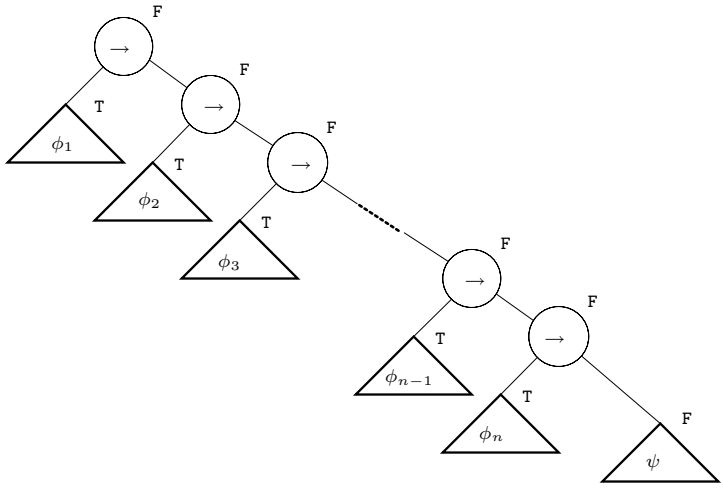


Figure 1.11. The only way this parse tree can evaluate to F. We represent parse trees for ϕ_1, ϕ_2, ..., ϕ_n as triangles as their internal structure does not concern us here.

Step 1:

Definition 1.36 A formula of propositional logic ϕ is called a *tautology* iff it evaluates to T under all its valuations, i.e. iff $\models \phi$.

Supposing that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds, let us verify that $\phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$ is indeed a tautology. Since the latter formula is a nested implication, it can evaluate to F only if all $\phi_1, \phi_2, \ldots, \phi_n$ evaluate to T *and* ψ evaluates to F; see its parse tree in Figure 1.11. But this contradicts the fact that $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds. Thus, $\models \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$ holds.

Step 2:

Theorem 1.37 *If $\models \eta$ holds, then $\vdash \eta$ is valid. In other words, if η is a tautology, then η is a theorem.*

This step is the hard one. Assume that $\models \eta$ holds. Given that η contains n distinct propositional atoms $p_1, p_2, \ldots, p_n$ we know that η evaluates to T for all 2^n lines in its truth table. (Each line lists a valuation of η.) How can we use this information to construct a proof for η? In some cases this can be done quite easily by taking a very good look at the concrete structure of η. But here we somehow have to come up with a *uniform* way of building such a proof. The key insight is to 'encode' each line in the truth table of η

as a sequent. Then we construct proofs for these 2^n sequents and assemble them into a proof of η.

Proposition 1.38 *Let ϕ be a formula such that $p_1, p_2, \ldots, p_n$ are its only propositional atoms. Let l be any line number in ϕ's truth table. For all $1 \leq i \leq n$ let $\hat{p}_i$ be p_i if the entry in line l of p_i is* `T`*, otherwise $\hat{p}_i$ is $\neg p_i$. Then we have*

1. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi$ is provable if the entry for ϕ in line l is* `T`
2. *$\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi$ is provable if the entry for ϕ in line l is* `F`

Proof: This proof is done by structural induction on the formula ϕ, that is, mathematical induction on the height of the parse tree of ϕ.

1. If ϕ is a propositional atom p, we need to show that $p \vdash p$ and $\neg p \vdash \neg p$. These have one-line proofs.
2. If ϕ is of the form $\neg\phi_1$ we again have two cases to consider. First, assume that ϕ evaluates to `T`. In this case ϕ_1 evaluates to `F`. Note that ϕ_1 has the same atomic propositions as ϕ. We may use the induction hypothesis on ϕ_1 to conclude that $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\phi_1$; but $\neg\phi_1$ is just ϕ, so we are done.
   Second, if ϕ evaluates to `F`, then ϕ_1 evaluates to `T` and we get $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ by induction. Using the rule $\neg\neg$i, we may extend the proof of $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \phi_1$ to one for $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \neg\neg\phi_1$; but $\neg\neg\phi_1$ is just $\neg\phi$, so again we are done.

The remaining cases all deal with two subformulas: ϕ equals $\phi_1 \circ \phi_2$, where $\circ$ is $\rightarrow$, $\wedge$ or $\vee$. In all these cases let $q_1, \ldots, q_l$ be the propositional atoms of ϕ_1 and $r_1, \ldots, r_k$ be the propositional atoms of ϕ_2. Then we certainly have $\{q_1, \ldots, q_l\} \cup \{r_1, \ldots, r_k\} = \{p_1, \ldots, p_n\}$. Therefore, whenever $\hat{q}_1, \ldots, \hat{q}_l \vdash \psi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \psi_2$ are valid so is $\hat{p}_1, \ldots, \hat{p}_n \vdash \psi_1 \wedge \psi_2$ using the rule $\wedge$i. In this way, we can use our induction hypothesis and only owe proofs that the conjunctions we conclude allow us to prove the desired conclusion for ϕ or $\neg\phi$ as the case may be.

3. To wit, let ϕ be $\phi_1 \rightarrow \phi_2$. If ϕ evaluates to `F`, then we know that ϕ_1 evaluates to `T` and ϕ_2 to `F`. Using our induction hypothesis, we have $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ follows. We need to show $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg(\phi_1 \rightarrow \phi_2)$; but using $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$, this amounts to proving the sequent $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \rightarrow \phi_2)$, which we leave as an exercise.
   If ϕ evaluates to `T`, then we have three cases. First, if ϕ_1 evaluates to `F` and ϕ_2 to `F`, then we get, by our induction hypothesis, that $\hat{q}_1, \ldots, \hat{q}_l \vdash \neg\phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \neg\phi_2$, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ follows. Again, we need only to show the sequent $\neg\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \rightarrow \phi_2$, which we leave as an exercise. Second, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, we use our induction hypothesis to arrive at

$\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ and have to prove $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to `T`, we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, using our induction hypothesis, and need to prove $\phi_1 \wedge \phi_2 \vdash \phi_1 \to \phi_2$, which we leave as an exercise as well.

4. If ϕ is of the form $\phi_1 \wedge \phi_2$, we are again dealing with four cases in total. First, if ϕ_1 and ϕ_2 evaluate to `T`, we get $\hat{q}_1, \ldots, \hat{q}_l \vdash \phi_1$ and $\hat{r}_1, \ldots, \hat{r}_k \vdash \phi_2$ by our induction hypothesis, so $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$ follows. Second, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, then we get $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$ using our induction hypothesis and the rule $\wedge$i as above and we need to prove $\neg\phi_1 \wedge \phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Third, if ϕ_1 and ϕ_2 evaluate to `F`, then our induction hypothesis and the rule $\wedge$i let us infer that $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$; so we are left with proving $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to `T` and ϕ_2 to `F`, we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ by our induction hypothesis and we have to show $\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \wedge \phi_2)$, which we leave as an exercise.
5. Finally, if ϕ is a disjunction $\phi_1 \vee \phi_2$, we again have four cases. First, if ϕ_1 and ϕ_2 evaluate to `F`, then our induction hypothesis and the rule $\wedge$i give us $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \neg\phi_2$ and we have to show $\neg\phi_1 \wedge \neg\phi_2 \vdash \neg(\phi_1 \vee \phi_2)$, which we leave as an exercise. Second, if ϕ_1 and ϕ_2 evaluate to `T`, then we obtain $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \phi_2$, by our induction hypothesis, and we need a proof for $\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Third, if ϕ_1 evaluates to `F` and ϕ_2 to `T`, then we arrive at $\hat{p}_1, \ldots, \hat{p}_n \vdash \neg\phi_1 \wedge \phi_2$, using our induction hypothesis, and need to establish $\neg\phi_1 \wedge \phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. Fourth, if ϕ_1 evaluates to `T` and ϕ_2 to `F`, then $\hat{p}_1, \ldots, \hat{p}_n \vdash \phi_1 \wedge \neg\phi_2$ results from our induction hypothesis and all we need is a proof for $\phi_1 \wedge \neg\phi_2 \vdash \phi_1 \vee \phi_2$, which we leave as an exercise. □

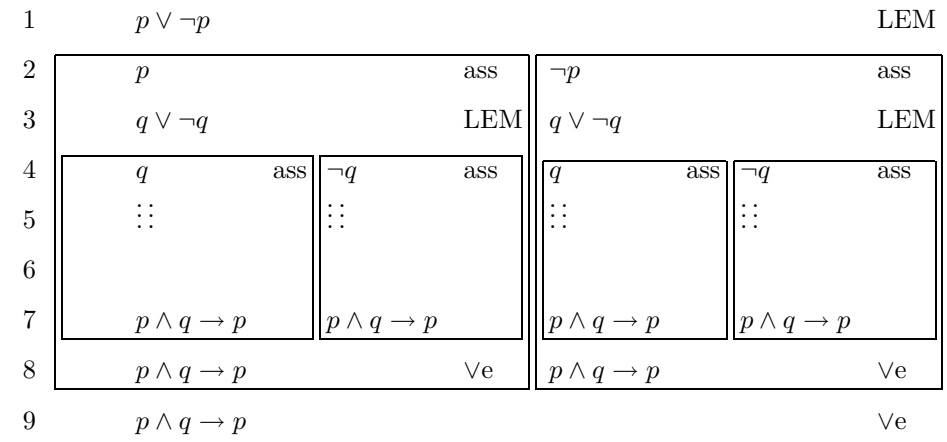
We apply this technique to the formula $\vDash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots(\phi_n \to \psi)\ldots)))$. Since it is a tautology it evaluates to `T` in all 2^n lines of its truth table; thus, the proposition above gives us 2^n many proofs of $\hat{p}_1, \hat{p}_2, \ldots, \hat{p}_n \vdash \eta$, one for each of the cases that $\hat{p}_i$ is p_i or $\neg p_i$. Our job now is to assemble all these proofs into a single proof for η which does not use any premises. We illustrate how to do this for an example, the tautology $p \wedge q \to p$.

The formula $p \wedge q \to p$ has two propositional atoms p and q. By the proposition above, we are guaranteed to have a proof for each of the four sequents

$$\begin{aligned} p, q &\vdash p \wedge q \to p \\ \neg p, q &\vdash p \wedge q \to p \\ p, \neg q &\vdash p \wedge q \to p \\ \neg p, \neg q &\vdash p \wedge q \to p. \end{aligned}$$

Ultimately, we want to prove $p \wedge q \to p$ by appealing to the four proofs of the sequents above. Thus, we somehow need to get rid of the premises on

the left-hand sides of these four sequents. This is the place where we rely on the law of the excluded middle which states $r \vee \neg r$, 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 $\vee$e. That way we can invoke all four proofs of the sequents above and use the rule $\vee$e repeatedly until we have got rid of all our premises. We spell out the combination of these four phases schematically:



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 $p \wedge q \to p$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$. Take the proof for $\vdash \phi_1 \to (\phi_2 \to (\phi_3 \to (\ldots (\phi_n \to \psi) \ldots)))$ given by step 2 and augment its proof by introducing $\phi_1, \phi_2, \ldots, \phi_n$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$.

Corollary 1.39 (Soundness and Completeness) *Let $\phi_1, \phi_2, \ldots, \phi_n, \psi$ be formulas of propositional logic. Then $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ is holds iff the sequent $\phi_1, \phi_2, \ldots, \phi_n \vdash \psi$ 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 $\phi_1, \phi_2, \ldots, \phi_2$ does not semantically entail ψ; then soundness implies that the sequent $\phi_1, \phi_2, \ldots, \phi_2 \vdash \psi$ 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 ($\models$).

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 $\models$ by applying Definition 1.34 literally is only one of many ways of deciding whether $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ holds. We now investigate various alternatives for deciding $\phi_1, \phi_2, \ldots, \phi_n \models \psi$ 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 $\neg p \vee q$ 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 $p \wedge q \to p$ and $r \vee \neg r$? At first glance, they have little in common, having different atomic formulas and different connectives. Moreover, the truth table for $p \wedge q \to p$ is four lines long, whereas the one for $r \vee \neg r$ consists of only two lines. However, both formulas are always true. This suggests that we define the equivalence of formulas ϕ and ψ via $\models$: 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 $\phi \models \psi$ and $\psi \models \phi$ hold. In that case we write $\phi \equiv \psi$. Further, we call ϕ *valid* if $\models \phi$ holds.

Note that we could also have defined $\phi \equiv \psi$ to mean that $\models (\phi \to \psi) \wedge (\psi \to \phi)$ 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

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

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 $\phi_1, \phi_2, \ldots, \phi_n$ and ψ of propositional logic, $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ 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 $\phi_1, \phi_2, \ldots, \phi_n$ 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 $\phi_1, \phi_2, \ldots, \phi_n \vDash \psi$ 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. □

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

1. using the equivalence $\phi \to \psi \equiv \neg\phi \vee \psi$ 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:

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

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. □

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 $p \vee q \rightarrow p$ is satisfiable since it computes `T` if we assign `T` to p. Clearly, $p \vee q \rightarrow p$ 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 $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$ in Figure 1.8 (page 40). For each line where $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$ 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 $\neg p \vee q$ which is readily seen to be in CNF and to be equivalent to $(p \rightarrow \neg q) \rightarrow (q \vee \neg p)$.

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 $\rightarrow$ 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` (ϕ):
/* precondition: ϕ implication free and in NNF */
/* postcondition: `CNF` (ϕ) computes an equivalent CNF for ϕ */
begin function
  case
    ϕ is a literal: **return** ϕ
    ϕ is $\phi_1 \wedge \phi_2$: **return** $\texttt{CNF}\,(\phi_1) \wedge \texttt{CNF}\,(\phi_2)$
    ϕ is $\phi_1 \vee \phi_2$: **return** $\texttt{DISTR}\,(\texttt{CNF}\,(\phi_1), \texttt{CNF}\,(\phi_2))$
  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(η1, η2):
/* precondition: η1 and η2 are in CNF */
/* postcondition: DISTR(η1, η2) computes a CNF for η1 ∨ η2 */
begin function
  case
    η1 is η11 ∧ η12: return DISTR(η11, η2) ∧ DISTR(η12, η2)
    η2 is η21 ∧ η22: return DISTR(η1, η21) ∧ DISTR(η1, η22)
    otherwise (= no conjunctions): return η1 ∨ η2
  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 (ϕ):
/* precondition: ϕ is implication free */
/* postcondition: NNF (ϕ) computes a NNF for ϕ */
begin function
  case
    ϕ is a literal: **return** ϕ
    ϕ is $\neg\neg\phi_1$: **return** NNF (ϕ_1)
    ϕ is $\phi_1 \wedge \phi_2$: **return** NNF $(\phi_1) \wedge$ NNF (ϕ_2)
    ϕ is $\phi_1 \vee \phi_2$: **return** NNF $(\phi_1) \vee$ NNF (ϕ_2)
    ϕ is $\neg(\phi_1 \wedge \phi_2)$: **return** NNF $(\neg\phi_1) \vee$ NNF $(\neg\phi_2)$
    ϕ 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 $\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL\_FREE}\,(\neg p \wedge q \to p \wedge (r \to q))))$. 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 $\texttt{IMPL\_FREE}\ (\phi)$:

$$\begin{aligned}
\texttt{IMPL_FREE}\,(\phi) &= \neg\texttt{IMPL_FREE}\,(\neg p \wedge q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\texttt{IMPL_FREE}\,\neg p) \wedge (\texttt{IMPL_FREE}\,q)) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg((\neg p) \wedge \texttt{IMPL_FREE}\,q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee \texttt{IMPL_FREE}\,(p \wedge (r \to q)) \\
&= \neg(\neg p \wedge q) \vee ((\texttt{IMPL_FREE}\,p) \wedge \texttt{IMPL_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge \texttt{IMPL_FREE}\,(r \to q)) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg(\texttt{IMPL_FREE}\,r) \vee (\texttt{IMPL_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee (\texttt{IMPL_FREE}\,q))) \\
&= \neg(\neg p \wedge q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

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

$$\begin{aligned}
\texttt{NNF}\ (\texttt{IMPL_FREE}\ \phi) &= \texttt{NNF}\ (\neg(\neg p \wedge q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= \texttt{NNF}\ (\neg(\neg p) \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (\texttt{NNF}\ (\neg\neg p)) \vee (\texttt{NNF}\ (\neg q)) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee (\texttt{NNF}\ (\neg q))) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee \texttt{NNF}\ (p \wedge (\neg r \vee q)) \\
&= (p \vee \neg q) \vee ((\texttt{NNF}\ p) \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge (\texttt{NNF}\ (\neg r \vee q))) \\
&= (p \vee \neg q) \vee (p \wedge ((\texttt{NNF}\ (\neg r)) \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee (\texttt{NNF}\ q))) \\
&= (p \vee \neg q) \vee (p \wedge (\neg r \vee q)).
\end{aligned}$$

Third, we finish it off with

$$\begin{aligned}
\texttt{CNF}\,(\texttt{NNF}\,(\texttt{IMPL_FREE}\,\phi)) &= \texttt{CNF}\,((p \vee \neg q) \vee (p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(\texttt{CNF}\,(p \vee \neg q), \texttt{CNF}\,(p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(p \vee \neg q, \texttt{CNF}\,(p \wedge (\neg r \vee q))) \\
&= \texttt{DISTR}\,(p \vee \neg q, p \wedge (\neg r \vee q)) \\
&= \texttt{DISTR}\,(p \vee \neg q, p) \wedge \texttt{DISTR}\,(p \vee \neg q, \neg r \vee q) \\
&= (p \vee \neg q \vee p) \wedge \texttt{DISTR}\,(p \vee \neg q, \neg r \vee q) \\
&= (p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q)\ .
\end{aligned}$$

The formula $(p \vee \neg q \vee p) \wedge (p \vee \neg q \vee \neg r \vee q)$ 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 $p \vee \neg q$. 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 $\texttt{CNF}\,(p \vee \neg q)$ and $\texttt{CNF}\,(p \wedge (\neg r \vee q))$. They return their input as a result since the input is already in conjunctive normal form.

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

$$\begin{aligned}
\texttt{IMPL_FREE}\,(\phi) &= \neg(\texttt{IMPL_FREE}\ r) \vee \texttt{IMPL_FREE}\,(s \to (t \wedge s \to r)) \\
&= \neg r \vee \texttt{IMPL_FREE}\,(s \to (t \wedge s \to r)) \\
&= \neg r \vee (\neg(\texttt{IMPL_FREE}\,s) \vee \texttt{IMPL_FREE}\,(t \wedge s \to r)) \\
&= \neg r \vee (\neg s \vee \texttt{IMPL_FREE}\,(t \wedge s \to r)) \\
&= \neg r \vee (\neg s \vee (\neg(\texttt{IMPL_FREE}\,(t \wedge s)) \vee \texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg((\texttt{IMPL_FREE}\,t) \wedge (\texttt{IMPL_FREE}\,s)) \vee \texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge (\texttt{IMPL_FREE}\,s)) \vee (\texttt{IMPL_FREE}\,r))) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge s)) \vee (\texttt{IMPL_FREE}\,r)) \\
&= \neg r \vee (\neg s \vee (\neg(t \wedge s)) \vee r)
\end{aligned}$$

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

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 $\bot$ ('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:

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

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

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

Examples of formulas which are *not* Horn formulas are

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

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, $\neg q \wedge r$, 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, $p_{13} \wedge p_{27}$, 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 $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ 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 $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ 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(ϕ):
/* precondition: ϕ is a Horn formula */
/* postcondition: HORN(ϕ) decides the satisfiability for ϕ */
begin function
    mark all occurrences of $\top$ in ϕ;
    while there is a conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \to P'$ of ϕ
        such that all P_j are marked but P' isn't **do**
        mark P'
    end while
    if $\bot$ 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 $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ 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 $\bot$ is marked, then there has to be some conjunct $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow \bot$ of ϕ such that all P_i are marked as well. By (1.8) that conjunct of ϕ evaluates to $\mathtt{T} \rightarrow \mathtt{F} = \mathtt{F}$ whenever ϕ is true. As this is impossible the reply 'unsatisfiable' is correct. Second, if $\bot$ is not marked, we simply assign $\mathtt{T}$ to all marked atoms and $\mathtt{F}$ 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 $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ 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 $P_1 \wedge P_2 \wedge \cdots \wedge P_{k_i} \rightarrow P'$ 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 $\bot$ 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:

'All marked subformulas evaluate to their mark value
for all valuations in which ϕ evaluates to $\mathtt{T}$.' (1.9)

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

time, (1.9) serves as a guide for designing an algorithm and as an invariant for proving its correctness.

1.6.1 A linear solver

We will execute this marking algorithm on the parse tree of formulas, except that we will translate formulas into the adequate fragment

$$\phi ::= p \mid (\neg\phi) \mid (\phi \wedge \phi) \tag{1.10}$$

and then share common subformulas of the resulting parse tree, making the tree into a directed, acyclic graph (DAG). The inductively defined translation

$$\begin{array}{ll} T(p) = p & T(\neg\phi) = \neg T(\phi) \\ T(\phi_1 \wedge \phi_2) = T(\phi_1) \wedge T(\phi_2) & T(\phi_1 \vee \phi_2) = \neg(\neg T(\phi_1) \wedge \neg T(\phi_2)) \\ T(\phi_1 \rightarrow \phi_2) = \neg(T(\phi_1) \wedge \neg T(\phi_2)) & \end{array}$$

transforms formulas generated by (1.3) into formulas generated by (1.10) such that ϕ and $T(\phi)$ are semantically equivalent and have the same propositional atoms. Therefore, ϕ is satisfiable iff $T(\phi)$ is satisfiable; and the set of valuations for which ϕ is true equals the set of valuations for which $T(\phi)$ is true. The latter ensures that the diagnostics of a SAT solver, applied to $T(\phi)$, is meaningful for the original formula ϕ. In the exercises, you are asked to prove these claims.

Example 1.48 For the formula ϕ being $p \wedge \neg(q \vee \neg p)$ we compute $T(\phi) = p \wedge \neg\neg(\neg q \wedge \neg\neg p)$. The parse tree and DAG of $T(\phi)$ are depicted in Figure 1.12.

Any valuation that makes $p \wedge \neg\neg(\neg q \wedge \neg\neg p)$ true has to assign `T` to the topmost $\wedge$-node in its DAG of Figure 1.12. But that forces the mark `T` on the p-node and the topmost $\neg$-node. In the same manner, we arrive at a complete set of constraints in Figure 1.13, where the time stamps '1:' etc indicate the order in which we applied our intuitive reasoning about these constraints; this order is generally not unique.

The formal set of rules for forcing new constraints from old ones is depicted in Figure 1.14. A small circle indicates any node ($\neg$, $\wedge$ or atom). The force laws for negation, $\neg_{\mathrm{t}}$ and $\neg_{\mathrm{f}}$, indicate that a truth constraint on a $\neg$-node forces its dual value at its sub-node and vice versa. The law $\wedge_{\mathrm{te}}$ propagates a `T` constraint on a $\wedge$-node to its two sub-nodes; dually, $\wedge_{\mathrm{ti}}$ forces a `T` mark on a $\wedge$-node if both its children have that mark. The laws $\wedge_{\mathrm{fl}}$ and $\wedge_{\mathrm{fr}}$ force a `F` constraint on a $\wedge$-node if any of its sub-nodes has a `F` value. The laws $\wedge_{\mathrm{fll}}$

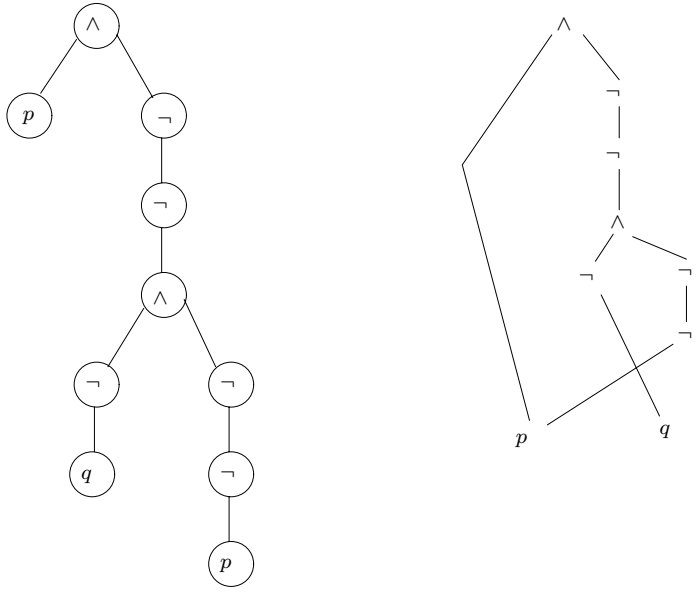


Figure 1.12. Parse tree (left) and directed acyclic graph (right) of the formula from Example 1.48. The p-node is shared on the right.

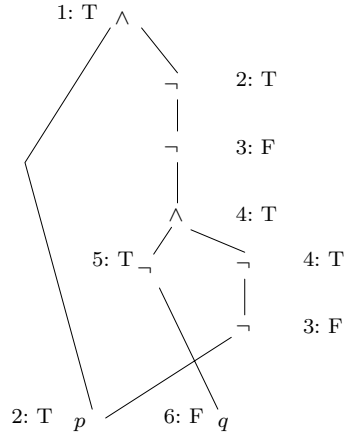


Figure 1.13. A witness to the satisfiability of the formula represented by this DAG.

and $\wedge_{\mathrm{frr}}$ are more complex: if an $\wedge$-node has a `F` constraint and one of its sub-nodes has a `T` constraint, then the *other* sub-node obtains a `F`-constraint. Please check that all constraints depicted in Figure 1.13 are derivable from these rules. The fact that each node in a DAG obtained a forced marking does not yet show that this is a witness to the satisfiability of the formula

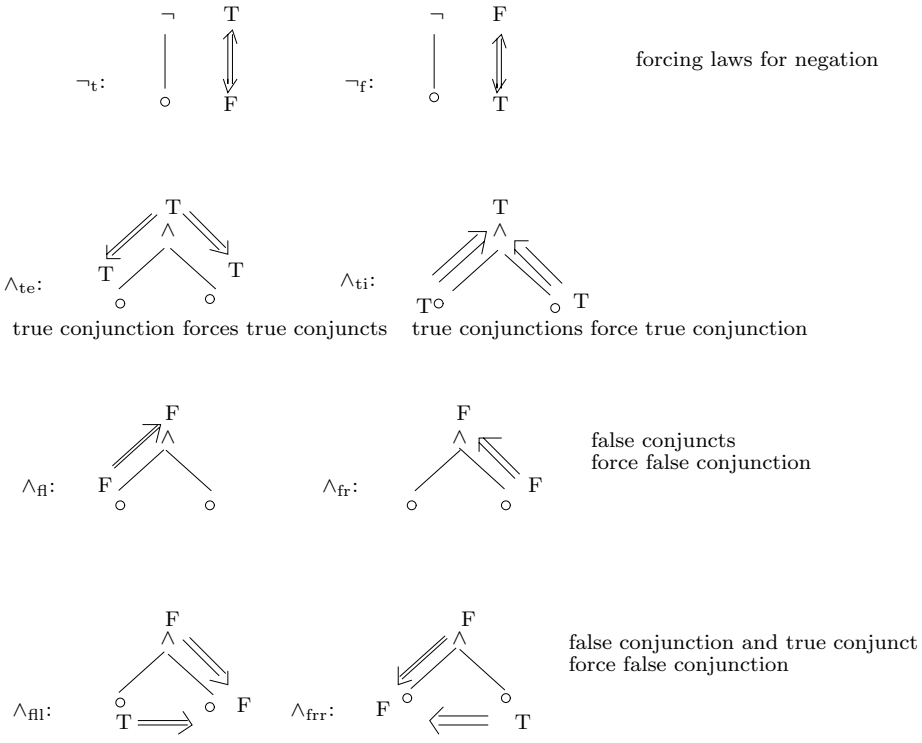


Figure 1.14. Rules for flow of constraints in a formula's DAG. Small circles indicate arbitrary nodes ($\neg$, $\wedge$ or atom). Note that the rules $\wedge_{\mathrm{fll}}$, $\wedge_{\mathrm{frr}}$ and $\wedge_{\mathrm{ti}}$ require that the source constraints of both $\Longrightarrow$ are present.

represented by this DAG. A post-processing phase takes the marks for all atoms and re-computes marks of all other nodes in a bottom-up manner, as done in Section 1.4 on parse trees. Only if the resulting marks match the ones we computed have we found a witness. Please verify that this is the case in Figure 1.13.

We can apply SAT solvers to checking whether sequents are valid. For example, the sequent $p \wedge q \to r \vdash p \to q \to r$ is valid iff $(p \wedge q \to r) \to p \to q \to r$ is a theorem (why?) iff $\phi = \neg((p \wedge q \to r) \to p \to q \to r)$ is *not* satisfiable. The DAG of $T(\phi)$ is depicted in Figure 1.15. The annotations "1" etc indicate which nodes represent which sub-formulas. Notice that such DAGs may be constructed by applying the translation clauses for T to sub-formulas in a bottom-up manner – sharing equal subgraphs were applicable.

The findings of our SAT solver can be seen in Figure 1.16. The solver concludes that the indicated node requires the marks `T` *and* `F` for (1.9) to be met. Such contradictory constraints therefore imply that all formulas $T(\phi)$ whose DAG equals that of this figure are not satisfiable. In particular, all

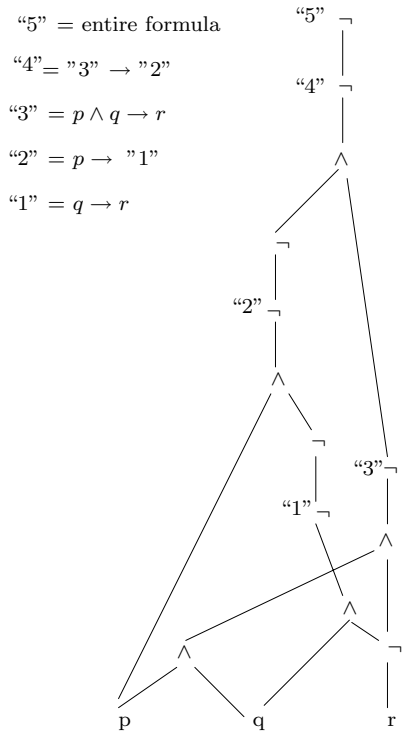


Figure 1.15. The DAG for the translation of $\neg((p \wedge q \to r) \to p \to q \to r)$. Labels "1" etc indicate which nodes represent what subformulas.

such ϕ are unsatisfiable. This SAT solver has a linear running time in the size of the DAG for $T(\phi)$. Since that size is a linear function of the length of ϕ – the translation T causes only a linear blow-up – our SAT solver has a linear running time in the length of the formula. This linearity came with a price: our linear solver fails for all formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

1.6.2 A cubic solver

When we applied our linear SAT solver, we saw two possible outcomes: we either detected contradictory constraints, meaning that no formula represented by the DAG is satisfiable (e.g. Fig. 1.16); or we managed to force consistent constraints on all nodes, in which case all formulas represented by this DAG are satisfiable with those constraints as a witness (e.g. Fig. 1.13). Unfortunately, there is a third possibility: all forced constraints are consistent with each other, but not all nodes are constrained! We already remarked that this occurs for formulas of the form $\neg(\phi_1 \wedge \phi_2)$.

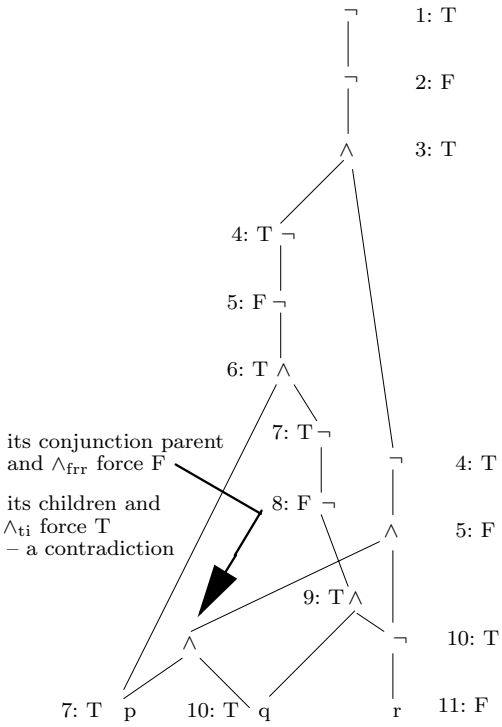


Figure 1.16. The forcing rules, applied to the DAG of Figure 1.15, detect contradictory constraints at the indicated node – implying that the initial constraint '1:T' cannot be realized. Thus, formulas represented by this DAG are not satisfiable.

Recall that checking validity of formulas in CNF is very easy. We already hinted at the fact that checking satisfiability of formulas in CNF is hard. To illustrate, consider the formula

$$((p \vee (q \vee r)) \wedge ((p \vee \neg q) \wedge ((q \vee \neg r) \wedge ((r \vee \neg p) \wedge (\neg p \vee (\neg q \vee \neg r))))))) \tag{1.11}$$

in CNF – based on Example 4.2, page 77, in [Pap94]. Intuitively, this formula should not be satisfiable. The first and last clause in (1.11) 'say' that at least one of p, q, and r are false and true (respectively). The remaining three clauses, in their conjunction, 'say' that p, q, and r all have the same truth value. This cannot be satisfiable, and a good SAT solver should discover this without any user intervention. Unfortunately, our linear SAT solver can neither detect inconsistent constraints nor compute constraints for all nodes. Figure 1.17 depicts the DAG for $T(\phi)$, where ϕ is as in (1.11); and reveals

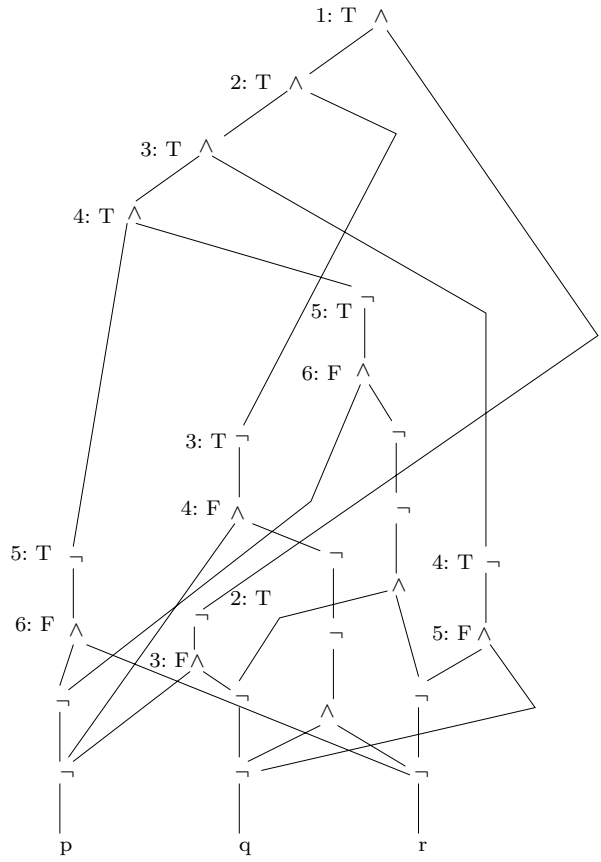


Figure 1.17. The DAG for the translation of the formula in (1.11). It has a $\wedge$-spine of length 4 as it is a conjunction of five clauses. Its linear analysis gets stuck: all forced constraints are consistent with each other but several nodes, including all atoms, are unconstrained.

that our SAT solver got stuck: no inconsistent constraints were found and not all nodes obtained constraints; in particular, no atom received a mark! So how can we improve this analysis? Well, we can mimic the role of LEM to improve the precision of our SAT solver. For the DAG with marks as in Figure 1.17, pick any node n that is not yet marked. Then *test* node n by making two independent computations:

1. determine which *temporary* marks are forced by adding to the marks in Figure 1.17 the `T` mark only to n; and
2. determine which *temporary* marks are forced by adding, again to the marks in Figure 1.17, the `F` mark only to n.

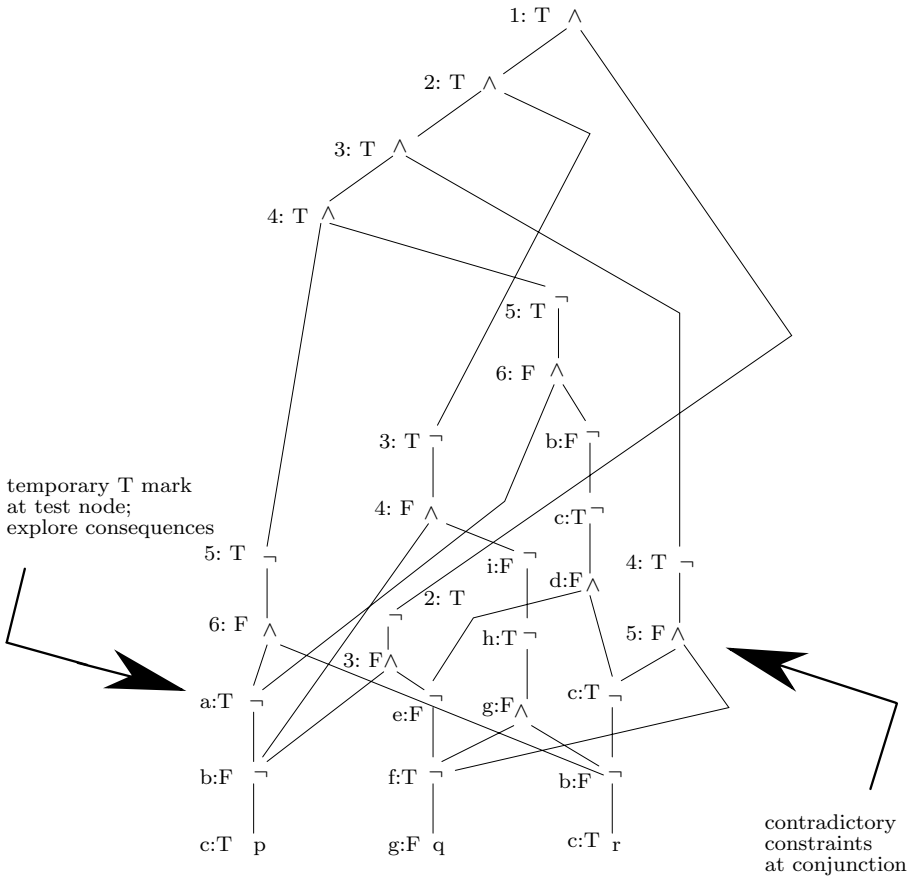


Figure 1.18. Marking an unmarked node with T and exploring what new constraints would follow from this. The analysis shows that this test marking causes contradictory constraints. We use lowercase letters 'a:' etc to denote temporary marks.

If both runs find contradictory constraints, the algorithm stops and reports that $T(\phi)$ is unsatisfiable. Otherwise, all nodes that received the same mark in both of these runs receive that very mark as a *permanent* one; that is, we update the mark state of Figure 1.17 with all such shared marks.

We test any further unmarked nodes in the same manner until we either find contradictory *permanent* marks, a complete witness to satisfiability (all nodes have consistent marks), or we have tested *all* currently unmarked nodes in this manner without detecting any shared marks. Only in the latter case does the analysis terminate without knowing whether the formulas represented by that DAG are satisfiable.

Example 1.49 We revisit our stuck analysis of Figure 1.17. We test a ¬-node and explore the consequences of setting that ¬-node's mark to `T`; Figure 1.18 shows the result of that analysis. Dually, Figure 1.19 tests the consequences of setting that ¬-node's mark to `F`. Since both runs reveal a contradiction, the algorithm terminates, ruling that the formula in (1.11) is not satisfiable.

In the exercises, you are asked to show that the specification of our cubic SAT solver is sound. Its running time is indeed cubic in the size of the DAG (and the length of original formula). One factor stems from the linear SAT solver used in each test run. A second factor is introduced since each unmarked node has to be tested. The third factor is needed since each new permanent mark causes *all* unmarked nodes to be tested again.

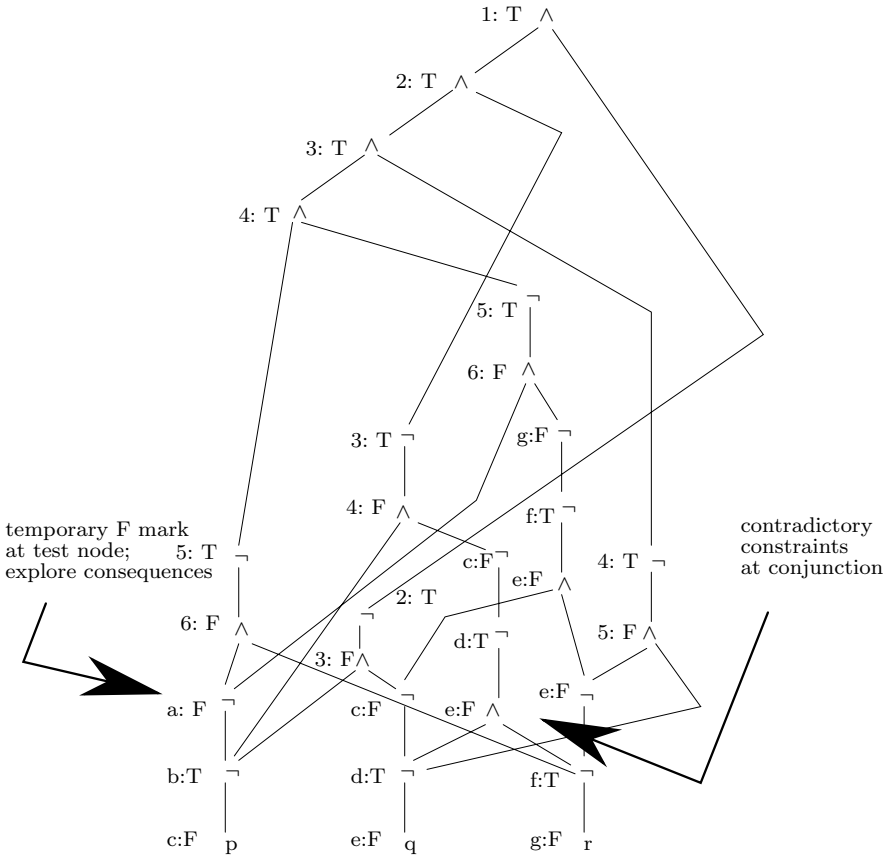


Figure 1.19. Marking the same unmarked node with `F` and exploring what new constraints would follow from this. The analysis shows that this test marking also causes contradictory constraints.

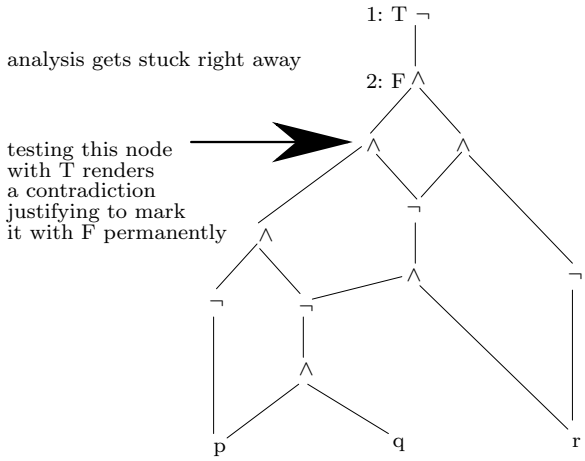


Figure 1.20. Testing the indicated node with T causes contradictory constraints, so we may mark that node with F permanently. However, our algorithm does not seem to be able to decide satisfiability of this DAG even with that optimization.

We deliberately under-specified our cubic SAT solver, but any implementation or optimization decisions need to secure soundness of the analysis. All replies of the form

1. 'The input formula is not satisfiable' and
2. 'The input formula is satisfiable under the following valuation ...'

have to be correct. The third form of reply 'Sorry, I could not figure this one out.' is correct by definition. :-) We briefly discuss two sound modifications to the algorithm that introduce some overhead, but may cause the algorithm to decide many more instances. Consider the state of a DAG right after we have explored consequences of a temporary mark on a test node.

1. If that state – permanent plus temporary markings – contains contradictory constraints, we can erase all temporary marks and mark the test node permanently with the dual mark of its test. That is, if marking node n with v resulted in a contradiction, it will get a permanent mark $\overline{v}$, where $\overline{\mathtt{T}} = \mathtt{F}$ and $\overline{\mathtt{F}} = \mathtt{T}$; otherwise
2. if that state managed to mark *all* nodes with consistent constraints, we report these markings as a witness of satisfiability and terminate the algorithm.

If none of these cases apply, we proceed as specified: promote shared marks of the two test runs to permanent ones, if applicable.

Example 1.50 To see how one of these optimizations may make a difference, consider the DAG in Figure 1.20. If we test the indicated node with

T, contradictory constraints arise. Since any witness of satisfiability has to assign some value to that node, we infer that it cannot be T. Thus, we may permanently assign mark F to that node. For this DAG, such an optimization does not seem to help. No test of an unmarked node detects a shared mark or a shared contradiction. Our cubic SAT solver fails for this DAG.

1.7 Exercises

Exercises 1.1

1. Use $\neg$, $\rightarrow$, $\wedge$ and $\vee$ to express the following declarative sentences in propositional logic; in each case state what your respective propositional atoms p, q, etc. mean:
   * (a) If the sun shines today, then it won't shine tomorrow.
   (b) Robert was jealous of Yvonne, or he was not in a good mood.
   (c) If the barometer falls, then either it will rain or it will snow.
   * (d) If a request occurs, then either it will eventually be acknowledged, or the requesting process won't ever be able to make progress.
   (e) Cancer will not be cured unless its cause is determined and a new drug for cancer is found.
   (f) If interest rates go up, share prices go down.
   (g) If Smith has installed central heating, then he has sold his car or he has not paid his mortgage.
   * (h) Today it will rain or shine, but not both.
   * (i) If Dick met Jane yesterday, they had a cup of coffee together, or they took a walk in the park.
   (j) No shoes, no shirt, no service.
   (k) My sister wants a black and white cat.
2. The formulas of propositional logic below implicitly assume the binding priorities of the logical connectives put forward in Convention 1.3. Make sure that you fully understand those conventions by reinserting as many brackets as possible. For example, given $p \wedge q \rightarrow r$, change it to $(p \wedge q) \rightarrow r$ since $\wedge$ binds more tightly than $\rightarrow$.
   * (a) $\neg p \wedge q \rightarrow r$
   (b) $(p \rightarrow q) \wedge \neg(r \vee p \rightarrow q)$
   * (c) $(p \rightarrow q) \rightarrow (r \rightarrow s \vee t)$
   (d) $p \vee (\neg q \rightarrow p \wedge r)$
   * (e) $p \vee q \rightarrow \neg p \wedge r$
   (f) $p \vee p \rightarrow \neg q$
   * (g) Why is the expression $p \vee q \wedge r$ problematic?

Exercises 1.2

1. Prove the validity of the following sequents:
   (a) $(p \wedge q) \wedge r, s \wedge t \vdash q \wedge s$

(b) $p \wedge q \vdash q \wedge p$
* (c) $(p \wedge q) \wedge r \vdash p \wedge (q \wedge r)$
(d) $p \to (p \to q), p \vdash q$
* (e) $q \to (p \to r), \neg r, q \vdash \neg p$
* (f) $\vdash (p \wedge q) \to p$
(g) $p \vdash q \to (p \wedge q)$
* (h) $p \vdash (p \to q) \to q$
* (i) $(p \to r) \wedge (q \to r) \vdash p \wedge q \to r$
* (j) $q \to r \vdash (p \to q) \to (p \to r)$
(k) $p \to (q \to r), p \to q \vdash p \to r$
* (l) $p \to q, r \to s \vdash p \vee r \to q \vee s$
(m) $p \vee q \vdash r \to (p \vee q) \wedge r$
* (n) $(p \vee (q \to p)) \wedge q \vdash p$
* (o) $p \to q,\ r \to s \vdash p \wedge r \to q \wedge s$
(p) $p \to q \vdash ((p \wedge q) \to p) \wedge (p \to (p \wedge q))$
(q) $\vdash q \to (p \to (p \to (q \to p)))$
* (r) $p \to q \wedge r \vdash (p \to q) \wedge (p \to r)$
(s) $(p \to q) \wedge (p \to r) \vdash p \to q \wedge r$
(t) $\vdash (p \to q) \to ((r \to s) \to (p \wedge r \to q \wedge s))$; here you might be able to 'recycle' and augment a proof from a previous exercise.
(u) $p \to q \vdash \neg q \to \neg p$
* (v) $p \vee (p \wedge q) \vdash p$
(w) $r, p \to (r \to q) \vdash p \to (q \wedge r)$
* (x) $p \to (q \vee r), q \to s, r \to s \vdash p \to s$
* (y) $(p \wedge q) \vee (p \wedge r) \vdash p \wedge (q \vee r)$.

2. For the sequents below, show which ones are valid and which ones aren't:
* (a) $\neg p \to \neg q \vdash q \to p$
* (b) $\neg p \vee \neg q \vdash \neg(p \wedge q)$
* (c) $\neg p, p \vee q \vdash q$
* (d) $p \vee q, \neg q \vee r \vdash p \vee r$
* (e) $p \to (q \vee r), \neg q, \neg r \vdash \neg p$ without using the MT rule
* (f) $\neg p \wedge \neg q \vdash \neg(p \vee q)$
* (g) $p \wedge \neg p \vdash \neg(r \to q) \wedge (r \to q)$
(h) $p \to q, s \to t \vdash p \vee s \to q \wedge t$
* (i) $\neg(\neg p \vee q) \vdash p$.

3. Prove the validity of the sequents below:
(a) $\neg p \to p \vdash p$
(b) $\neg p \vdash p \to q$
(c) $p \vee q, \neg q \vdash p$
* (d) $\vdash \neg p \to (p \to (p \to q))$
(e) $\neg(p \to q) \vdash q \to p$
(f) $p \to q \vdash \neg p \vee q$
(g) $\vdash \neg p \vee q \to (p \to q)$

(h) $p \to (q \vee r),\ \neg q,\ \neg r \vdash \neg p$
(i) $(c \wedge n) \to t,\ h \wedge \neg s,\ h \wedge \neg(s \vee c) \to p \vdash (n \wedge \neg t) \to p$
(j) the two sequents implict in (1.2) on page 20
(k) $q \vdash (p \wedge q) \vee (\neg p \wedge q)$ using LEM
(l) $\neg(p \wedge q) \vdash \neg p \vee \neg q$
(m) $p \wedge q \to r \vdash (p \to r) \vee (q \to r)$
* (n) $p \wedge q \vdash \neg(\neg p \vee \neg q)$
(o) $\neg(\neg p \vee \neg q) \vdash p \wedge q$
(p) $p \to q \vdash \neg p \vee q$ possibly without using LEM?
* (q) $\vdash (p \to q) \vee (q \to r)$ using LEM
(r) $p \to q,\ \neg p \to r,\ \neg q \to \neg r \vdash q$
(s) $p \to q,\ r \to \neg t,\ q \to r \vdash p \to \neg t$
(t) $(p \to q) \to r,\ s \to \neg p,\ t,\ \neg s \wedge t \to q \vdash r$
(u) $(s \to p) \vee (t \to q) \vdash (s \to q) \vee (t \to p)$
(v) $(p \wedge q) \to r,\ r \to s,\ q \wedge \neg s \vdash \neg p$.

4. Explain why intuitionistic logicians also reject the proof rule PBC.
5. Prove the following theorems of propositional logic:
* (a) $((p \to q) \to q) \to ((q \to p) \to p)$
(b) Given a proof for the sequent of the previous item, do you now have a quick argument for $((q \to p) \to p) \to ((p \to q) \to q)$?
(c) $((p \to q) \wedge (q \to p)) \to ((p \vee q) \to (p \wedge q))$
* (d) $(p \to q) \to ((\neg p \to q) \to q)$.
6. Natural deduction is not the only possible formal framework for proofs in propositional logic. As an abbreviation, we write Γ to denote any finite sequence of formulas $\phi_1, \phi_2, \ldots, \phi_n$ $(n \geq 0)$. Thus, any sequent may be written as $\Gamma \vdash \psi$ for an appropriate, possibly empty, Γ. In this exercise we propose a different notion of proof, which states rules for transforming valid sequents into valid sequents. For example, if we have already a proof for the sequent $\Gamma, \phi \vdash \psi$, then we obtain a proof of the sequent $\Gamma \vdash \phi \to \psi$ by augmenting this very proof with one application of the rule $\to$i. The new approach expresses this as an inference rule between sequents:

$$\frac{\Gamma, \phi \vdash \psi}{\Gamma \vdash \phi \to \psi} \to\text{i}.$$

The rule 'assumption' is written as

$$\frac{}{\phi \vdash \phi}\ \text{assumption}$$

i.e. the premise is empty. Such rules are called axioms.
(a) Express all remaining proof rules of Figure 1.2 in such a form. (Hint: some of your rules may have more than one premise.)
(b) Explain why proofs of $\Gamma \vdash \psi$ in this new system have a tree-like structure with $\Gamma \vdash \psi$ as root.
(c) Prove $p \vee (p \wedge q) \vdash p$ in your new proof system.

7. Show that $\sqrt{2}$ cannot be a rational number. Proceed by proof by contradiction: assume that $\sqrt{2}$ is a fraction k/l with integers k and $l \neq 0$. On squaring both sides we get $2 = k^2/l^2$, or equivalently $2l^2 = k^2$. We may assume that any common 2 factors of k and l have been cancelled. Can you now argue that $2l^2$ has a different number of 2 factors from k^2? Why would that be a contradiction and to what?
8. There is an alternative approach to treating negation. One could simply ban the operator $\neg$ from propositional logic and think of $\phi \to \bot$ as 'being' $\neg\phi$. Naturally, such a logic cannot rely on the natural deduction rules for negation. Which of the rules $\neg$i, $\neg$e, $\neg\neg$e and $\neg\neg$i can you simulate with the remaining proof rules by letting $\neg\phi$ be $\phi \to \bot$?
9. Let us introduce a new connective $\phi \leftrightarrow \psi$ which should abbreviate $(\phi \to \psi) \wedge (\psi \to \phi)$. Design introduction and elimination rules for $\leftrightarrow$ and show that they are derived rules if $\phi \leftrightarrow \psi$ is interpreted as $(\phi \to \psi) \wedge (\psi \to \phi)$.

Exercises 1.3

In order to facilitate reading these exercises we assume below the usual conventions about binding priorities agreed upon in Convention 1.3.

1. Given the following formulas, draw their corresponding parse tree:
   (a) p
 * (b) $p \wedge q$
   (c) $p \wedge \neg q \to \neg p$
 * (d) $p \wedge (\neg q \to \neg p)$
   (e) $p \to (\neg q \vee (q \to p))$
 * (f) $\neg((\neg q \wedge (p \to r)) \wedge (r \to q))$
   (g) $\neg p \vee (p \to q)$
   (h) $(p \wedge q) \to (\neg r \vee (q \to r))$
   (i) $((s \vee (\neg p)) \to (\neg p))$
   (j) $(s \vee ((\neg p) \to (\neg p)))$
   (k) $(((s \to (r \vee l)) \vee ((\neg q) \wedge r)) \to ((\neg(p \to s)) \to r))$
   (l) $(p \to q) \wedge (\neg r \to (q \vee (\neg p \wedge r)))$.
2. For each formula below, list all its subformulas:
 * (a) $p \to (\neg p \vee (\neg\neg q \to (p \wedge q)))$
   (b) $(s \to r \vee l) \vee (\neg q \wedge r) \to (\neg(p \to s) \to r)$
   (c) $(p \to q) \wedge (\neg r \to (q \vee (\neg p \wedge r)))$.
3. Draw the parse tree of a formula ϕ of propositional logic which is
 * (a) a negation of an implication
   (b) a disjunction whose disjuncts are both conjunctions
 * (c) a conjunction of conjunctions.
4. For each formula below, draw its parse tree and list all subformulas:
 * (a) $\neg(s \to (\neg(p \to (q \vee \neg s))))$
   (b) $((p \to \neg q) \vee (p \wedge r) \to s) \vee \neg r$.

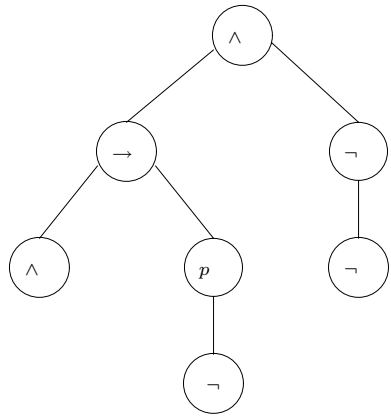


Figure 1.21. A tree that represents an ill-formed formula.

* 5. For the parse tree in Figure 1.22 find the logical formula it represents.

6. For the trees below, find their linear representations and check whether they correspond to well-formed formulas:
   (a) the tree in Figure 1.10 on page 44
   (b) the tree in Figure 1.23.

* 7. Draw a parse tree that represents an ill-formed formula such that
   (a) one can extend it by adding one or several subtrees to obtain a tree that represents a well-formed formula;
   (b) it is inherently ill-formed; i.e. any extension of it could not correspond to a well-formed formula.

8. Determine, by trying to draw parse trees, which of the following formulas are well-formed:
   (a) $p \wedge \neg(p \vee \neg q) \rightarrow (r \rightarrow s)$
   (b) $p \wedge \neg(p \vee q \wedge s) \rightarrow (r \rightarrow s)$
   (c) $p \wedge \neg(p \vee \wedge s) \rightarrow (r \rightarrow s)$.
   Among the ill-formed formulas above which ones, and in how many ways, could you 'fix' by the insertion of brackets only?

Exercises 1.4

* 1. Construct the truth table for $\neg p \vee q$ and verify that it coincides with the one for $p \rightarrow q$. (By 'coincide' we mean that the respective columns of `T` and `F` values are the same.)

2. Compute the complete truth table of the formula
   * (a) $((p \rightarrow q) \rightarrow p) \rightarrow p$
   (b) represented by the parse tree in Figure 1.3 on page 34

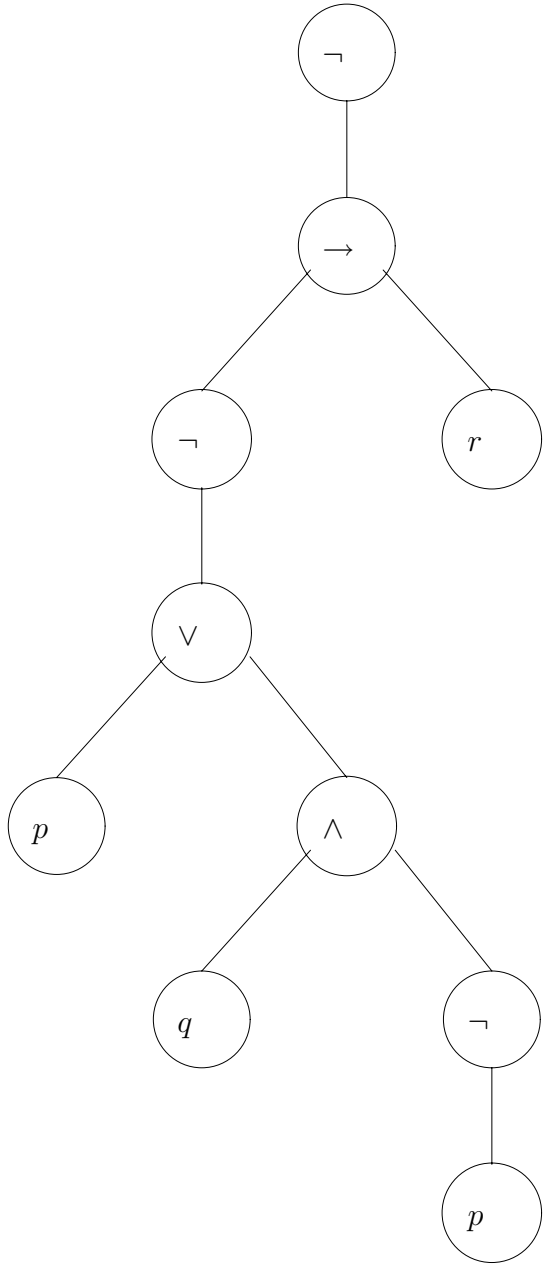


Figure 1.22. A parse tree of a negated implication.

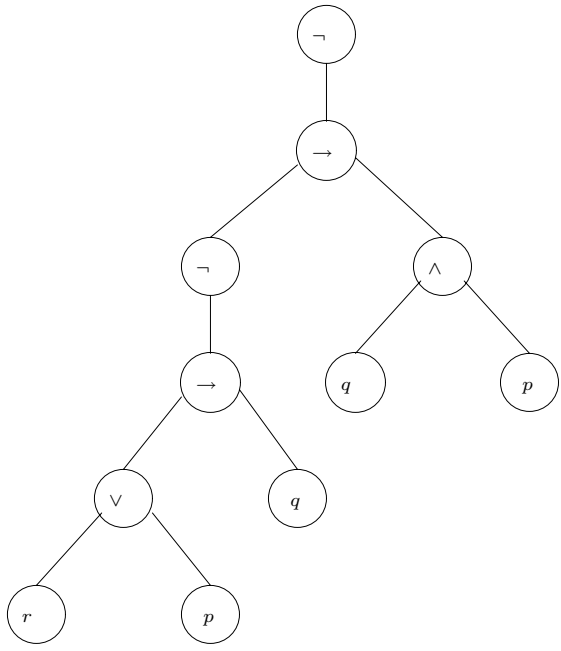


Figure 1.23. Another parse tree of a negated implication.

* (c) $p \vee (\neg(q \wedge (r \to q)))$
(d) $(p \wedge q) \to (p \vee q)$
(e) $((p \to \neg q) \to \neg p) \to q$
(f) $(p \to q) \vee (p \to \neg q)$
(g) $((p \to q) \to p) \to p$
(h) $((p \vee q) \to r) \to ((p \to r) \vee (q \to r))$
(i) $(p \to q) \to (\neg p \to \neg q)$.

3. Given a valuation and a parsetree of a formula, compute the truth value of the formula for that valuation (as done in a bottom-up fashion in Figure 1.7 on page 40) with the parse tree in
* (a) Figure 1.10 on page 44 and the valuation in which q and r evaluate to `T` and p to `F`;
(b) Figure 1.4 on page 36 and the valuation in which q evaluates to `T` and p and r evaluate to `F`;
(c) Figure 1.23 where we let p be `T`, q be `F` and r be `T`; and
(d) Figure 1.23 where we let p be `F`, q be `T` and r be `F`.
4. Compute the truth value on the formula's parse tree, or specify the corresponding line of a truth table where
* (a) p evaluates to `F`, q to `T` and the formula is $p \to (\neg q \vee (q \to p))$
* (b) the formula is $\neg((\neg q \wedge (p \to r)) \wedge (r \to q))$, p evaluates to `F`, q to `T` and r evaluates to `T`.