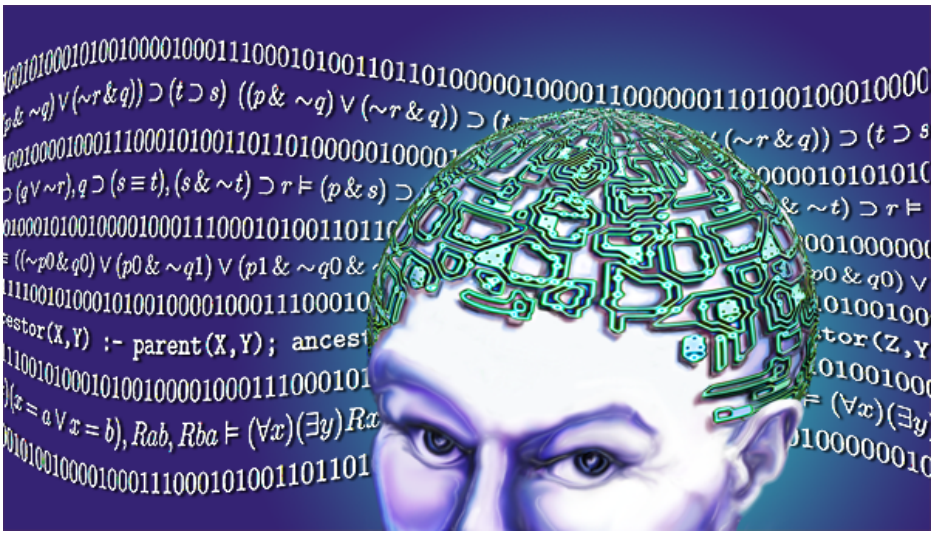


LOGIC
Language & Information
VOLUME I

Jen Davoren and Greg Restall

The University of Melbourne

davoren@unimelb.edu.au · restall@unimelb.edu.au

VERSION OF MARCH 1, 2015

CONTENTS

INTRODUCTION

These are the notes for use in *Logic: Language and Computation 1*, an introduction to propositional logic and its applications, available on Coursera from March 2015. The course comes out of 10 years of experience teaching introductory logic at the University of Melbourne, to students in Philosophy, Engineering, Mathematics, Computer Science, Linguistics and other disciplines. We have learned a lot from our students, and our colleagues who initially developed the course with us: Steven Bird, the late Greg Hjorth and Lesley Stirling. With the experience of teaching logic to students at Melbourne over many years, we are excited to bring this work to a large and diverse international audience.

This text has a generous margin, for you to make your own notes, and for us to occasionally add comments of our own on the way through. (If truth tables or proofs are too large to fit in the main text, you might find them poking out into the margin, too.)

Greg Hjorth, who died tragically in 2011, was not only an accomplished mathematician. He was also a International Master in Chess, and joint Commonwealth Chess Champion in 1983.

WHAT THIS COURSE IS ABOUT: Information is everywhere: in our words and our world, our thoughts and our theories, our devices and our databases. Logic is the study of that information: the features it has, how it's represented, and how we can manipulate it. *Learning* logic helps you formulate and answer many different questions about information:

- Does this hypothesis clash with the evidence we have or is it consistent with the evidence?
- Is this argument watertight, or do we need to add more to make the conclusion to really follow from the premises?
- Do these two sentences say the same things in different ways, or do they say something subtly different?
- Does this information follow from what's in this database, and what procedure could we use to get the answer quickly?
- Is there a more cost-effective design for this digital circuit? And how can we specify what the circuit is meant to do so we could check that this design does what we want?

These are questions about *Logic*. When you learn logic you'll learn to recognise patterns of information and the way it can be represented. These skills are used whether we're dealing with theories, databases, digital circuits, meaning in language, or mathematical reasoning, and they will be used in the future in ways we haven't yet imagined. Learning logic is a central part of learning to think well, and this course will help you learn logic and how you can apply it.

If you take this subject, you will learn how to use the core tools in logic: the idea of a formal language, which gives us a way to talk about

logical structure; and we'll introduce and explain the central logical concepts such as consistency and validity; models; and proofs. But you won't only learn concepts and tools. We will also explore how these techniques connect with issues in *linguistics*, *computer science*, *electronic engineering*, and *philosophy*.

The class is taught over a period of five weeks. In the first two weeks, we learn *core propositional logic*.

- WEEK 1—*The Syntax of Propositional Logic*; *Truth Tables*; *Classifying Propositions*.
- WEEK 2—*Relationships between Propositions*; *Tree Proofs*; *Soundness and Completeness*

Then in the second part of the subject, we cover different *applications of propositional logic*. In the subject, we expect students to take at least two. You can take them in any order. They depend on the background from core logic, but you do not need to have covered any particular application area in order to start another one. Explore these in whatever order you like.

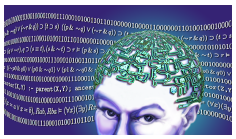
- ELECTRONIC ENGINEERING—*simplifying digital circuits*
- PHILOSOPHY—*vagueness* and *borderline cases*
- COMPUTER SCIENCE—*databases*, *resolution* and *propositional Prolog*.
- LINGUISTICS—*meaning*: *implication* vs *implicature*

This subject presumes no specialist background knowledge. You just need to be able to read and write, and be prepared to think, work and learn new skills—in particular, a willingness to work with symbolic representation and reasoning.

HOW TO USE THESE NOTES: These notes provide you with a written complement to the course videos and discussion boards available on `Coursera`. Most students will find it easiest to watch the videos, and then refer to the notes to follow up things in greater depth as necessary. But if you learn best by reading, start with the notes, and then find if you find something needs more explanation, try the video that corresponds to the material in the chapter you're working through. These are presented in the same order on `Coursera` as they are in these notes.

OTHER RESOURCES: We use Greg's textbook *Logic* [12] at the University of Melbourne. It's not necessary for this subject, but if you want to cover things in more depth, this is the text which is the closest match to the subject we teach. There are some other texts which cover some aspects of logic in a way close to this course. Colin Howson's *Logic with Trees* [7] is a good introduction to the tree technique of proofs, taught in a clear and accessible fashion.

ACKNOWLEDGEMENTS: In addition to our fellow lecturers mentioned above who initially developed 800-123—LOGIC: LANGUAGE AND INFORMATION (and then UNIB10002), and who have taught it with us—Steven Bird, Greg Hjorth and Lesley Stirling—we thank the other lecturers who have joined us in later years; Brett Baker, Leigh Humphries, Dave Ripley and Peter Schachte, and the tutors who have worked with us: Conrad Asmus, Rohan French, Ben Horsfall, Nada Kale, David Prior, Raj Dahya, Sandy Boucher, Toby Meadows, Sylvia Mackie. We have learned a great deal in teaching logic with such a wide range of capable and enthusiastic colleagues.



We thank the artist Xi for developing our logo.

We thank Professor Alex Simpson from The University of Edinburgh for the idea of weather examples in propositional PROLOG, which were adapted from some examples used in his lecture course Logic Programming.

We thank our colleagues in the Learning Environments team at the University of Melbourne, who helped us develop the digital resources for the subject—especially Eileen Wall, who tirelessly filmed and processed the hours of video over months of work, and Peter Mellow and Susan Batur who have advice and feedback on course design and helped us with social media, and Astrid Bovell and Wilfrid Villareal who helped us negotiate the complex world of copyright restrictions.

We hope you enjoy this introduction to logic as much as we've enjoyed producing it. If you have comments or feedback for us, or questions about the text, post comments in the Coursera discussion boards.

Jen Davoren & Greg Restall

Melbourne,
March 2015

PART 1

Core Techniques in Propositional Logic

LANGUAGE AND MODELS OF PROPOSITIONAL LOGIC

1

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

These are the ideas and concepts we'll learn in this chapter of the notes.

Use these summary sections at the start of each chapter to keep track of what you're learning. We have quizzes and discussion forums online for you to practice your skills and get familiar with each of these concepts.

- *Propositions*: these are the fundamental units of information. They are the kinds of things we can assert or deny, agree or disagree, believe or not believe.
- *Propositional connectives*: These are the logical concepts AND (&), OR ($\vee$), NOT ($\sim$), ONLY IF ($\supset$), IF AND ONLY IF ($\equiv$).
- We will define the *well-formed formulas* of *propositional logic*.
- We will introduce the tools of *truth valuations* and *truth tables*. You'll get familiar with these rules–A & B is true iff both A and B are true; A $\vee$ B is true iff at least one of A and B are true, $\sim$A is true iff A is not true; A $\supset$ B is false iff A is true and B is false; A $\equiv$ B is true iff A and B have the same truth value.
- We will classify formulas according to their logical behaviour. You'll be able to recognise *tautologies*, *contingencies*, and *contradictions*.
- We will classify the way different formulas can relate to each other. You'll understand the relations of *logical consequence*, *logical equivalence*, being *contradictories*, *contraries* and *sub-contraries*.

SKILLS

These are the skills you will learn to demonstrate.

- Distinguishing sentences which express propositions from those that don't.
- Identify well-formed formulas, and the main connective of a formula.
- Translate natural language sentences into formulas of propositional logic; esp. varieties of implication: IF, ONLY IF, UNLESS, etc.
- Use truth tables to determine whether a formula is a tautology, a contradiction, a contingency.
- Use truth tables to determine whether or not formulas are related by logical consequence or logical equivalence, or whether they are contradictory, contraries or sub-contraries.

There are many way you can think about logic. Some people think of logic as a whole lot of tips and tricks for thinking well. Other people think of logic is the body of the most general truths about things. Still others think of logic as the laws of thought. Each of these ideas provide different ways into topic. The way *we* understand the topic of logic is a different from all of those ideas. When you look at the history of logic—especially the kind of logic that has flowered in the great works of the 20th and the beginning of the 21st Century, you see a lot of work about *proofs*, about *models*, about *sentences*, the way they *relate to each other*, how they can be *derived* or *refuted*, or how they can be *interpreted*. This work in logic is applied in many different areas, like computer science (in the study of databases, and in the interpretation of programming languages and algorithms), in electronic engineering (in the study of digital circuits), in linguistics (in the study of the meanings of words and sentences), in mathematics (in the study of proofs and models for different mathematical theories), and in philosophy (in the development of fundamental theories about existence, or possibility and necessity, and much more besides).

In all of this, logic pays attention to *language* and *what we can express in our language*. Logic is especially centred on the way the claims we make relate to each other. Logicians are concerned with notions like these:

- CONSISTENCY

  Can these claims be true together? Or are they INCONSISTENT—in that they *clash* and can not be all true.

- CONSEQUENCE

  Do these premises have this statement as a consequence? If we are committed to the premises of this reasoning, are we (at least implicitly) also committed to the conclusion? Or is there some way we could grant the premises but avoid that conclusion?

- EQUIVALENCE

  When do different sentences amount to saying the same thing—while expressing it in different ways? How can you show that two sentences *don't* say the same thing? Is the only way to find a circumstance where one could hold without the other one also holding?

Questions like these can all addressed using the tools from logic: tools that we'll learn in this subject. Let's start our study of logic by looking more at the kinds of sentences which play a role in consistencey, consequence and equivalence.

1.1 | LANGUAGE OF PROPOSITIONAL LOGIC

In this section we will introduce the language of propositional logic.

1.1.1 | PROPOSITIONS

A sentence expresses a PROPOSITION iff it makes a statement about how things are. Propositions are the kinds of things that can be *true* or *false*. You can *agree with* or *disagree with*, etc.

This yellow box offers a *definition*. Pay special attention to yellow boxes when revising, because these are the definitions of key concepts.

So, propositions are important because they're the units that feature when we consider consistency, consequence and equivalence.

We use *orange boxes* for examples. These illustrate the definitions.

These are sentences that express propositions:

- We have students from all over the world.
- There is no difference between the mind and the brain.
- $3 + 5 < 4 + 3$.

You can see that these sentences are the kinds of things that are either *true* or *false*. That we can *agree with* or *disagree with*. We could use them as a *premise* in our reasoning, or as a *conclusion*. I agree that we do have students from all over the world. I have read arguments which purport to show that there is no difference between the mind and the brain, and I have read arguments that purport to show that the mind is not the brain. Both conclusions would show that "There is no difference between the mind and the brain" does express a proposition, even if we don't agree on whether it's true or not. Finally "$3 + 5 < 4 + 3$" expresses a proposition, even if it's an obviously false one, since 8 is larger than 7, not smaller.

The 'I' here is Greg, but sometimes the 'I' in these notes is Jen, or both of us simultaneousy.

It should be clear that not all sentences express propositions. There are many things we can do with sentences besides staking our claim on what is true.

Some people complain about the clause that propositions should "make a statement about how things are." But false propositions are still propositions, and statements about abstract and invisible things (like numbers) are still about this world,

These are sentences that *don't* express propositions.

- Please don't watch these lectures while driving a car.
- What is the capital of Ethiopia?
- Thank goodness; the poor cat!
- Hi, Jen!

The first sentence here makes a *request*. It is an *imperative*. The request cannot be *true* or *false*. It can be wise or foolish to make the request, and it can be complied with or flouted, and you can agree to follow it, or object to it. But it cannot be the kind of thing that is used as a premise or a conclusion in an argument. The second sentence asks a

Even though we take propositions to be the central focus of logic, Nuel Belnap's "Declaratives are not enough" [5] reminds us that imperatives, greetings, and other speech acts are as important for logic and philosophy as declaratives, the sentences that express propositions.

question. That is a special kind of request: a request for information. Again, that's not the kind of thing that can be true or false. The third sentence is an expression of gratitude or surprise. The last is a *greeting*. These, also, cannot be true or false.

Here are guidelines for distinguishing sentences that express propositions from those that don't. A PROPOSITION is the kind of thing you could:

- believe or disbelieve;
- agree or disagree with;
- assert or deny;
- give a reason for or against;
- use as a reason for or against something else; and
- express with a *declarative sentence*.

These are the examples we also use in the video lectures to test your knowledge. We go into the answers in more depth here below, in the green box. Don't read the answers immediately. Try the questions for yourself to test your own understanding of the material. Head to the `Coursera` discussion boards if you have questions about these answers.

Here is an opportunity for you to test your understanding:

FOR YOU: Which of these sentences express propositions?

(a) Greg lives in Melbourne.
(b) What do you think of logic so far?
(c) I wish I had learned logic before!
(d) It's immoral to consume meat.
(e) This last sentence on the list is not true.

ANSWERS:

(a) *Yes*. This expresses a (true) proposition.
(b) *No*. This asks a question.
(c) *Yes*. This can be either true or false. It expresses a sentiment—a wish, but it expresses the sentiment by way of making a statement. One way you can tell that this expresses a proposition is to see that you could use this sentence to insincere. You could say those words when in fact you do not wish you had learned logic before.
(d) *Yes*. This expresses a proposition. It's the kind of thing we can give reasons for or against, and agree or disagree with.
(e) This is *controversial*—it is paradoxical. Some people think that paradoxical statements are defective, and don't express propositions. Others think that despite being paradoxical, this does express a proposition.

1.1.2 | CONNECTIVES

Connectives are like *glue*: we use connectives to combine different propositions.They can combine any different kinds of propositions, into a new proposition containing the smaller statements as parts.

If it's raining *and* I don't have my umbrella,
then I'll get wet *unless* I'm lucky.

Here is another sentence, made up of different propositions, which uses the same connectives, in the same structure as the first sentence.

If I strike the match *and* it isn't wet,
then it will light *unless* it's in a vacuum.

And here is another.

If $n > 5$ *and* m isn't even,
then $f(m, n) > 5$ *unless* $m = n$.

As you can see, we can use these words like 'if' and 'and' and 'not' and 'unless' to join statements about the weather, matches, or mathematics. Connectives join propositions to make new propositions, *whatever* those propositions are about, without adding to or changing the topic under discussion. In this section, we'll look at the connectives we'll examine in our introduction to logic.

CONJUNCTION Consider these two propositions:

It's raining. I don't have my umbrella.

We can assert both of them at once by asserting their *conjunction*:

It's raining *and* I don't have my umbrella.

For propositions p and q, their CONJUNCTION is the proposition which we'll write as "p & q."

p, q are the CONJUNCTS of the conjunction p & q.

In some other logic texts, you'll see a conjunction written with a wedge "$\wedge$." We use the ampersand.

Not all uses of the word "*and*" form conjunctions. Look at the following propositions. We'll see how some of them form conjunctions, and some do not.

- *Clancy is tired and Madison has a headache.*

  — Yes, this is a conjunction. The conjuncts are simple: "Clancy is tired" and "Madison has a headache"

- *I learned logic and I never regretted it.*

  — Yes, this is a conjunction. The first conjunct is "I learned logic." The second is best not expressed as "I never regretted it," because when we take this clause out of the sentence, it's not so clear what it means. Better to express it as "I never regretted learning logic."

- *Lee and Darcy are lovers.*

— This is *not* a straightforward conjunction. If by the sentence you mean that Lee and Darcy love each other, then this is not the conjunction of "Lee is a lover" and "Darcy is a lover." So, it doesn't have the same structure as "Lee and Darcy are logicians", which is the conjunction of "Lee is a logician" and "Darcy is a logician."

- *I went out and had dinner.*

  — This is also not a straightforward conjunction: at least, it is not the conjunction of "I went out" and "I had dinner" if it says that I had dinner *after* I went out (or better, *while* I was out).

- *One false move and I shoot.*

  — No. This is not a conjunction. It expresses the claim that *if* you make one false move, I shoot. This is a *conditional*, not a conjunction.

Hang on for a page—we'll get to conditionals soon.

FOR YOU: Which of the following propositions are conjunctions? Of those that are, what are their conjuncts?

(a) I'm taking *Logic 1* and *Logic 2*.
(b) Clancy and Madison got married.
(c) Mangoes and bananas are nutritious.
(d) *Doctor Who* and *Sherlock Holmes* have never been on the same TV program.
(e) *Doctor Who* and *James Bond* have been played by many different actors.

ANSWER:

(a) Yes: "I'm taking *Logic 1*" and "I'm taking *Logic 2*".
(b) No: not if it means that Clancy and Madison got married to each other.
(c) Yes: "Mangoes are nutritious" and "Bananas are nutritious."
(d) No: this means that they've never been on a TV program *with each other*.
(e) Yes: "*Doctor Who* has been played by many different actors" and "*James Bond* has been played by many different actors."

Conjunction is not the only way to combine two statements. We can also use *disjunction*.

DISJUNCTION We can assert that *at least one* of the two propositions are true by asserting their *disjunction*.

I'll get wet, *or* I'm lucky.

Disjunctions can be interpreted as *inclusive*:

I'll get wet, *or* I'm lucky (and maybe both).

or *exclusive*:

I'll get wet, *or* I'm lucky (and not both).

Often, when asserting a disjunction, we do not make clear whether we mean to include the case that both disjuncts are true, or whether we mean to exclude it. Ther are many interesting issues on how we interpret disjunctions in natural language. In this text, we will focus on inclusive disjunction as it is easier to work with in many ways; in particular, it has a *dual* relationship with conjunction.

As we will see in the next section when we study truth tables: a conjunction is *true* when and only when both conjuncts are true. An inclusive disjunction is *false* when and only when both disjuncts are false. This is the duality we mean.

For propositions p and q, their INCLUSIVE DISJUNCTION is the proposition which we'll write as "$p \vee q$."

p and q are called the DISJUNCTS of the disjunction.

The symbol "$\vee$" is from *"vel,"* the Latin word for *"or."*

"Unless" acts like a disjunction too.

I'll get wet, unless I'm lucky.

This says either I get wet or I am lucky. If neither holds, then the statement is false. If either holds, the statement is true.

It's one thing for "unless" to be a disjunction. Should "unless" be treated as inclusive or exclusive? What do you think?

I'll go to the party unless Clancy is there.

This says that either I go to the party, or Clancy is there. If Clancy is not at the party, you'd be right to expect me to be there, because I've said that I'll go unless Clancy is there.

CONDITIONALS We can express relations *between* propositions by way of conditionals. Here is an example:

If it rains then I'll get wet.

This connects the propositions "it rains" and "I'll get wet". It does not say that both are true (it is not a conjunction), and it does not say that either are true. The statement would be true on a sunny day when I am outside without an umbrella. It expresses a conditional connection between the two propositions: their truth is connected in that if the first ("it rains") is true then so is the second ("I'll get wet").

The one connection expressed in a conditional can be expressed in many different forms of words.

- If p then q
- If p, q
- q if p

- p only if q
- p implies q

In each of these cases, we say that p is the *antecedent*, and q is the *consequent*. And in each case, the statement tells us that if p is true, so is q. In each case, the statement is refuted if p is true and q is not. Notice that in "q if p" the consequent comes first. The "only" in "p only if q" flips the order in the "if" back to the antecedent first and consequent second.

It's important to understand that in a conditional, we are not making a claim about whether the antecedent *causes* the consequent. When I say "if it rains then I'll get wet" then the antecedent is "it rains" and the consequent is "I'll get wet," and here, typically, the rain causes me to get wet. However, I could equally say "I'll get wet only if it rains." In this case, the "I'll get wet" is the *antecedent* while "it rains" is the *consequent*. In this case, my getting wet doesn't *cause* the rain. (No, the rain—if it rains—causes me to get wet, not *vice versa*.) But if we know that the conditional is true, we can *learn* that it has rained when we learn that I have got wet. The connection expressed in the conditional is a connection in the truth or falsity of the antecedent and the consequent. It is *not* claiming that the antecedent causes the consequent.

For propositions p and q, their CONDITIONAL "if p then q" is the proposition which we'll write as "$p \supset q$."

p is the ANTECEDENT and q is the CONSEQUENT of the conditional.

A handy test is to ask: What would convince you that this statement is *false*? A conditional is false when the antecedent is true and the consequent is false.

It's an important skill to recognise conditionals, and in particular, to be able to recognise the *antecedent* and the *consequent*. Confusing the antecedent and the consequent of conditionals is a source of many logical mistakes.

FOR YOU: Which of these are conditionals? For those that are conditionals, identify the antecedent and the consequent.

(a) If I don't do the practice exercises, I won't learn.

(b) I'll learn if I do the practice exercises.

(c) I'll get wet if it rains and I don't have an umbrella.

(d) I'll get wet only if it rains and I don't have an umbrella.

(e) The report implies that 2 million jobs may be lost.

ANSWER:

(a) Yes—*Antecedent*, I don't do the practice exercises. *Consequent*, I won't learn.

(b) Yes—*Antecedent*, I do the practice exercises. *Consequent*, I'll learn.

(c) Ambiguous—It could be the conditional with *antecedent* it rains and I don't have my umbrella, and *consequent* I'll get wet. Or, it could be the *conjunction* whose first conjunct is the conditional "I'll get wet if it rains." How can you tell which it is? Ask yourself the question: does it follow from the original statement that I don't have an umbrella? If so, it's a conjunction. If not, it's a conditional.

(d) Ambiguous—It could be the conditional with *antecedent* I'll get wet, and *consequent* it rains and I don't have my umbrella. Or, it could be the *conjunction* whose first conjunct is the conditional "I'll get wet only if it rains." How can you tell which it is? The same test as with (c).

(e) No—it is a statement of implication, but the antecedent is "the report" and this phrase is not itself a sentence expressing a proposition. The report itself might express a (very long) proposition, but that is not the antecedent in this statement.

FOR YOU: Which of these two conditionals has "you'll do well" as the antecedent, and "you study hard" as the consequent?

(a) You'll do well if you study hard.

(b) You'll do well only if you study hard.

(c) If you study hard, you'll do well.

ANSWER: Only (b). In (a) and (c) "you study hard" is the *antecedent* and "you'll do well" is the *consequent*.

BI-CONDITIONALS In philosophy, mathematics and other technical domains, you will often see a form of words that expresses a two-way connection between propositions. This form of words is the *bi-conditional*:

I'll get wet if and only if it's raining.

The BI-CONDITIONAL of p and q is $p \equiv q$.
p and q are the *left-* and *right-hand expressions* of the bi-conditional.

The words "if and only if" are often abbreviated "iff."

NEGATION The basic way of *denying* a proposition is to assert its *negation*:

Smoking is not always bad for your health.

This is the negation of "smoking *is* always bad for yourself."

If p is a proposition, its NEGATION is ~p.
p is said to be the *negand* of ~p.

Considr the following Which of the following propositions are negations, and what are their negands?

- It is not the case that I've won the London Marathon.

— This is the negation of "I *have* won the London Marathon."

- That outfit won't cost you $250.

— This is the negation of "that outfit will cost you $250."

- Smoking is never bad for your health.

— This is the negation of "Smoking is sometimes bad for your health."

- I am not a crook.*This is the famous denial from Richard Nixon, in the wake of the Watergate scandal.*

— This is a negation, and it's the negation of "I am a crook."

That's enough for introducing negation and some of its behaviours. No it's time for you.

FOR YOU: Which of the following propositions are negations, and what are their negands?

(a) I won't go to the party.
(b) I won't go to the party if Clancy is there.
(c) You mustn't think that logic is only for clever people.
(d) The fridge didn't get cleaned, unless you cleaned it.

ANSWER:

(a) *Yes*—the negation of "I will go to the party."

(b) *No*—not a negation. It's a conditional, with the negation "I won't go to the party" in the consequent.

(c) *No*—this is *not* negation of "you must think that logic is only for clever people." (If it's purely optional for you to think that logic is only for clever people, then both "you must think that logic is only for clever people" and "you must not think that logic is only for clever people" are false. "Must not" is not the same as "it's not the case that you must".)

(d) *No*—the disjunction of a negation (the negation of "the fridge did get cleaned") with the proposition "you cleaned the fridge."

OTHER CONNECTIVES There are plenty of *other* ways to make propositions out of other propositions. For example, if we take any of the following sentence fragments

- Clancy believes that …
- It is possible that …
- It is necessary that …
- … because …

and replace each ellpsis by a sentence expressing a proposition, then the result is also sentence expressing a proposition. These phrase fragments do the job of connectives or operators in the same sort of way that other connectives do.

An 'ellipsis' is the three dots we use to indicate that something has left out of text.

We won't look at these sorts of connectives in this subject, some of these and *more* are also studied by logicians, but examining them would take us too far away from where we are going in this subject.

Well, they have the same sort of *grammar* as the traditional connectives. However, they are logically much more complex. *Epistemic logic* is the logic of knowledge and belief (the logical behaviour of statements like 'x believes that p' are studied in epistemic logic); and *modal logic* is the logic of possibility and necessity. Both are active areas of study in logic at this time.

1.1.3 | PROPOSITIONAL FORMULAS

So, we have a family of connectives (conjunction, disjunction, the conditional and the bi-conditional), and an operator (negation). Here is a summary of the notation and vocabulary we will use.

ENGLISH WORDS	CONNECTIVE	SYMBOL
… and …	*conjunction*	$\&$
… or …	*disjunction*	$\vee$
if …, then …	*conditional*	$\supset$
… iff …	*bi-conditional*	$\equiv$
not …	*negation*	$\sim$

We will use these symbols to define a *formal language* to represent the structures. You can think of a formal language in two ways. (1) A formal language is a special little language of its own. (2) It's a way of represting the *form* or *shape* of sentences in our everyday languages. Both of these are appropriate ways of understanding a formal language. The formal language is made up of *formulas*. A formula in a formal language is like a sentence in an everyday language. A special feature of formal languages is that we provide precis rules for when something is a formula in the language. There are no judgement calls or borderline cases. Either something is clearly in the language (it's a 'well-formed formula') or it's clearly not (it's ungrammatical). How do we manage this? By giving a *precise grammar* for the formal language.

We define the *formulas* of our language by starting with a collection of *atoms*. We'll use some collection of symbols as the starting point of a language for propositional logic.

An 'atom' is something that is indivisible. (*a-tom*—Greek for "cannot be divided"). Now, just like everyday atoms can be divided (they're made of protons, neutrons, electrons), our atoms perhaps can be divided too, but the important feature of atoms is that they have no *formulas* inside them.

DEFINITION: A collection of *atoms* is some collection of symbols we'll use to build up formulas. We'll often use letters like $\mathtt{p}$, $\mathtt{q}$, $\mathtt{r}$, $\mathtt{s}$, $\mathtt{x}$, $\mathtt{y}$, $\mathtt{r}_3$, $\mathtt{s}_{15}$, $\mathtt{p1}$, $\mathtt{p0}$, ... as atoms.

Once we have our atoms, our building blocks, we then use them to build up a language of formulas.

DEFINITION: The language of propositional logic (relative to a choice of a collection of atoms) is defined as follows:

(1) *Atoms* are *formulas*.

(2) If $\mathtt{A}$ is a *formula*,
then so is $\sim\mathtt{A}$.

(3) If $\mathtt{A}$ and $\mathtt{B}$ are *formulas*,
then so are $(\mathtt{A} \,\&\, \mathtt{B})$, $(\mathtt{A} \vee \mathtt{B})$, $(\mathtt{A} \supset \mathtt{B})$ and $(\mathtt{A} \equiv \mathtt{B})$.

($\star$) *Nothing else* is a *formula*.

Another way to compactly specify the 'grammar' of this formal to use the following notation

This is BNF, or "Backus Naur Form," first used in the specification of formal computer programming languages such as ALGOL. http://cui.unige.ch/db-research/Enseignement/analyseinfo/AboutBNF.html

$$\text{FORMULA} \;:=\; \text{ATOM} \mid {\sim}\text{FORMULA} \mid (\text{FORMULA} \,\&\, \text{FORMULA}) \mid (\text{FORMULA} \vee \text{FORMULA}) \mid (\text{FORMULA} \supset \text{FORMULA}) \mid (\text{FORMULA} \equiv \text{FORMULA})$$

This *grammar* gives us rules to follow to generate formulas out of atoms. For example, if $\mathtt{p}$, $\mathtt{q}$ and $\mathtt{r}$ are atoms, we can show that $(\sim\mathtt{p} \supset (\mathtt{q} \,\&\, \mathtt{r}))$ is a formula too. Here is the construction process:

- $\mathtt{p}$, $\mathtt{q}$ and $\mathtt{r}$ are atoms, so by condition (1), they are formulas, too.

- Since p is a formula, by condition (2), ~p is a formula.
- Since q and r are formulas, by condition (3), (q & r) is a formula.
- Since ~p and (q & r) are formulas, by condition (3), (~p ⊃ (q & r)) is a formula, too.

These are formulas too. Can you see why?

- ((p & q) ⊃ r)
- ~~((p ∨ ~q) ≡ ~r)
- (((p ⊃ q) & (q ⊃ r)) ∨ (p ⊃ r))

It is not too difficult to go through the construction process for each of these formulas, starting at the atoms (p, q and r) and building up the constituents piece by piece. Notice that there is no problem in applying *repeated* negation symbols.

By condition (2), if p is a formula so is ~p. By condition (2), if ~p is a formula (and it is), so is ~~p. By condition (2), if ~~p is a formula (and it is), then so is ~~~p. By condition (2), if ~~~p is a formula, so is ~~~~p. And so on, *ad infinitum*.

Whereas, these strings of symbols are not formulas.

- p~
- (p & q ∨ r)
- p ≡ (q ⊃ ~r)

Can you see why? p~ is not a formula because whenever we add a '~' to a formula we use condition (2), and the '~' occurs *before* a formula, not after it. So p~ is not a formula.

(p & q∨r) is not a formula because whenever we add '&' or '∨' to a formula we use condition (3), and according to this condition, whenever we add either an '&' or an '∨' then we add two parentheses (one left, one right). So a formula with one '&' and one '∨,' should contain four parentheses. But '(p & q ∨ r)' is missing a pair of parentheses.

In this case, the missing parentheses really do make a difference. If we were to add them back in, we have two options: (p & (q ∨ r)), a conjunction of p with q ∨ r or it is ((p & q) ∨ r), which is a disjunction of (p & q) with r. These mean very different things.

So, too, is 'p ≡ (q ⊃ ~r).' It only has two parentheses, when it should have four. But in this case it is is the outermost pair that is missing there is no ambiguity in interpreting this string of symbols. Adding the parentheses, we get '(p ≡ (q ⊃ ~r)),' a bi-conditional with 'p' as LHE and '(q ⊃ ~r)' as RHE, a conditional with antecedent q and consequent ~r. We could remove the outermost parentheses of each formula with no loss or confusion. From now on we'll leave out writing down outermost parentheses when this is convenient. (The parentheses are *there*, but we don't always write them down.)

Why put the outermost parentheses in at all? It's because of when we add the formula inside another formula that we need the parentheses. 'p ∨ q' is not ambiguous by itself, but when we conjoin it to r as p ∨ q & r we have ambiguity.

Now, let's consider an exercise to check how you're going with formulas.

FOR YOU: Consider a tiny language which contains only *two* ATOMS—p and q. And suppose we form all the formulas of our baby language just using these *two* atoms.

Which of these following are formulas of this little language?

- $\supset p$
- $((p \vee q) \supset q)$
- $(p \vee r)$

How many formulas in this baby language contain just 2 symbols? How about 3? Or 4? Or more?

ANSWER: $\supset p$ is not a formula, but $((p \vee q) \supset q)$ is a formula. $(p \vee r)$ is not a formula because it contains the component "r" which is not one of the atoms in the language we have specified.

There are only 2 formulas with 2 symbols: $\sim p$ and $\sim q$. There are also 2 formulas with 3 symbols: $\sim\sim p$ and $\sim\sim q$. (We don't count $p \,\&\, q$ as a formula with three symbols. It has *five*, even though we didn't write down the outermost parentheses.) Indeed, there are only 2 formulas with 4 symbols. $\sim\sim\sim p$ and $\sim\sim\sim q$. For 5 symbols there are $2 + (4 \times 4) = 18$ formulas: $\sim\sim\sim\sim p$ and $\sim\sim\sim\sim q$, together with $(p \,\&\, p)$, $(p \,\&\, q)$, $(q \,\&\, p)$, and $(q \,\&\, p)$ for conjunction, and similarly, disjunction, the conditional and the bi-conditional.

1.1.4 | FORMALISATION

If we want to use logic to assess sentences of natural language we need to *translate* from sentences of English (or any other natural language) to formulas in our propositional language. Another way of understanding this is that we are *identifying* the form of the sentence in natural language. Doing this helps us identify logical structures present in our statements. This is a step along the way of finding *forms* our propositions have.

Formalising a sentence is a kind of *focussing*. You attend to *some* of the features of the sentence (the logical connectives and operators, and the way they combine things) and you ignore others (the particular *content*). This isn't to say that the content isn't important, of course. But in formal logic it's important to focus on the way that propositions are structured and combined.

This is *formalisation*. There are three steps you need to go through when you're finding a logical form:

1. *Identify the Connectives*.
2. *Create a Dictionary*.
3. *Find the Form*.

Let's look at a few examples.

EXAMPLE 1: Formalise this sentence:

Morgan can't do advanced logic
unless she's mastered connectives.

Let's go through the three stages, one by one.

1. What connectives can you find in this sentence? "Unless" is a disjunction. There's also another operator hiding there: the "n't" in can't is a negation. "Morgan *can't* do advanced logic" is the negation of "Morgan *can* do advanced logic."

2. Now we make a dictionary. We look for those propositions that are a part of this statement, which have no logical connectives of their own. "Morgan can't do advanced logic" is the negation of "Morgan can do advanced logic" and this statement has no logical connectives, so we'll treat that as an *atom*. Let's choose the letter l to stand for that proposition.

   The other atom inside our proposition is expressed by the phrase "she's mastered connectives." In this context in the sentence, it's clear what the "she" means: it means *Morgan*. Given that we're going to take this phrase "she's mastered connectives" out of context to consider it by itself, we'll replace this pronoun by the name it points back to. So we'll take "Morgan has mastered connectives" as our other atomic proposition, and let's choose another letter, say c, for this proposition.

   This gives us the following *dictionary*, pairing up atomic *formulas* (the l and the c) with propositions.

l	=	Morgan can do advanced logic
c	=	Morgan has mastered connectives

3. Given the dictionary, we can proceed to find the form of our sentence. We've seen that "Morgan can't do advanced logic" is the negation of "Morgan can do advanced logic," so

   Morgan can't do advanced logic *is* ~l

   Then the "unless" is a disjunction. In general, "A unless B" has the logical form of the disjunction of A and B. If we say that Morgan can't do advanced logic unless he's mastered connectives, we're saying that either Morgan can't do advanced logic, or Morgan has mastered connectives. So our logical form is:

   $$(\sim l \vee c)$$

   the disjunction of ~l with c. As we've already said, we can leave out the outer parentheses if you like, and just write '~l ∨ c.'

Let's look at another example.

It's purely a matter of choice as to what letters you use. We like to use letters that remind us of the proposition being used. We could have chosen m for Morgan, but Morgan features in the other proposition, as does *m*astering connectives. But we could have equally chosen a for "advanced," and that would have worked just as well.

EXAMPLE 2: Formalise this sentence:

If you don't finish *Game of Thrones* then I can't read it.

1. Here there are two negations, in "don't" and "can't," and the central conditional.

2. The atomic propositions then are

f	=	You finish *Game of Thrones*
r	=	I can read *Game of Thrones*

3. Then the formalisation is the conditional with antcedent "you don't finish *Game of Thrones*" (which is formalised as ~f) and whose consequent is "I can't read *Game of Thrones*" (which is formalised as ~r). So the whole formula is then

$$\sim f \supset \sim r$$

Notice, we've left out he outer parentheses here. We won't mention this any more.

FOR YOU: Which of the following three sentences has this form: $s \supset (l \vee w)$?

1. If you strike this match it will light, unless it is wet.
2. If you strike this match, either it will light or it's wet.
3. This match will light if you strike it, unless it is wet.

where we use the following dictionary DICTIONARY:

- s = you strike this match.
- l = this match will light.
- w = this match is wet.

ANSWER: 2. Is the correct answer. 1. and 3. both have the form $(s \supset l) \vee w$. In both cases, the comma before the 'unless' indicates that the clause before is one disjunct of the disjunction. In both cases, that clause has the form $(s \supset l)$.

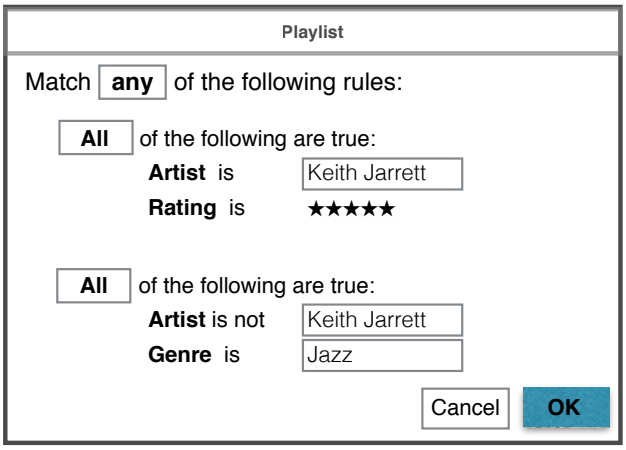
This is the process of finding logical form of sentences in a language like English. But finding logical structure of natural language sentences is only one of the many things that formalisation—and formalisation into the language of propositional logic—is good for.

We only have experience with a few languages—and English is our only native language. Are there special issues when it comes to finding logical form in languages other than English. Are there special issues when it comes to formalising and other languages? Let us know in the `Coursera` discussion boards. We'd love to hear from you.

1.1.5 | EXPRESSIVE POWER

Here are some examples of what you can do with the language of propositional logic beyond simple examples of translations of logical form. There's a lot that the language of propositional logic is good for.

DATABASES One place for propositional logic is in our interactions with digital databases. I (Greg) have a music player on my computer. It's a big repository—or database—of the recorded music I own. Sometimes sometimes I use 'smart playlists' to choose tracks to listen to. The smart playlist editor looks something like this:



This playlist selects all of the tracks which are *either* Five-Star rated Keith Jarrett tracks, or other Jazz tracks that aren't by Keith Jarrett. We can *formalise* this playlist in the following way:

Confession: I have a *lot* of Keith Jarrett.

Consider a language with the these atoms, each understood as describing a given track.

- KEITH_JARRETT
- FIVE_STARS
- JAZZ

So, if we are considering a track, KEITH_JARRETT is *true* if it's a Keith Jarrett track, and it's *false* otherwise. If it's rated five starts, then FIVE_STARS is true, and so on. Then the smart playlist collects all the tracks of which *this* formula is true:

$$(\text{KEITH\_JARRETT} \,\&\, \text{FIVE\_STARS}) \vee (\sim\text{KEITH\_JARRETT} \,\&\, \text{JAZZ})$$

Notice, we're moving beyond thinking of propositions as just true or false, and thinking of them as true or false *of* something. This is a tiny move in the direction of 'predicate logic.' We'll focus on predicate logic in the second `Coursera` course, *Logic: Language and Information 2*.

The 'any' and 'all' in the playlist editor do the job of *disjunction* and *conjunction* respectively.

This is a simple example of logical structure found in things other than sentences in a natural language. You'll see many more of these when it comes to databases, digital circuits, maps, and the many other places we store information.

SCOPE AMBIGUITIES Formalisation of sentences in natural language can do much more than just help us focus on logical structure. When you formalise, you'll notice that sometimes things are not easy at all. Natural language is *highly ambiguous*. The one sentence could mean different things. One source of ambiguity is the issue of *scope*, where it is not clear how the parts of the sentence are meant to fit together. The language of propositional logic has no scope ambiguitites at all—it is precisely and rigorously defined. So one of the advantages of formalising sentences in the language of propositional logic is that scope ambiguities can be identified—and hopefully, resolved. Here are some examples of sentences with scope ambiguities.

Another source of ambiguity is *lexical*—the one word could mean different things.

- Clancy quickly ate and went to work.
  - Did Clancy eat quickly and then leisurely make his way to work, or did he both eat and go to work quickly?
- Madison: I'll have fish and chips or salad.
  - Does Madison want fish with either chips or salad on the side ($f \,\&\, (c \vee s)$), or does Madison want either fish and chips, or (instead) salad ($(f \,\&\, c) \vee s$)?
- The trains are on strike today and I need to get to get to the other side of town by 9am unless I'm misremembering things.
  - Does my faulty memory point back to my morning appointment ($s \,\&\, (a \vee m)$)? Am I saying that the trains are on strike today, and that either I have an appointment or I'm mistaken? Or does it point back to both claims ($(s \,\&\, a) \vee m$)? Am I saying that either I'm mistaken, or if I'm not there's a strike and I have my appointment across town?

Scope ambiguities are easier to pinpoint given the precise (and inflexible) language of formulas.

FOR YOU: Select which of these options are possible forms of the sentence "If it rains I'll get wet unless I'm lucky".

1. $r \supset (w \vee l)$
2. $r \supset (w \,\&\, l)$
3. $(r \supset w) \vee l$
4. $(r \supset w) \,\&\, l$

ANSWER: The correct answers are (1) and (3). The other answers, (2) and (4) are incorrect because "unless"

This last example is very subtle. Maybe you can't actually *hear* the difference between these two claims when you consider the English

sentence. Translating into the language of propositional logic helps us get an grip on the scope distinction.

1.2 | MODELS FOR PROPOSITIONAL LOGIC: TRUTH TABLES

So far, we have developed the *syntax* or symbols of propositional logic—by defining a formal language of formulas. In this section we have to examine the *semantics* or *meaning* of propositional logic formulas—by looking at how formulas can be *true* or *false* of formulas depending on whether their components are *true* or *false*.

Why do we focus on truth and falsity? It's because of what propositions *are*. They're the sorts of things that are *true* or *false*, that you can agree with or disagree with, prove or disprove or arue about, etc. There's much *more* to the meaning of statements than their status as true or false—but this status is an important part of meaning.

Since we're going to see 'true' and 'false' a lot, we will use symbols as a shorthand.

This is classical two-valued logic. In the section on Vagueness (Chapter 4 of these notes), we'll look at some reasons why we might want to look *beyond* two truth values. But please, stick with this assumption for now.

> We write 1 for 'true' and 0 for 'false'.

POSSIBLE TRUTH VALUATIONS: Given three atomic propositions p, q and r, each of them is either true or false, so there there are *eight* possible combinations of truth values. Why? First, p is either true or false: there are 2 options, and given each of these options for p we have 2 options for q, so there are $2 \times 2 = 4$ options for p and q. Then for each of these 4 options for p and q there are 2 options for r, so there are $4 \times 2 = 8$ options for p and q and r together. There are 8 different ways things might turn out, when it comes to answering the questions of whether or not p, whether or not q, and whether or not r.

Can you see why it makes sense to write these 8 possibilities as 000, 001, 010, 011, 100, 101, 110, 111? They're the first 8 binary numbers, from 0 to 7.

> We call each of these different choices of 0 and 1 for the atomic propositions a TRUTH VALUATION or an INTERPRETATION.
>
> A truth valuation assigns a value (either 0 or 1) to each atomic proposition.
>
> We sometimes write '$v(p) = 1$' or '$v(q) = 0$' to say that the truth valuation v assigns the value 1 to p, or 0 to q.

In general, given n different atomic propositions, the total number of different possible truth valuations is $2 \times 2 \times \cdots \times 2$ (n times), which is 2^n. The number 2^n grows *very fast indeed* as n gets larger.

There is some confusion caused by the use of nearly identical prefixes for *binary* and *metric* numerical prefixes for very similar quantities. See http://en.wikipedia.org/wiki/Binary_prefix.

n	5	10	20	30	40	50	60
2^n	32	1,024	1,048,576	$> 10^9$	$> 10^{12}$	$> 10^{15}$	$> 10^{18}$
PREFIX		Kilo	Mega	Giga	Tera	Peta	Exa

That's a 1 followed by 18 zeros. That's a *seriously* big number.

If you have 60 atomic propositions, then there are 2^{60}, which is more than 10^{18} different truth valuations for those atomic propositions. There are more than 10^{18} different ways the world might have turned out, just taking these 60 propositions into account.

This is why logic can get tricky. There are lots of possibilities to consider.

1.2.1 | SETTING UP TRUTH TABLES

A *truth table* is a systematic presentation of the different possible truth valuations, for a given set of atomic propositions. If we have the atoms p, q and r, we might draw a table like this

p	q	r		
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

with eight rows, one for each truth valuation for p, q and r. Then in each row, we can represent the truth value of a formula that's built up out of p, q and r. For example, we might have a table like this, for the formula $\sim(p \vee q)$?

Notice, this has only $4 = 2 \times 2$ rows, as there are are only 2 atomic propositions.

p	q	∼	(	p	∨	q	)
0	0	1	0	0	0	0	0
0	1	0	0	1	1	1	1
1	0	0	1	1	1	1	0
1	1	0	1	1	1	1	1

We write the truth value of a complex formula under its main connective. So the truth value of $p \vee q$, the disjunction of p and q is written under the '$\vee$.' We write down a 1 in any row where p or q got the value 1, and that's the second, third and fourth rows. Then the value of the formula $\sim(p \vee q)$ is written under the negation symbol, and this is true in the first row (where $p \vee q$ was false) and is false in the other three rows (where $p \vee q$ was true).

The result is a truth table for the complex formula $\sim(p \vee q)$ which says that it's true if p and q are both false (that's the first row) and it's false otherwise (that's the other three rows).

These values seem like common sense. If you understand what the formula says (it says 'neither p nor q') then indeed, it's true in exactly the row where neither p nor q is true.

So why go through the rigmarole of truth tables when we can simply read and interpret formulas? One reason is that formulas can be much more complex than the simple formula $\sim(p \vee q)$:

$$\sim p \vee ((p \,\&\, \sim(q \supset (r \,\&\, \sim p))) \vee (\sim r \equiv p))$$

When is *this* formula true? When is it false? We would like a systematic and straightforward way to explain how a complex formula depends for its truth value on the truth values of its parts.

1.2.2 | NEGATION, CONJUNCTION AND DISJUNCTION

Here is what we'll do. First we'll get clear on the rules for the connectives. Let's explain how the truth value of a negation, a conjunction or a disjunction depends on the truth values of its constituents.

If a proposition A is true, then its *negation* $\sim$A is false. On the other hand, if A is false, then its *negation* $\sim$A is true.

A	$\sim$A
0	1
1	0

This is the *truth table* for the formula $\sim$A. It shows how the truth or falsity of $\sim$A depends on the truth or falsity of A.

We write a capital letter A rather than the lowercase p, since this works for *every* formula, not just atomic formulas. The A here is a letter that stands for *any* formula. If you like a technical term for this, call it a *schematic formula*.

Conjunctions are true when *both* of the conjuncts are true. And they're *false* the rest of the time.

A	B	(A & B)
0	0	0
0	1	0
1	0	0
1	1	1

That is conjunction, and it's straightforward. Disjunction is a little bit less straightforward. Remember, we'll interpret '(A$\vee$B)' as the *inclusive* disjunction of A and B—either A or B *or both*.

Disjunctions are true when *at least one* of the disjuncts is true. And

they're *false* the rest of the time.

A	B	(A ∨ B)
0	0	0
0	1	1
1	0	1
1	1	1

That's not all of the truth table rules, but let's pause to check how you're going.

FOR YOU: Suppose we define (A⋆B) as "neither A nor B." Complete the truth table for the ⋆ connective:

A	B	(A ⋆ B)
0	0	
0	1	
1	0	
1	1	

ANSWER: This is the same truth table as for ~(A ∨ B).

A	B	(A ⋆ B)
0	0	1
0	1	0
1	0	0
1	1	0

COMBINING CONNECTIVES: We can evaluate one or more *complex formulas* in a single table, starting from inner-most atomic propositions and working outwards. For example, the formula ~(~p & ~q).

STAGE 0: Write out the table with enough rows an columns for the formula ~(~p & ~q), and write the values for each atomic proposition in the appropriate columns.

p	q	~	(~	p	&	~	q)
0	0			0			0
0	1			0			1
1	0			1			0
1	1			1			1

STAGE 1: Then process any values that can be defined in terms of the values of atoms in one step. In this case, it's the values of ~p and ~q.

p	q	~	(~	p	&	~	q)
0	0		1	0		1	0
0	1		1	0		0	1
1	0		0	1		1	0
1	1		0	1		0	1

STAGE 2: Then we can process the conjunction of the two negations:

p	q	~	(~	p	&	~	q)
0	0		1	0	1	1	0
0	1		1	0	0	0	1
1	0		0	1	0	1	0
1	1		0	1	0	0	1

STAGE 3: Finally, we can negate the result, to get the completed table.

p	q	~	(~	p	&	~	q)
0	0	0	1	0	1	1	0
0	1	1	1	0	0	0	1
1	0	1	0	1	0	1	0
1	1	1	0	1	0	0	1

This column of zeros and ones represents the truth value of ~(~p & ~q) in each of the different rows of the truth table. Each *column* represents the different truth values of a proposition which we encounter on the way toward building up the formula. Each *row* represents a different *possibility*. The first row represents a possibility in which p and q are both false. The zero at the head of the column we have highlighted (in the column for ~(~p & ~q)) says this: if p and q are both false, then ~(~p & ~q) is false too. Given that the four rows of the truth table represent the four different possibilities for p and q, then we have comprehensive information about the truth value of ~(~p & ~q). A TRUTH TABLE like this shows how the truth value of ~(~p & ~q) depends on the truth values of p and of q.

A possibility, or *the* possibility? Is there more than one possibility in which p and q are both false? That depends on how you count them.

If you compare this column with the column for the disjunction p ∨ q you'll notice something quite special:

p	q	~	(~	p	&	~	q)	(p ∨ q)
0	0	0	1	0	1	1	0	0
0	1	1	1	0	0	0	1	1
1	0	1	0	1	0	1	0	1
1	1	1	0	1	0	0	1	1

Notice that these two columns are *identical*. This tells us something very special—that in each circumstance, ~(~p & ~q) has the same truth value as p ∨ q. There is no circumstance where ~(~p & ~q) and p ∨ q have different truth values. They are, in an important sense we'll see in an upcoming section, they are *logically equivalent*.

1.2.3 | THE MATERIAL CONDITIONAL & BICONDITIONAL

The conditional is a more complex connective than the connectives we've already seen. When is it true to say "if I catch the tram before 8:20am, I'll get to my office on time"? Does the truth value of this depend only on the truth value of "I catch the tram before 8:20am" and "I'll get to my office on time"? Maybe not. But there is quite a lot of good work you can do with conditionals when you use a simple fact about conditionals as the focus of how you interpret them.

If A then B is *false* when A is true and B is false.

We can argue around in circles about when "if I catch the tram before 8:20am, I'll get to my office on time" is true or false. But we can *all* agree that in a circumstance where I *do* catch the tram before 8:20am and I *don't* get to my office on time, the conditional is *false*. The *material conditional* $A \supset B$ takes this to be the truth table of the statement "if A then B."

The circumstance where I get the tram but am late is a *counterexample* to the conditional. In general, a truth valuation v where $v(A) = 1$ and $v(B) = 0$ is a counterexample to the conditional $A \supset B$. A material conditional is true according to truth valuation v if v is not a counterexample to it.

The *material conditional* $A \supset B$ is *false* when A is true and B is false, and it is *true* otherwise.

A	B	(A ⊃ B)
0	0	1
0	1	1
1	0	0
1	1	1

In other words, $A \supset B$ asserts that there is a kind of *truth-transfer* directed from A to B. If $A \supset B$ is true (we are in rows 1, 2 or 4 of the table, not row 3) then if A is true too (so row 4, not 1 or 2) we then know that B is true too. Truth of $A \supset B$ is enough for the truth of A to transfer to the truth of B.

Notice that the the conditional $(A \supset B)$ has the same truth table as $\sim(A \,\&\, \sim B)$.

A	B	(A ⊃ B)	~	(A	&	~	B)
0	0	1	1	0	0	1	0
0	1	1	1	0	0	0	1
1	0	0	0	1	1	1	0
1	1	1	1	1	0	0	1

The material conditional $A \supset B$ is true in all cases *except for when* A is true and B false. The *material conditional* is the *truth-functional core* of the "implies" or "if...then..." connective in English, with the truth of the conditional expressed purely in terms of the truth or falsity of antecedent and consequent.

In natural language usage, we tend to assert "A implies B" or "if A then B" when we mean there is some kind *content connection* leading from A to B, in addition to a truth-transfer connection from A to B.

Here is an example of the distinctive behaviour of the material conditional: the formula $p \supset (q \supset p)$ is always true.

p	q	p	⊃	(q	⊃	p)
0	0	0	1	0	1	0
0	1	0	1	1	0	0
1	0	1	1	0	1	1
1	1	1	1	1	1	1

Let p be "I am a penguin", and let q be "Elvis is dead". Does the statement $p \supset (q \supset p)$ ("If I'm a penguin, then if Elvis is dead, I'm a penguin") strike you as true?

If *that* strikes you as true, what about this: p is "it's Tuesday" and q is "it's the weekend". Is "If it's Tuesday, then if it's the weekend, it's Tuesday" true?

FOR YOU: Compare the formulas $(p \supset \sim q)$ and $\sim(p \supset q)$ by completing their truth tables.

p	q	(p ⊃ ~ q)	~ (p ⊃ q)
0	0		
0	1		
1	0		
1	1		

ANSWER: $(p \supset \sim q)$ and $\sim(p \supset q)$ have different truth tables.

p	q	(p ⊃ ~ q)	~ (p ⊃ q)
0	0	1 0 1	0 1 0
0	1	1 1 0	0 1 1
1	0	0 1 1	1 0 0
1	1	1 0 0	1 0 1

When p is false, $(p \supset \sim q)$ is true while $\sim(p \supset q)$ is false, but when p is true, $(p \supset \sim q)$ and $\sim(p \supset q)$ have same truth value.

This is an important lesson to learn: the scope distinction between $\sim(p \supset q)$ and $(p \supset \sim q)$ makes a difference in their truth values. Natural languages don't always make it easy for us to distinguish between these two statements. We could express $\sim(p \supset q)$ in English as "Never q if p" while $(p \supset \sim q)$ could be expressed as "Not q if p" or more clearly as "p only if not q".

THE BI-CONDITIONAL: We'll round this section off with the semantics of the bi-conditional.

A *bi-conditional* formula $(A \equiv B)$ is true precisely when A and B have exactly the same truth value; $(A \equiv B)$ is false precisely when A and B have different truth values.

A	B	$(A \equiv B)$
0	0	1
0	1	0
1	0	0
1	1	1

The bi-conditional "if and only if" is indeed formed by conjoining an "if" with an "only if" as you'll see with this exercise:

FOR YOU: Let A and B be any formulas. Compare $(A \equiv B)$ with $((A \supset B) \,\&\, (B \supset A))$ by completing this truth table.

A	B	$(A \equiv B)$	$((A \supset B) \,\&\, (B \supset A))$
0	0		
0	1		
1	0		
1	1		

ANSWER:

A	B	$(A \equiv B)$	$((A \supset B) \,\&\, (B \supset A))$
0	0	1	1 1 1 1 1 1 1
0	1	0	0 1 0 1 0 1 0
1	0	0	0 1 0 1 0 0 0
1	1	1	1 1 1 1 1 1 1

Result: $(A \equiv B)$ has the same truth table as $((A \supset B) \,\&\, (B \supset A))$. The formula $(A \equiv B) \equiv ((A \supset B) \,\&\, (B \supset A))$ is always true!

The bi-conditional connective allows us to express *equality of truth-values*, since $(A \equiv B)$ is true exactly when A and B have equal truth values on every row of the truth table. Negating a bi-conditional then gives us a means to express *inequality* or *oppositeness* of truth-values.

The following worked truth table example shows that the negated bi-conditional $\sim(A \equiv B)$ in fact means the same as the bi-conditional $(A \equiv \sim B)$, of A with the negation of B. Notice this is in contrast with the situation for the one-directional conditional, where as we have seen, $\sim(A \supset B)$ does *not* mean the same as $(A \supset \sim B)$.

For those that choose the Combinational Digital Systems application area in Chapter 3, you will see the negated bi-conditional again as the *exclusive-OR* connective, in §3.2, where (AXORB) is true if and only if exactly one of A and B are true.

Let A and B be any formulas. Then the formula

$$\sim(A \equiv B) \equiv (A \equiv \sim B)$$

is always true.

A	B	~	(	A	≡	B	)	≡	(	A	≡	~	B	)
0	0	0	0	1	0	1	0	0	1	0				
0	1	1	0	0	1	1	0	1	0	1				
1	0	1	1	0	0	1	1	1	1	0				
1	1	0	1	1	1	1	1	0	0	1				

1.3 | PROPOSITIONAL CONCEPTS A

Once we have a way of explaining how the truth value of a complex proposition depends on the truth values of its constituents, the way is open for us to gain a better understanding of propositions—and the relationships between them. In this section, we'll look at two special classes of propositions, the *tautologies* and the *contradictions*; an we'll also look at what truth tables can tell us about relationships between propositions. We will start by classifying propositions into tautologies, contradictions and the rest.

"Among the possible groups of truth-conditions there are two extreme cases. In one of these cases the proposition is true for all the truth-possibilities of the elementary propositions. We say that the truth-conditions are *tautological*. In the second case the proposition is false for all the truth-possibilities: the truth-conditions are *contradictory*. In the first case we call the proposition a tautology; in the second, a contradiction. Propositions show what they say: tautologies and contradictions show that they say nothing. A tautology has no truth-conditions, since it is unconditionally true: and a contradiction is true on no condition. Tautologies and contradictions lack sense … Tautologies and contradictions are not, however, nonsensical. They are part of the symbolism, much as '0' is part of the symbolism of arithmetic." — Ludwig Wittgenstein, *Tractatus Logico-Philosophicus*, §4.46–4.4611.

1.3.1 | CLASSIFYING PROPOSITIONS

A propositional logic formula A is a tautology exactly when A evaluates to 1 on every truth valuation (every row of its truth table).

If we think of a proposition as drawing a *distinction* between those circumstances where it is true, an those where it is false, then a tautology is an extreme case, where *everything* counts as true.

EXAMPLE: $p \vee \sim p$ and $((p \supset q) \,\&\, \sim q) \supset \sim p$ are *tautologies*.

p	(	p	∨	~	p	)
0		0	1	1	0	
1		1	1	0	1	

p	q	(	(	p	⊃	q	)	&	~	q	)	⊃	~	p
0	0		0	1	0	1	1	0	1	1	0			
0	1		0	1	1	0	0	1	1	1	0			
1	0		1	0	0	0	1	0	1	0	1			
1	1		1	1	1	0	0	1	1	0	1			

FOR YOU: Show that $(p \vee q) \equiv \sim(\sim p \,\&\, \sim q)$ and $(p \supset q) \equiv (\sim q \supset \sim p)$ are tautologies.

We have provided a list of useful tautologies at the end of this chapter.

A tautology is always true. A *contradiction* is always *false*.

A propositional logic formula A is a contradiction exactly when A evaluates to 0 on every truth valuation (every row of its truth table).

EXAMPLE: $p \,\&\, \sim p$ and $(p \supset q) \,\&\, (p \,\&\, \sim q)$ are *contradictions*.

p	(p	&	~	p)
0	0	0	1	0
1	1	0	0	1

p	q	(p	⊃	q)	&	(p	&	~	q)
0	0	0	1	0	0	0	0	1	0
0	1	0	1	1	0	0	0	0	1
1	0	1	0	0	0	1	1	1	0
1	1	1	1	1	0	1	0	0	1

Some formulas are always true. Others are always false. But that leaves a lot of propositions in between. First of all, a statement that isn't a contradiction is at least *sometimes* true. We call that sort of statement *satisfiable*.

A propositional logic formula A is *satisfiable* exactly when A evaluates to 1 on at least one truth valuation.

A propositional logic formula A is *contingent* exactly when A is satisfiable but not a tautology. A contingent formula is true in at least one valuation and false in at least one valuation.

EXAMPLE: p and $(p \,\&\, \sim q) \,\&\, (\sim r \,\&\, s)$ are examples of contingent formulas.

Here are some facts connecting tautologies and contradictions.

A is a *tautology*	if and only if	~A is *not satisfiable*
	if and only if	~A is a *contradiction*.
A is *satisfiable*	if and only if	A is *not a contradiction*
	if and only if	~A is *not a tautology*.

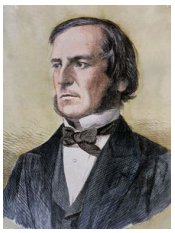
FOR YOU: Is $p \supset (\sim p \supset q)$ a tautology, a contradiction or contingent?

p	q	p	⊃	(~	p	⊃	q)
1	1	1	1	0	1	1	1
1	0	1	1	0	1	1	0
0	1	0	1	1	0	1	1
0	0	0	1	1	0	0	0

ANSWER: $p \supset (\sim p \supset q)$ is a tautology.

1.3.2 | INTERLUDE ON THE HISTORY OF PROPOSITIONAL LOGIC

Many people have been involved in the development of modern propositional logic over the last 170 or so years. Here are a just three of the significant figures in the early history of propositional logic.



George Boole (1815–1864): symbolic methods of algebra applied to logic. 1847: *The Mathematical Analysis of Logic*; 1854: *An Investigation of the Laws of Thought*.

(IMAGE CREDIT: *George Boole in color* by Author Unknown (c. 1860) [Public domain], via *Wikimedia Commons*. http://en.wikipedia.org/wiki/George_Boole.)

From Boole, we get the name 'Boolean logic' for classical two-valued propositional logic. In Boole's '*Mathematical Analysis of Logic*,' he applied mathematical techniques to represent logical relations.

Augustus De Morgan (1806–1871): algebra of logic; and-or duality. "De Morgan's Laws" are the equivalence of $\sim(A \vee B)$ and $\sim A \,\&\, \sim B$, and of $\sim(A \,\&\, B)$ and $\sim A \vee \sim B$.

(IMAGE CREDIT: *Augustus De Morgan, the mathematician* by Sophia Elizabeth De Morgan (1882) (*Memoir of Augustus De Morgan*) [Public domain], via *Wikimedia Commons*. http://en.wikipedia.org/wiki/Augustus_De_Morgan.)

Charles Sanders Peirce (1839–1914): truth tables, algebra of logic, not-or/nor (Peirce arrow).

(IMAGE CREDIT: *Charles Sanders Peirce* by Author Unknown (c. 1900) [Public domain], via *Wikimedia Commons*. http://en.wikipedia.org/wiki/Charles_Sanders_Peirce.)

There are many other people significant in the development of propositional logic, including Gottlob Frege, Bertrand Russell and Ludwig Wittgenstein, and logic has developed beyond philosophy and mathematics to other domains such as linguistics, engineering and computer science, as we'll see later in the course.

1.3.3 | RELATIONSHIPS BETWEEN PROPOSITIONS

For many applications of logic, we want to know about the relationships *between* propositions.

- Arguments from premises to conclusion.
- Linguistic analysis.
- Digital electronic circuits.

We start here with relationships between single propositions. The core relationship in logic is the relationship of *logical consequence*. The key idea is that one formula has another as a logical consequence if whenever the first is true, so is the second. There's no way for the first to be true when the second is false.

> Formula B is a LOGICAL CONSEQUENCE of formula A exactly when for all truth valuations in which A is true, B is also true.
>
> We will write this as $A \models B$.

This can also be read as: A logically entails B, or A logically implies B. It requires a *truth-transferring* directed connection from A to B.

A logically entails B means that for every row in a truth table in which A gets the value 1, B must also get the value 1 on that row.

EXAMPLE: $p \,\&\, {\sim}q \models {\sim}p \vee {\sim}q$.

p	q	p	&	~	q	~	p	∨	~	q
0	0	0	0	1	0	1	0	1	1	0
0	1	0	0	0	1	1	0	1	0	1
1	0	1	1	1	0	0	1	1	1	0
1	1	1	0	0	1	0	1	0	0	1

There is only one row where $(p \,\&\, {\sim}q)$ is true (the third row) and in that row, $({\sim}p \vee {\sim}q)$ is true too. So $p \,\&\, {\sim}q$ entails ${\sim}p \vee {\sim}q$.

Formula A is a LOGICALLY EQUIVALENT to formula B exactly when for all truth valuations in A and B have the same values.

We will write this as $A \dashv\models B$.

We've seen lots of logical equivalences already. Here are some.

$$\begin{aligned} A \vee B &\dashv\models {\sim}({\sim}A \,\&\, {\sim}B) \\ A \,\&\, B &\dashv\models {\sim}({\sim}A \vee {\sim}B) \\ A \supset B &\dashv\models {\sim}(A \,\&\, {\sim}B) \\ A \supset B &\dashv\models {\sim}A \vee B \end{aligned}$$

Here is an example equivalence.

EXAMPLE: $p \,\&\, {\sim}q \models {\sim}p \vee {\sim}q$.

p	q	p	&	~	q	~	(~	p	∨	q)
0	0	0	0	1	0	0	1	0	1	0
0	1	0	0	0	1	0	1	0	1	1
1	0	1	1	1	0	1	0	1	0	0
1	1	1	0	0	1	0	0	1	1	1

The two formulas have the same values in each row: true in row 3 and false in all others.

Following direct from the definitions, here are some equivalent characterisations of logical consequence, logical equivalence, and being a tautology.

- $A \dashv\models B$ if and only if $A \models B$ and $B \models A$.
- $A \models B$ if and only if $(A \supset B)$ is a tautology.
- $A \dashv\models B$ if and only if $(A \equiv B)$ is a tautology.

FOR YOU: Suppose A and B are propositional formulas containing atoms p, q and r, and their truth tables look like this:

p	q	r	A	B
0	0	0	0	0
0	0	1	1	0
0	1	0	1	1
0	1	1	1	1
1	0	0	0	0
1	0	1	1	1
1	1	0	1	1
1	1	1	0	0

Which of these correctly describe the logical relationship between formulas A and B?

(a) $A \dashv\models B$. (b) $A \models B$ but $B \not\models A$.
(c) $B \models A$ but $A \not\models B$. (d) None of the above.

ANSWER: Option (c): $B \models A$ but $A \not\models B$. All of the rows where B is true are rows where A is true too, so $B \models A$. On the other hand, there is a row (the second row, where $i(p) = 0$, $i(q) = 0$ and $i(r) = 1$) where A is true and B isn't, so $A \not\models B$.

By applying a sequence of known logical equivalences (such as those listed among the tautologies at the end of this Chapter around page 43), we can see how they can be seen as "re-writing" rules, as a means to get a formula into an equivalent desired shape or form. This becomes important in Chapter 3 on **Combinational Digital Systems**, and also in Chapter 6, on **Propositional** PROLOG. For example:

	$(r \,\&\, \sim s) \supset (\sim p \vee \sim q)$	
$\equiv$	$(r \,\&\, \sim s) \supset \sim(p \,\&\, q)$	De Morgan's equivalence
$\equiv$	$\sim\sim(r \,\&\, \sim s) \supset \sim(p \,\&\, q)$	Double negation
$\equiv$	$(p \,\&\, q) \supset \sim(r \,\&\, \sim s)$	contraposition equivalence
$\equiv$	$(p \,\&\, q) \supset \sim(\sim\sim r \,\&\, \sim s)$	Double negation
$\equiv$	$(p \,\&\, q) \supset (\sim r \vee s)$	De Morgan's equivalence
$\equiv$	$(p \,\&\, q) \supset (r \supset s)$	material conditional
$\equiv$	$((p \,\&\, q) \,\&\, r) \supset s$	Curry's formula
$\equiv$	$(p \,\&\, q \,\&\, r) \supset s$	Associativity of &

(Formulas of the form $((p \,\&\, q \,\&\, r) \supset s)$ will be known as *program clauses* in Chapter 6.) In this example, we have made every step explicit. As you get more practiced, you will be able to combine steps, particularly those involving double negation. For example, $(A \supset \sim B) \equiv$

$(B \supset \sim A)$ via contraposition equivalence with a double negation left implicit. The reasoning principle being used here goes by the name of *Substitutivity of Equivalents*, and is along the same lines as substitution of equals for equals in elementary mathematics. The principle says:

$$\text{If } A \equiv B \quad \text{then } C[p := A] \equiv C[p := B]\,,$$

where A, B and C are logic formulas with C containing atom p, and $C[p := A]$ is the result of replacing every occurrence of p in C with formula A. The principle says: if A and B are logically equivalent and they are both substituted for an atom in a further formula C, then the result is two more logically equivalent formulas, $C[p := A]$ and $C[p := B]$.

For another example,

$$\begin{aligned}
& (p \,\&\, \sim q \,\&\, \sim r) \vee (p \,\&\, \sim q \,\&\, r) \vee (p \,\&\, q \,\&\, \sim r) \\
\equiv\ & (p \,\&\, \sim q \,\&\, (\sim r \vee r)) \vee (p \,\&\, \sim r \,\&\, (\sim q \vee q)) \\
& \quad \text{Distribution of } \& \text{ over } \vee\text{, plus Commutativity of } \&, \vee \\
\equiv\ & (p \,\&\, \sim q) \vee (p \,\&\, \sim r) \\
& \quad \text{Cancelling equivalence} \\
\equiv\ & (p \,\&\, (\sim q \vee \sim r)) \\
& \quad \text{Distribution of } \& \text{ over } \vee \\
\equiv\ & (p \,\&\, \sim(q \,\&\, r)) \\
& \quad \text{De Morgan's equivalence}
\end{aligned}$$

Here, the *Cancelling equivalence* $(A \,\&\, (\sim B \vee B)) \equiv A$ expresses that if you have a *conjunction* with a tautology like $(\sim B \vee B)$, then you can cancel out the always true part. Likewise, $(A \vee (\sim B \,\&\, B)) \equiv A$ expresses that if you have a *disjunction* with a contradiction like $(\sim B \,\&\, B)$, then you can cancel out the always false part.

At the other extreme, the relationship opposite to logical equivalence is that of being *contradictories*.

> Formulas A and B are contradictories exactly when for all truth valuations, A and B have *opposite* truth values.

So A and B being contradictories means that B is be false whenever A is true, and B is true whenever A is false.
For any propositional formulas A, B:

$$\begin{aligned}
& A \text{ and } B \text{ are } \textit{contradictories} \\
\text{if and only if } \quad & A \dashv\models \sim B \\
\text{if and only if } \quad & (A \equiv \sim B) \ \text{ is a } \textit{tautology}.
\end{aligned}$$

Here are some further logical relations worth knowing, that date back to Aristotelian and medieval logic.

Formulas A and B are *contraries* when they cannot both be true.

So A and B being contraries means that B is false whenever A is true, and A is false whenever B is true.

Formulas A and B are *sub-contraries* when they cannot both be false.

In contrast, A and B being sub-contraries means that B is true whenever A is false, and A is true whenever B is false.

The relationships of being contraries and sub-contraries have the following equivalent characterisations. For any formulas A, B:

	A and B are *contraries*
if and only if	$A \models \sim B$
if and only if	$(A \supset \sim B)$ is a *tautology*
if and only if	$(A \,\&\, B)$ is a *contradiction*
if and only if	$\sim(A \,\&\, B)$ is a *tautology*.
	A and B are *sub-contraries*
if and only if	$\sim A \models B$
if and only if	$(\sim A \supset B)$ is a *tautology*
if and only if	$(A \vee B)$ is a *tautology*.

SOME TAUTOLOGIES

This list is by no means exhaustive: you are encouraged to add to the list as you discover new ones. You do not have to memorise the list, but to develop your logic intuitions, it will be useful to read through and notice the patterns in it.

Let A, B and C be any formulas of a propositional logic language. The following are tautologies of propositional logic.

Double negation elim.:	$\sim\sim A \equiv A$
Excluded middle:	$(A \vee \sim A)$ and $\sim(A \,\&\, \sim A)$
Associativity of &:	$((A \,\&\, B) \,\&\, C) \equiv (A \,\&\, (B \,\&\, C))$
Associativity of $\vee$:	$((A \vee B) \vee C) \equiv (A \vee (B \vee C))$
Commutativity of &:	$(A \,\&\, B) \equiv (B \,\&\, A)$
Commutativity of $\vee$:	$(A \vee B) \equiv (B \vee A)$
Distribution of & over $\vee$:	$(A \,\&\, (B \vee C)) \equiv ((A \,\&\, B) \vee (A \,\&\, C))$
Distribution of $\vee$ over &:	$(A \vee (B \,\&\, C)) \equiv ((A \vee B) \,\&\, (A \vee C))$
De Morgan's for $\sim$ of & to $\vee$:	$\sim(A \,\&\, B) \equiv (\sim A \vee \sim B)$
De Morgan's for $\sim$ of $\vee$ to &:	$\sim(A \vee B) \equiv (\sim A \,\&\, \sim B)$
De Morgan's for $\sim$ of & to $\vee$:	$\sim(\sim A \,\&\, \sim B) \equiv (A \vee B)$
De Morgan's for $\sim$ of $\vee$ to &:	$\sim(\sim A \vee \sim B) \equiv (A \,\&\, B)$
Material conditional:	$(A \supset B) \equiv (\sim A \vee B)$
Negated conditional:	$\sim(A \supset B) \equiv (A \,\&\, \sim B)$
Conditionals for &:	$(A \,\&\, B) \supset A$ and $(A \,\&\, B) \supset B$
Conditionals for $\vee$:	$A \supset (A \vee B)$ and $B \supset (A \vee B)$
Weakening:	$A \supset (B \supset A)$
Distribution:	$(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
Contraposition:	$(\sim A \supset \sim B) \supset (B \supset A)$
Contraposition 2:	$(A \supset B) \supset (\sim B \supset \sim A)$
Contraposition equivalence:	$(A \supset B) \equiv (\sim B \supset \sim A)$
Pierce's Law:	$((A \supset B) \supset A) \supset A$
Modus ponens conditional:	$(A \,\&\, (A \supset B)) \supset B$
Curry's Formula:	$((A \,\&\, B) \supset C) \equiv (A \supset (B \supset C))$
Excluding disjuncts:	$((A \vee B) \,\&\, \sim A) \supset B$
	$((A \vee B) \,\&\, \sim B) \supset A$

Using only the inference rule of *Modus Ponens*: from A and $(A \supset B)$, conclude B, one can give a minimal axiomatisation of classical propositional logic by taking as axioms just the three: Weakening, Distribution and Contraposition. This axiomatisation was first discovered by the Polish logician Jan Łukasiewicz in the 1920s.

BLANK TRUTH TABLES

Here are some blank truth tables to print out and use for your own work, for three, four and five atomic propositions.

p	q	r		
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

0	0	0	0	
0	0	0	1	
0	0	1	0	
0	0	1	1	
0	1	0	0	
0	1	0	1	
0	1	1	0	
0	1	1	1	
1	0	0	0	
1	0	0	1	
1	0	1	0	
1	0	1	1	
1	1	0	0	
1	1	0	1	
1	1	1	0	
1	1	1	1	

0	0	0	0	0	
0	0	0	0	1	
0	0	0	1	0	
0	0	0	1	1	
0	0	1	0	0	
0	0	1	0	1	
0	0	1	1	0	
0	0	1	1	1	
0	1	0	0	0	
0	1	0	0	1	
0	1	0	1	0	
0	1	0	1	1	
0	1	1	0	0	
0	1	1	0	1	
0	1	1	1	0	
0	1	1	1	1	
1	0	0	0	0	
1	0	0	0	1	
1	0	0	1	0	
1	0	0	1	1	
1	0	1	0	0	
1	0	1	0	1	
1	0	1	1	0	
1	0	1	1	1	
1	1	0	0	0	
1	1	0	0	1	
1	1	0	1	0	
1	1	0	1	1	
1	1	1	0	0	
1	1	1	0	1	
1	1	1	1	0	
1	1	1	1	1	

PROOFS FOR PROPOSITIONAL LOGIC 2

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

These are the ideas and concepts we've introduced in this chapter.

Use these summary sections at the start of each chapter to keep track of what you're learning. We have quizzes and discussion forums online for you to practice your skills and get familiar with each of these concepts.

- *Validity* and *soundness* of argument forms.
- Method of assigning truth values.
- *Proof trees* as a proof technique for propositional logic, and their rules for development and completion: for each 2-place propositional connective, two rules: one positive and one negative.
- *Proof trees for testing validity of argument forms*, tautology status of formulas, and satisfiability of formulas.
- *Soundness* and *completeness* of the proof tree method for propositional logic (understand the theorem).

SKILLS

These are the skills you should be able to demonstrate.

- Determine if an *argument form* is *valid* using *truth tables*.
- Use a *proof tree* to determine *validity* of argument form, or the *status* of formula.
- Determine when a proof tree is *completely developed*.
- From an open branch of a proof tree, *construct a truth valuation* for atomic formulas that makes all formulas on the branch true.

2.1 | PROPOSITIONAL CONCEPTS B

In the previous chapter we looked at truth tables for the language of propositional logic, and we used them to classify formulas into the tautologies, the contradictions and the contingencies. We also looked at relationships between pairs of formulas. We can do more than this, and philosophers, especially, are interested in another kind of relationship between propositions—the relation of validity, for an *argument*.

2.1.1 | VALIDITY

An argument connects a collection of *premises* to a *conclusion*. We offer an argument when justify a *conclusion* on the basis of *premises*. Not all arguments are equally good. Presumably, and arument is *really bad* if

the premises of the argument are all true and the conclusion is not true. Then you've definitely made a mistake if you step from the premises to the conclusion, because the premises were all good (all true) yet the conclusion is not (it's false). There, the mistake you make is a mistake in the *argument* (the transition between the premises and the conclusion) and not in the premises themselves (they were all true).

An argument, on the other hand, is *really good* if that mistake *can never happen*. That is, an argument is really good—we use the word *valid* for this—if there is no possibility at all where the premises are true and the conclusion is false. Or if you like, it's valid if in any circumstance where the premises are true, the conclusion is true, too. In this way, for a valid argument, the conclusion is somehow already contained in the conclusion. Any way to make the world that makes the premises true will bring the conclusion along with it. There is no extra information needed to make the conclusion true, over and above what is in the premises.

This motivates the following definition of validity:

An argument is *valid* if and only if *in any valuation where the premises are all true, the conclusion is also true*.

For an argument with premises $A_1, A_2, \ldots, A_n$ and conclusion B, we write $A_1, A_2, \ldots A_n \models B$ when the argument from $A_1, A_2, \ldots, A_n$ to conclusion B is valid.

When Σ is a *collection* or *set* of premises, we also write $\Sigma \models B$ when the argument from premises Σ to conclusion B is valid. The validity $A_1, A_2, \ldots A_n \models B$ requires truth-transfer from the collection of premises $A_1, A_2, \ldots, A_n$ to the conclusion B.

Equivalently, an argument is valid if and only if there is no valuation where the premises $A_1, A_2, \ldots, A_n$ are all true and at the same time the conclusion B is false.

EXAMPLE: Does $(p \,\&\, {\sim}q) \supset r, {\sim}r \models p \supset q$?

You can answer this by checking if there is a row of a truth table where $(p \,\&\, {\sim}q) \supset r$ and ${\sim}r$ are true, and $p \supset q$ is false. You can check this by doing a simultaneous truth table for the premises and the conclusion, and checking each row.

Here is the truth table.

p	q	r	(p & ~q) ⊃ r	~r	p ⊃ q	
0	0	0	1	1	1	✓
0	0	1	1	0	1	
0	1	0	1	1	1	✓
0	1	1	1	0	1	
1	0	0	0	1	0	
1	0	1	1	0	0	
1	1	0	1	1	1	✓
1	1	1	1	0	1	

We have highlit every row where the premises of the argument are all true. And you can check that in these rows, the conclusion is true, too. So, the argument is valid.

VALIDITY AND THE MATERIAL CONDITIONAL: From the relationship between logical consequence and tautologies, we can also derive a corresponding relationship between argument *validity* and *tautologies*.

DEDUCTION THEOREM (SEMANTIC FORM):

$$A_1, A_2, \ldots, A_n \models B$$
if and only if
$(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) \supset B$ is a tautology
if and only if
$$A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n \models B.$$

Can you see why these three statements are equivalent? First, we have $A_1, A_2, \ldots, A_n \models B$ if and only if in every truth valuation i, if $i(A_1) = i(A_2) = \cdots = i(A_n) = 1$ then $i(B) = 1$ too. This means that in every truth valuation i, if $i(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) = 1$ then $i(B) = 1$ (since $i(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) = 1$ if and only if $i(A_1) = i(A_2) = \cdots i(A_n) = 1$. So, it follows that $A_1, A_2, \ldots, A_n \models B$ if and only if $A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n \models B$. But $A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n \not\models B$ if and only if there's some valuation i where $i(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) = 1$ and $i(B) = 0$. That happens if and only if there is some valuation i where $i((A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) \supset B) = 0$, and that happens if and only if $(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) \supset B$ is not a tautology. So, $A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n \not\models B$ if and only if $(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) \supset B$ is not a tautology, or equivalently, $A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n B$ if and only if $(A_1 \mathbin{\&} A_2 \mathbin{\&} \cdots \mathbin{\&} A_n) \supset B$ is a tautology.

A conjunction is true if and only if each of the conjuncts are true

FOR YOU: Test whether or not $\sim p \supset q,\ p \supset \sim r \models q \supset r$

ANSWER: The truth table, with rows highlit where the premises are true gives us the answer.

p	q	r	~p ⊃ q	q ⊃ ~r	p ⊃ r
0	0	0	0	1	1
0	0	1	0	1	1
0	1	0	1	1	0 ×
0	1	1	1	1	1 ✓
1	0	0	1	1	1 ✓
1	0	1	1	0	1
1	1	0	1	1	0 ×
1	1	1	1	0	1

The firts and last of these rows (row 3—$i(p) = 0$, $i(q) = 1$ and $i(r) = 0$; and row 7—$i(p) = 1$, $i(q) = 1$ and $i(r) = 0$) have the premises true and the conclusion false. This shows that the argument form is not valid. We have $\sim p \supset q$, $p \supset \sim r \not\models q \supset r$.

You can test many different argument forms by doing simultaneous truth tables for the premises and the conclusion. A *counterexample* is a row which makes the premises true and the conclusion false. An argument with no counterexample is *valid*.

FOR YOU: Suppose you have an argument from premises Σ to conclusion B. And suppose B is a tautology. What can you say about the argument?

Suppose the premises Σ are jointly inconsistent—that is, there is no row where the premises Σ are all true. What can we say about the argument from Σ to B then?

These facts mean that you can have a valid argument where the conclusion has nothing to do with the premises (if the conclusion is a tautology or the premises are jointly inconsistent). This phenomenon prompts some people to search for a different account of validity, according to which the premises must be *relevant* to the conclusion of the argument. These so-called *relevant* logics are very interesting, but looking at them would take us too far afield [1, 2, 9, 11].

ANSWER: In both cases, the argument is valid, because there is no counterexample. In the first case, since B is a tautology, there is no valuation where B is false, so there is no valuation which makes all of the premises Σ true and B false.

On the other hand, if Σ is jointly inconsistent, then there is no counterexample to the argument from Σ to B since there is no valuation where the elements of Σ are all true.

2.1.2 | ARGUMENT FORMS

Argument forms are the forms of real-life arguments. We use the formal validity of an argument form to tell us something about the validity of a real-life argument, expressed in a natural language. Here's an ex-

ample. Consider the following two arguments. They share a "shape" or a "form", and the validity of the two arguments is due to the validity of their common *argument form*.

If genetics determines all our behaviour then people are merely robots. People *aren't* merely robots. *Therefore*, Genetics doesn't determine all our behaviour.	if p, then q not q. ∴ not p.
If living expenses in this city are high, then many students are struggling financially. Not many students are struggling financially *Therefore*, Living expenses in this city are not high.	if p, then q. not q. ∴ not p.

An argument has a given argument form if you can find statements to substitute for the atomic propositions in the form, so that the resutl you get is (or is synonymous with) the original argument.

MULTIPLE FORMS: This means that one argument can be an instance of multiple different forms. The two arguments we have considered here are *also* instances of other quite different argument forms, such as:

If p then q.
r.
Therefore, s.

In this case, we can find values for p, q, r and s such that when we substitute *them*, we get the original argument. For example, if p is "genetics determines all our behaviour," q is "people are merely robots," r is "people aren't merely robots," and s is "genetics doesn't determine all our behaviour" then we get our original argument back. It has more than one form, depending in how much structure is in focus. With *this* form, we focus on the conditional in the first premise and forget the rest. (*Any* argument with two premises, the first of which is a conditional, has the form "if p then q; r; therfore s.")

FOR YOU: What is wrong with this argument form, just from looking at it?

ANSWER: It's invalid. Any valuation that makes p, q and r true and s false makes the premises true and the conclusion false.

That form is, clearly, invalid, and it has invalid instances. Here's an example in an instance of this form.

If you are a logician, then you are a geek.
Greenhouse gas levels have increased.
Therefore, the earth is flat.

In fact, we think that the premises are *true* and the conlcusion *false*.

This is cleary an invalid argument. But *valid* arguments can also be instances of *invalid* forms. The two valid arguments we've already seen are instances of the invalid form above, as well as the valid form we've already seen.

The more interesting question is this: does every valid argument have a valid form? That is, does all argument validity come down to *formal* validity?

FOR YOU: Which is the best choice of argument form for this concrete argument?

> We can conclude that the people will be unhappy. This is because either the government will increase the tax rate or it won't. If it does, the people will have less income to spend and they will be unhappy. If it doesn't, there will be fewer public services and the people will be unhappy.

(a)
$$\frac{\begin{array}{c} p \\ q \vee r \\ q \supset (s \,\&\, p) \end{array}}{\therefore r \supset (t \,\&\, p)}$$

(b)
$$\frac{\begin{array}{c} q \vee r \\ q \supset (s \,\&\, p) \\ r \supset (t \,\&\, p) \end{array}}{\therefore p}$$

(c)
$$\frac{\begin{array}{c} q \vee \sim q \\ q \supset (s \,\&\, p) \\ \sim q \supset (t \,\&\, p) \end{array}}{\therefore p}$$

(d)
$$\frac{\begin{array}{c} p \\ q \vee \sim q \\ q \supset (s \,\&\, p) \end{array}}{\therefore \sim q \supset (t \,\&\, p)}$$

To find the answer, form a dictionary, locate the connectives in the premises and conclusion, and then simultaneously translate each statement into the language of propositional logic using that dictionary.

(c)
$$\frac{\begin{array}{c} p \\ (s \mathbin{\&} p) \supset q \\ (t \mathbin{\&} p) \supset \sim q \end{array}}{\therefore \sim(s \mathbin{\&} t)} \qquad \sim q \vee \sim\sim q$$

We have the following form:

p : The people will be unhappy.
q : The government will increase the tax rate.
s : The people will have less income to spend.
t : There will be fewer public services.

ANSWER: Using this *dictionary*

IS VALIDITY ENOUGH? Consider this argument.

(*Premise*) If living expenses in this city are high,
then many students are struggling financially.
(*Premise*) Not many students are struggling financially.
(*Conclusion*) *Hence*, living expenses in this city are not high.

QUESTION: This argument is valid—so why is there still something wrong with it? The conclusion is *false*, at least when you consider Melbourne, where living expenses are quite high.

DEFINITION: An argument is said to be *sound* if and only if it is an is is *valid*, and in addition, the premises are in fact all true.

Hence the *conclusion* of a sound argument must also be true, since it's valid, and the premises are true.

This completes our tour of models (interpretations, truth tables) for propositional logic, and concepts we can define using those models. We'll turn to another way of studying logic, by way of *proofs* rather than models. Let's start, by looking at why proof systems—like proof trees—are important.

2.2 | PROOFS FOR PROPOSITIONAL LOGIC: TREES

2.2.1 | WHY WE NEED PROOF TREES

Proof trees are useful and important because of the problem of exponential growth we've already seen. As the number of atomic propositions increases, the number of rows of the truth table required to check these propositions grows ever faster. We won't consider an argument with 60 atomic propositions. One with 4 will be enough to illustrate the point. Consider this argument:

With 60 atomic propositions, $2^{60} > 10^{18}$ rows are required.

Prem. If Melbourne's population reaches 5 million by 2032 then this city needs a massive investment in either public transport or roads.

Prem. If Melbourne is serious about combating climate change, then it does not need a massive investment in roads.

Therefore,

Conc. If Melbourne's population reaches 5 million by 2032 and this city is serious about climate change, then it needs a massive investment in public transport.

Here is the dictionary and formalisation:

p = Melbourne's population reaches 5 million by 2032.
q = Melbourne needs a massive investment in public transport.
r = Melbourne needs a massive investment in roads.
s = Melbourne is serious about climate change.

Prem. $p \supset (q \vee r)$

Prem. $s \supset \sim r$

Conc. $(p \,\&\, s) \supset q$

So, we have an argument form, with four atomic propositions, p, q, r and s. Completing a truth table for this will require 16 rows. The truth table is in Figure 2.1. We have highlit every row where the conclusion

p	q	r	s	p	⊃	(q ∨ r)	s	⊃	~r	(p & s)	⊃	q
0	0	0	0	0	1	0	0	1	1	0	1	0
0	0	0	1	0	1	0	1	1	1	0	1	0
0	0	1	0	0	1	1	0	1	0	0	1	0
0	0	1	1	0	1	1	1	0	0	0	1	0
0	1	0	0	0	1	1	0	1	1	0	1	1
0	1	0	1	0	1	1	1	1	1	0	1	1
0	1	1	0	0	1	1	0	1	0	0	1	1
0	1	1	1	0	1	1	1	0	0	0	1	1
1	0	0	0	1	0	0	0	1	1	0	1	0
1	0	0	1	1	0	0	1	1	1	1	0	0
1	0	1	0	1	1	1	0	1	0	0	1	0
1	0	1	1	1	1	1	1	0	0	1	0	0
1	1	0	0	1	1	1	0	1	1	0	1	1
1	1	0	1	1	1	1	1	1	1	1	1	1
1	1	1	0	1	1	1	0	1	0	0	1	1
1	1	1	1	1	1	1	1	0	0	1	1	1

Figure 2.1: A 16 row truth table

of the argument is false. And you can check that in these rows, the one of the premises is false, too. So, the argument is *valid*. There is no counterexample.

This required a very lengthy process of checking. It takes a long time to go through every row. This workload doubles with every new atomic proposition.

THERE IS A BETTER WAY: We don't need to go through *every* row of the truth table to establish whether there is a counterexample or not. Instead, we could attempt to *locate* a counterexample. We start off by writing out the premises and the negation of the conclusion, and in a row beneath them, attempt to place values for each formula, starting with the premises getting the value 1 an the conclusion getting the value 0, like this:

p	q	r	s	p	⊃	(q	∨	r)	s	⊃	~	r	(p	&	s)	⊃	q
							1		1							0	

Once you gave this, you work backwards, from complex formulas to their constituents, inferring the value of the constituents where possible from the value of the formula as a whole. From here, for example, given that (p & s) ⊃ q is false, it follows that p & s is true (so p and s are both true) and q is false. This, then, is what begins to determine the values of the atomic propositions (if we find a consistent valuation, that is). At this stage, we have

This is a point where the *video* of this class material is much more helpful, because you can see the process worked out step-by-step.

p	q	r	s	p	⊃	(q	∨	r)	s	⊃	~	r	(p	&	s)	⊃	q
1	0		1		1					1			1	1	1	0	0

We can fill in the values of p, q and s elsewhere, to get:

p	q	r	s	p	⊃	(q	∨	r)	s	⊃	~	r	(p	&	s)	⊃	q
1	0		1	1	1	0			1	1			1	1	1	0	0

Now since s ⊃ ~r is true and s is true, we must have ~r be true, which makes r false.

p	q	r	s	p	⊃	(q	∨	r)	s	⊃	~	r	(p	&	s)	⊃	q
1	0	0	1	1	1	0		0	1	1	1	0	1	1	1	0	0

But notice the problem with p ⊃ (q ∨ r). The conditional is meant to be true, and p is true, to, so q ∨ r must be true. But q and r are both false. So the value of q ∨ r is impossible to set consistently. We mark it with a cross.

p	q	r	s	p	⊃	(q	∨	r)	s	⊃	~	r	(p	&	s)	⊃	q
1	0	0	1	1	1	0	×	0	1	1	1	0	1	1	1	0	0

There is no row of a truth table that can make the premies true and the conclusion false. The argument is valid.

This technique is sometimes called the 'method of assigning values'—MAV.

This reasoning is a much more efficient treatment of testing for validity than naïvely collecting all of the truth table rows in the language. Unfortunately, the trace of the reasoning, once you've completed it, is simply a row or two of zeros and ones, under the formulas you're testing. This gives very little guidance as to how to read this list of zeros and ones, little idea of what steps of reasoning were taken, and in what order. *Tree proofs* are a way of representing that kind of reasoning much more explicitly. A tree proof fo this argument starts with the premise and the negation of the conclusion, written down in one list.

$$
\begin{array}{c}
p \supset (q \vee r) \\
s \supset \sim r \\
\sim((p \,\&\, s) \supset q)
\end{array}
$$

In a tree proof, whenever we write down a formula we're taking it to be true. So writing down the premises and the negation of the conclusion indicates that we're attempting to find a counterexample to the argument. The negation of the conclusion is a negated conditional. So if this is true, then the antecedent is true and the consequent is false. So we can write the antecedent and the negation of the consequent. These *follow* from what we've already written.

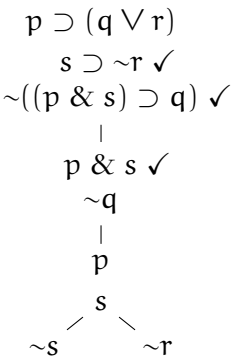
$$
\begin{array}{c}
p \supset (q \vee r) \\
s \supset \sim r \\
\sim((p \,\&\, s) \supset q) \checkmark \\
| \\
p \,\&\, s \\
\sim q
\end{array}
$$

We have marked $\sim((p \,\&\, s) \supset q)$ with a tick, indicating that we've *processed* it. We have extracted the information from it that we need. Then we wrote down the conjunction $p \,\&\, s$. If this is true, then so are p and s. So we'll write these down, too.

$$
\begin{array}{c}
p \supset (q \vee r) \\
s \supset \sim r \\
\sim((p \,\&\, s) \supset q) \checkmark \\
| \\
p \,\&\, s \checkmark \\
\sim q \\
| \\
p \\
s
\end{array}
$$

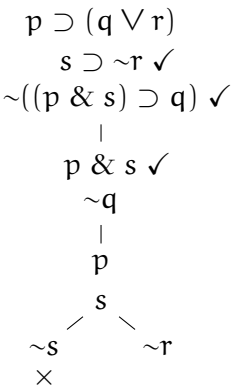
and we tick the $p \,\&\, s$ to indicate that it's been processed, too. The formulas left unticked are $p \supset (q \vee r)$, $s \supset \sim r$, $\sim q$, p and s. The last three are inert. There is no information to extract from these, other than the wish to make q false, and p and s true. (We call atomic formulas and their negations *literals*. Literals don't get processed in trees. All other formulas are complex, and can be processed. But literals just sit there and tell you what is to be true or false.) But $s \supset \sim r$ is complex—it tells us something. It tells us that either s is false or $\sim r$ is true. So, we process this formula and write down $\sim s$ and $\sim r$ *in two distinct branches*.

Remember: if a conditional is true then either the antecedent is false or the consequent is true.

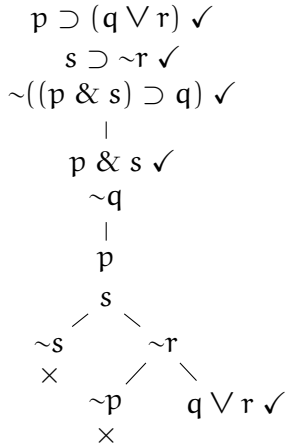


So now, our tree has two branches. One starting at the top and going down to ~s, a leaf, and the other, starting at the top and going down to ~r, the other leaf. These represent two distinct possibilities. But the first of these possibilities isn't actually a possibility, beccause we want s to be true (we'd already written that down) yet we want ~s to be true too. That can't happen, so the left branch *closes*.

Notice we have ticked s ⊃ ~r. We won't keep mentioning when we've processed formulas from now. You've got the idea, we hope.

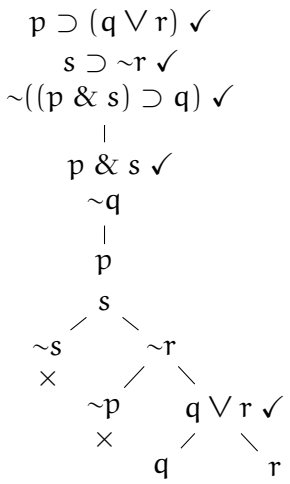


We mark that branch with a × to indicate that it's a dead end. In this tree only one option is left. We have only one complex unprocessed formula, the first premise p ⊃ (q ∨ r). This is a conditional, and it tells us that either p is false, or q ∨ r is true. So we split into two branches, one with ~p and the other with q ∨ r.

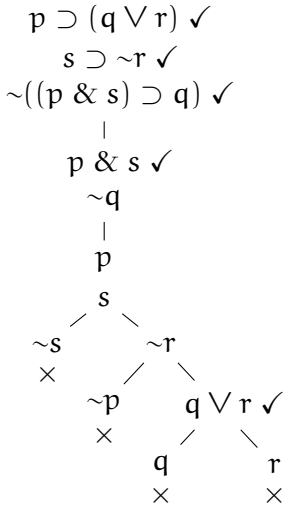


The left branch contains ~p at the leaf and p further up (in the trunk) so we close that left branch immediately. The right branch, however,

contains $q \vee r$ which can itself be processed, by branching into a q branch, and an r branch.



And we can notice that these two branches contain contradictory pairs, too. For q our tre contains $\sim q$ in the trunk. For r, a little step up the branch we have $\sim r$. So neither of these branches is open. The result is a tree in which every branch is closed.



So, to test the validity of the argument from premises $p \supset (q \vee r)$ and $s \supset \sim r$ to conclusion $(p \,\&\, s) \supset q$, we have tried to make the *premises true* and the *conclusion false*, but all our attempts to do so have lead to contradictions (and we marked each branch with a $\times$). This shows that the argument is valid. There is no counterexample, no way to have the premises true and the conclusion false.

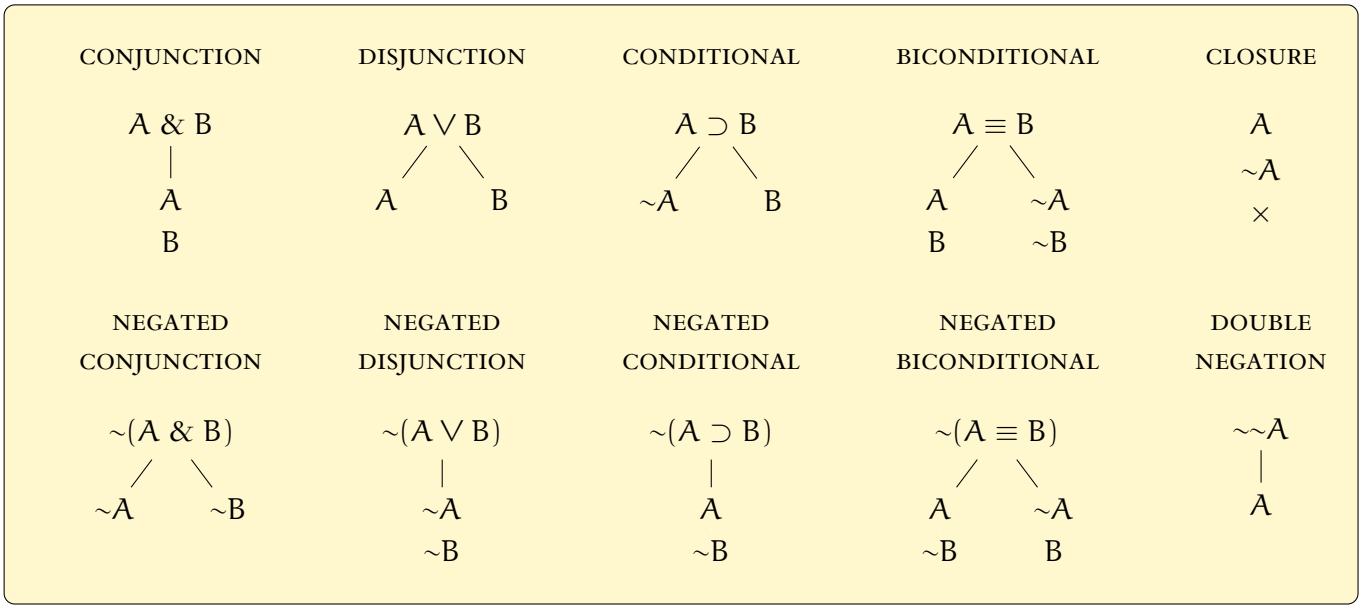
This is an example of a PROOF TREE, a *graphical* and *mechanical* way of determining whether an argument form is valid, or whether a formula is a tautology. The tree, read from top to bottom, clearly represents the unfolding reasoning, as we unpack each formula into its constituent parts, and systematically keep track of the options, closing some off as contradictions arise, and developing others, as far as we can.

A tree for a set of formulas CLOSES if and only if each of its branches contains a contradiction. Then there's no way for the starting formulas to be true together. So, if a tree starting with the formula A closes, then A is a contradiction. If a tree starting with ~A closes, then A is a tautology. If a tree for Σ and ~A closes, then the argument from Σ to A is valid.

In many cases, completing a proof tree for an argument will be a much shorter process than completing all 16 or 32 or 64… rows of a truth table. Proof trees are an efficient way of not only showing that an argument is valid, but doing so in such a way that makes the steps of reasoning explicit, and represented in such a way that they can be independently checked.

2.2.2 | RULES FOR PROOF TREES FOR PROPOSITIONAL LOGIC

So, what are the rules for producing trees? The first rules are the *processing rules*. These are rules for every complex formula.



There are rules for every kind of complex formula: for a conjunction, disjunction, conditional or biconditional, and negated versions of these, as well as for a double negation. Each of these rules is designed to extract information that's in the starting formula—and extract *all* of that information.

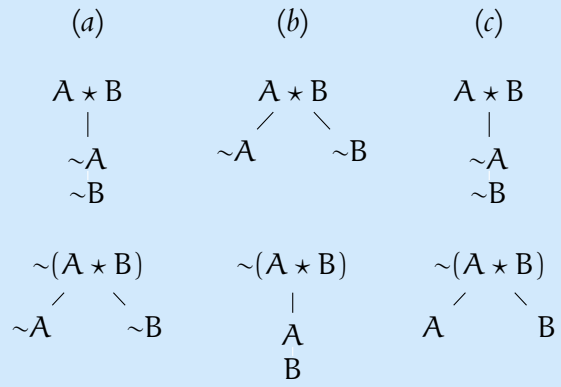
Check for yourself that every formula that is not a literal (not an atom or a negated atom) falls into exactly one of these nine kinds. HINT: look at the main connective or operator of each non-literal. If it's a negation, it's not a negation of an atom, it's a negation of something else.

For example, if A & B is true, then A and B are both true. On the other hand the information we extract is enough to reconstruct the starting formula—if A and B are both true, then A & B is true too.

As another example, if ~(A & B) is true, then either ~A is true or ~B is true. On the other hand, if either ~A or ~B is true, then (in either case) ~(A & B) is true. The same holds for each of the other rules too.

Let's see if you understand how tree proof rules work. Consider the following question.

FOR YOU: Suppose we define $A \star B$ as "neither A nor B." Which pair of rules is appropriate for $A \star B$?



ANSWER: (c) If $A \star B$ is true, then neither A nor B is true, so ~A or ~B are both true. If $A \star B$ is not true, then either A is true or B is true.

The most important condition in the development of trees is the distinction between closed and open branches.

CLOSURE: A branch of a tree is *closed* when it contains a formula and its negation. It is *open* otherwise.

Given the tree rules, we develop a tree, step by step, starting from the formulas at the top of the tree, and working downards, adding the result of processing a formula to each open branch in which the formula occurs.

PARTIALLY DEVELOPED TREES: A *partially developed tree* for a set Σ of formulas is a proof tree starting with the formulas in Σ, in which each formula in the tree is either in Σ, or follows from formulas higher up in the tree, by way of the tree rules.

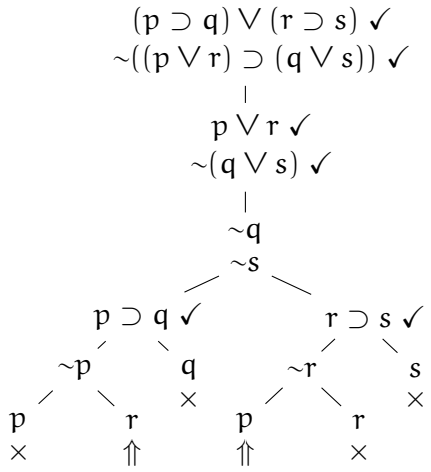
Finally, the process can *end*.

COMPLETED TREES: A *completed tree* is a *partially developed tree* where each complex formula in each open branch has been developed.

NOTATION: We write "$\Sigma \vdash$ " to say that a tree for Σ closes. We write "$\Sigma \vdash A$" for "$\Sigma, \sim A \vdash$"; in words "there is a proof from Σ to A".

So, we write "$\vdash A$" to say that a tree for $\sim A$ closes—and that A is a tautology. We write "$A \vdash$" to say that a tree for A closes—and that A is a contradiction.

EXAMPLE TREE 1: Here is a tree, which shows that $(p \supset q) \vee (r \supset s) \nvdash (p \vee r) \supset (q \vee s)$.



In this tree, there are two open branches, marked with the upward pointing arrow ⇑. In the left of these branches, the literals are $\sim q$, $\sim s$, $\sim p$ and r. Since this is a complete open branch we know that the valuation which sets $i(p) = 0$, $i(q) = 0$, $i(r) = 1$, $i(s) = 0$ (making each literal in that branch true) makes all the formulas in the branch true, and hence makes the premise $(p \supset q) \vee (r \supset s)$ true and the conclusion $(p \vee r) \supset (q \vee s)$ false.

It's good practice to check that your tree is correct by checking that the valuation indeed does make the premise true and the conclusion false. If this doesn't work, then you've made a mistake in the tree somewhere.

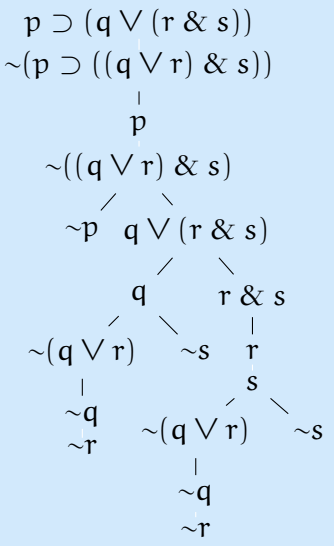
(p	⊃	q)	∨	(r	⊃	s)	(p	∨	r)	⊃	(q	∨	s)
0	1	0	1	1	0	0	0	1	1	0	0	0	0

The same goes for the other open branch. In that branch, the literals are $\sim q$, $\sim s$, p and $\sim r$. So, the valuation $i(p) = 1$, $i(q) = 0$, $i(r) = 0$, $i(s) = 0$ also makes the premise true and the conclusion false.

(p	⊃	q)	∨	(r	⊃	s)	(p	∨	r)	⊃	(q	∨	s)
1	0	0	1	0	1	0	1	1	0	0	0	0	0

This is an example of using a proof tree to construct a counterexample to an argument form. A closed tree counts as a proof. A complete open tree is not a proof, it provides a *counterexample*. In this case, the two open branches provide two different truth valuations which make the premise true and the conclusion false.

FOR YOU: Check this tree:



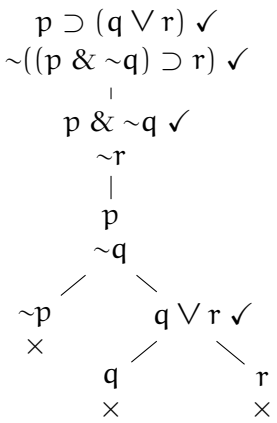
(*a*) There is a *mistake*.

(*b*) It's *correct*, and every branch *closes*.

(*c*) It's *correct*, and there's an *open branch*.

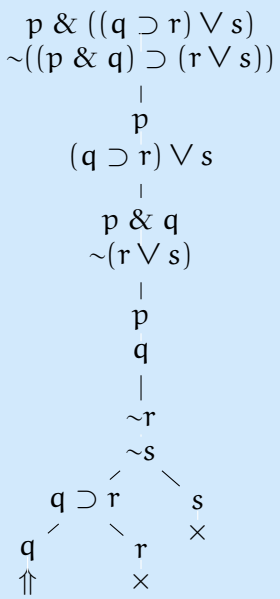
ANSWER: (*c*) The third branch from the left stays open, with literals p, q and ~s. (Notice that neither r nor ~r appear in the branch as single formulas.) You can check that a valuation with $i(p) = 1$, $i(q) = 1$, $i(s) = 0$ and either $i(r) = 0$ or $i(r) = 1$ (you choose!) makes the premise $p \supset (q \vee (r \,\&\, s))$ true and the conclusion $p \supset ((q \vee r) \,\&\, s)$ false.

EXAMPLE TREE 2: Here is another tree, this time a closed tree, which shows that $p \supset (q \vee r) \vdash (p \,\&\, \sim q) \supset r$.



This tree closes. So we cannot have the premise $p \supset (q \vee r)$ true and the conclusion $(p \,\&\, \sim q) \supset r$ false.

FOR YOU: Check this tree:



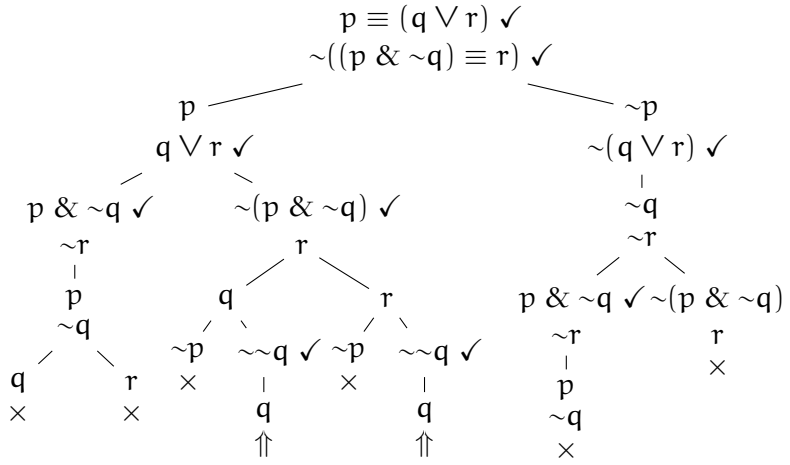
(*a*) There is a *mistake*.

(*b*) It's *correct*, and every branch *closes*.

(*c*) It's *correct*, and there's a *complete open branch*.

ANSWER: (*a*) There is a mistake. The last step, processing $q \supset r$ should result in the left branch containing $\sim q$, not q as it stands in this tree. That means that the left branch should close, since it contains q further up the tree. The tree should close. The argument is valid.

EXAMPLE TREE 3: This is an open tree, which shows that $p \equiv (q \vee r) \nvdash (p \,\&\, \sim q) \equiv r$.



This tree has a number of biconditional formulas, and these rules induce a great deal of branching. In this case, the tree has two open branches, both of which have the same literals, p, q and r. This means that a truth valuation i where $i(p) = i(q) = i(r) = 1$ will suffice to make the premise $p \equiv (q \vee r)$ true and the conclusion $(p \;\&\; \sim q) \equiv r$ false.

p	$\equiv$	$(q$	$\vee$	$r)$	$(p$	$\&$	$\sim$	$q)$	$\equiv$	r
1	1	1	1	1	1	0	0	1	0	1

The result is genuinely a truth valuation that makes the premise true and the conclusion false. The argument is invalid.

2.2.3 | WHY TREES AND TRUTH TABLES AGREE

We have seen two different ways to spell out the notion of validity. An argument is valid *from the point of view of models* if it has no counterexample (no valuation making the premises true and the conclusion false). An argument is valid *from the point of view of proofs* if a proof tree for it closes. We'll *show* that these two definitions of validity actually do amount to the same thing.

Our first step is will be to simplify what we'll try to show. Remember $\Sigma \vdash A$ if and only if $\Sigma, \sim A \vdash$—if a tree for Σ and $\sim A$ closes. We'll use format with $\models$, validity defined in terms of truth tables. We'll say that $\Sigma \models$ if and only if there is no valuation that makes every element of Σ true. Then it follows that $\Sigma \models A$ if and only if $\Sigma, \sim A \models$.

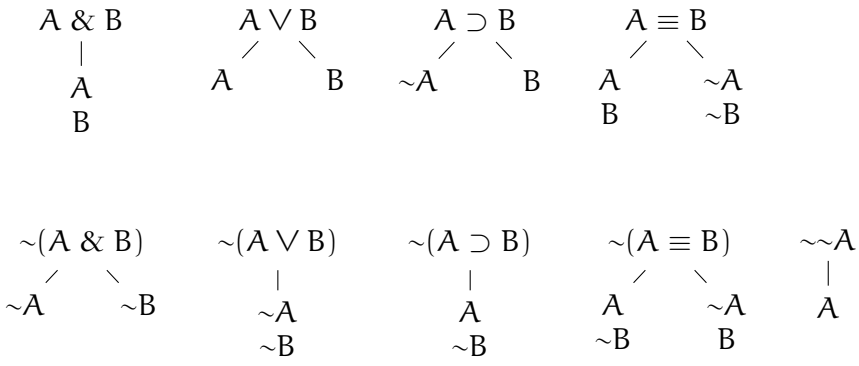
We'll show, then, that $\Sigma \vdash$ if and only if $\Sigma \models$. That is, a tree for Σ closes if and only if there is no valuation where each member of Σ is true. Or equivalently, $\Sigma \nvdash$ if and only if $\Sigma \not\models$. That is, a tree for Σ stays open if and only if there is some valuation where each member of Σ is true. That's what we'll try to show.

2.2.4 | SOUNDNESS: IF $\Sigma \not\models$ THEN $\Sigma \nvdash$

The first fact we'll show is the *soundness* fact. That is, if there is a valuation that makes Σ true, then a tree for Σ stays open. This is not difficult to prove. Start with a valuation i that makes Σ true. We'll call a set of formulas *safe* if every member of that set is true according to i. The

starting set Σ is safe, because that's how we started with $\mathfrak{i}$. We'll show that any tree for Σ will have an entire *branch* that is safe, which means that it will stay open. (Why? No *closed* branch is safe, since $\mathfrak{i}$ cannot make both a formula and its negation true.)

To show this, we show that if a partially developed tree has a safe branch, and we extend that branch by way of one of the tree rules, then one of the branches that results is safe, too. To show *that* we just need to check the rules. To remind you, here they are:



We just need to check that for each of these rules, if the starting formula is safe (true according to $\mathfrak{i}$) then so are the output formulas on one of the branches. I will explain a few cases, and leave the rest to you. Let's look at the disjunction and negated disjunction rules. Suppose $A \vee B$ is safe. This means that $\mathfrak{i}(A \vee B) = 1$ so either $\mathfrak{i}(A) = 1$ or $\mathfrak{i}(B) = 1$. If $\mathfrak{i}(A) = 1$, then the left branch (containing A) is safe. If $\mathfrak{i}(B) = 1$, then the right branch (containing B) is safe. In either case, the output formulas on one of the branches is safe, as we desired.

For the negated disjunction rule, suppose $\sim(A \vee B)$ is safe. This means that $\mathfrak{i}(\sim(A \vee B)) = 1$, so $\mathfrak{i}(A \vee B) = 0$ and hence, $\mathfrak{i}(A) = 0$ and $\mathfrak{i}(B) = 0$. This means that $\mathfrak{i}(\sim A) = 1$ and $\mathfrak{i}(\sim B) = 1$, so $\sim A$ and $\sim B$ are safe, and these are the formulas that result from $\sim(A \vee B)$ when you process it using the negated disjunction rule.

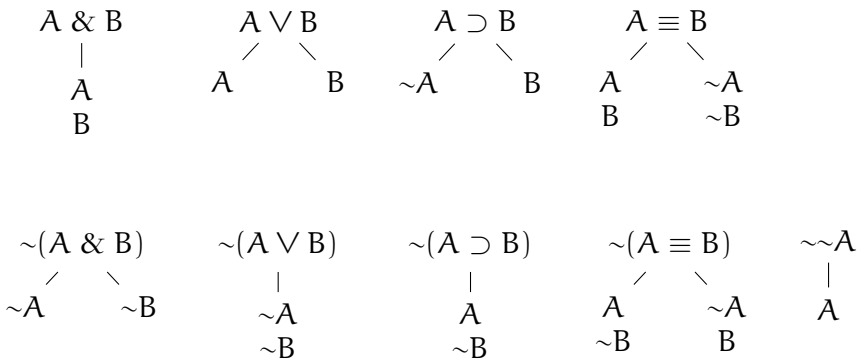
The same goes for the other rules, as you can check. Whenever you start with Σ and a valuation making every element in Σ true, then whenever you develop the tree, at least one branch remains safe, and therefore, the tree stays open. So, if $\Sigma \not\models$ then $\Sigma \not\vdash$.

2.2.5 | COMPLETENESS: IF $\Sigma \not\vdash$ THEN $\Sigma \not\models$

We want to show the converse, if $\Sigma \not\vdash$ then $\Sigma \not\models$—that if a tree for Σ stays open, then there is some valuation which makes Σ true. If a complete tree for Σ is open, then we construct a valuation from the literals on the branch. We choose a valuation that makes all the literals true—there is such a valuation, setting $\mathfrak{i}(p) = 1$ if p is on the branch, and $\mathfrak{i}(q) = 0$ if $\sim q$ is on the branch, and this works because the branch is open, it does not contain a formula and its negation. Then we show that any of the *complex* formulas on the branch are also true. Any complex formula is processed into its simpler parts, since the tree is complete, so we

will show that for any complex formula in the branch, if the results of processing that formula are true in the valuation, then so is the starting formula. We climb back up the rules in the tree, from output to input.

Let's look at the rules again, to check that this condition is satisfied for each rule—that if the ouptut of a rule is true according to a valuation, so is the input, so the valuation we construct out of the literals in the branch makes true each of the formulas in the branch.



Let's check the disjunction and negated disjunction rules, as before. Suppose $A \vee B$ is in the branch. That means it comes either from the A branch (working from the bottom) or the B branch. In either case, if A is true in the valuation, so is $A \vee B$, r if B is true, so is $A \vee B$.

For negated disjunction, suppose that $\sim(A \vee B)$ is in the branch. That means that $\sim A$ and $\sim B$ are both in the branch. This means that $\sim A$ and $\sim B$ hold in the valuation, so A and B are false in that valuation, and so is $A \vee B$, which means that $\sim(A \vee B)$ is true in the valuation.

The same holds for the other rules. This means that if $\Sigma \nvdash$ (a complete tree for Σ has an open branch) then $\Sigma \not\models$ (there is a valuation making Σ—and everything in that branch—true).

Combining these two results, we have $\Sigma \models$ if and only if $\Sigma \vdash$. That is, validity defined by models, and validity defined by trees, agree. We have two definitions which carve at the same joints.

Tree proofs are a useful and elegant proof system, with a long heritage [3]. For an important introduction to proof trees, read Smullyan's *First-Order Logic* [13]. For a more recent introduction of logic using trees, consult Howson's *Logic with Trees* [7].

PART II

Applications of Propositional Logic

ELECTRONIC ENGINEERING: COMBINATIONAL DIGITAL SYSTEMS

3

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

- Digital signals are sequences of 0 and 1s, and digital systems are machines which take digital signals as inputs and produce digital signals as outputs.
- AND, OR and NOT gates are the most elementary digital systems; combinational digital systems are those that can be built up from input signals using these gates, with signals named by atomic propositions $p, q, r, \ldots$ etc.
- Circuit diagrams for combinational systems function as templates for physical implementation of systems, so size/space matters.
- *Disjunctive Normal Form* (DNF) and what it means for a circuit diagram for a system whose outputs are expressed in DNF in terms of inputs.
- *Karnaugh Maps* (K-Maps) are compact truth tables with a very specific ordering; the method of "looping 1's" is a way to find *minimal* DNF formulas.
- Various representations of combinational systems: functional descriptions, truth tables, logic formulas, K-maps, circuit diagrams.

SKILLS

- Given a functional description of the output(s) of a combinational digital system in terms of the inputs, complete a truth table column or a K-map for each output of the system.
- Read a circuit diagram for a combinational digital system, and write a propositional logic formula expressing each output signal in terms of the inputs.
- Given a truth table for combinational digital system, determine a DNF formula for each output in terms of the inputs.
- Use a K-map to find a minimal DNF formula for a system output expressed in terms of the inputs.
- Use multiple K-maps to find a minimal DNF circuit for a multi-output system.

3.1 | DIGITAL SIGNALS AND SYSTEMS

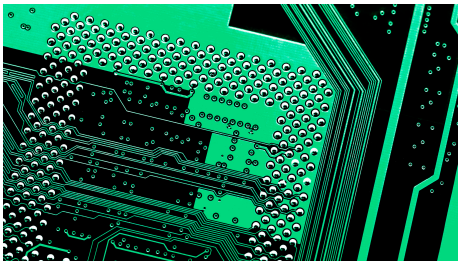
3.1.1 | INTRODUCTION

Digital systems is a core discipline within Electrical and Electronic Engineering, and it is grounded in logic.

For more than half a century, the world has witnessed the evolution of the digital revolution. We have seen sweeping changes to all aspects of our social and personal lives in virtue of digital computing and communication technology.

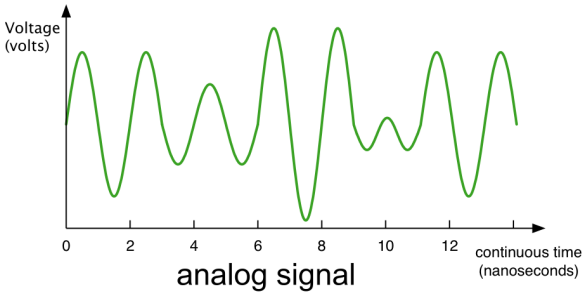
An educated person in the 21st century needs a basic understanding of digital technology; this section of the course aims to deliver that.

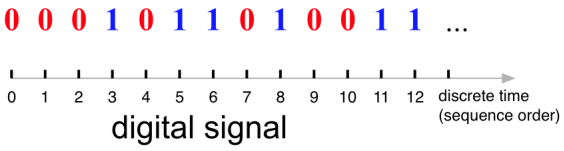
Digital electronic circuits are everywhere. They are in the obvious places like computers, internet servers, mobile phones and tablets, audio systems, digital televisions, GPS navigation, digital cameras, robots, and video games. They are in medical, industrial, transport and military equipment. They are in everyday items like clocks and cars, toasters and tea kettles. Take the cover off almost any piece of electric powered equipment and you will see the familiar green of digital electronic circuit boards.



(IMAGE CREDIT: Computer motherboard tracks by *Creativity103* [CC BY 2.0] http://www.flickr.com/photos/creative_stock/5228433146/)

ANALOG VERSUS DIGITAL SIGNALS: A familiar theme of the digital revolution is that old *analog* technologies are replaced by new *digital* successors. What is this analog/digital distinction?



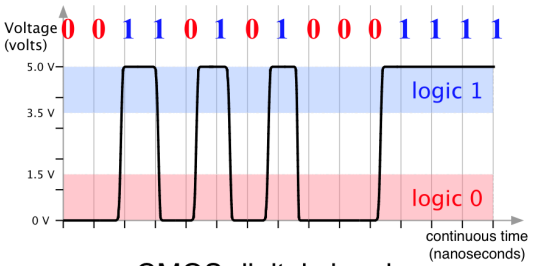


On the one side: *analog* physical quantities like temperature, pressure, sound, distance, and electric voltage, can vary over a continuous, real-number range of values, with the variation occurring in continuous, real time.

On the other side: *digital* signals are discrete sequences of 0s and 1s, with the variations in value occurring at discrete instances of time. All manner of complex *information* – text, voice, web pages, music, and video – can all be coded up in digital form as a stream of 0s and 1s.

The digital realm rests on a powerful *abstraction*. Digital systems processing sequences of 0s and 1s can be analysed and designed as abstract, logical systems. We can do this *independently* of how the system is physically implemented in an electronic circuit board. We can work at the level of 0s and 1s without needing to know any details about *how* the basic AND, OR and NOT gates are built out electronic switches, and about how digital circuits actually work.

PHYSICAL IMPLEMENTATION OF DIGITAL SYSTEMS: Engineers have *pragmatic* solutions to the problem of 'vagueness' of analog quantities versus 'sharpness' of digital/binary/2-valued world.



CMOS digital signal

At the level of physical implementation, digital circuits are in fact *analog*! The binary valued content of a digital signal is physically represented by a continuous time voltage signal that varies between a *high* band of values, near the maximum voltage, representing logic 1, and a *low* band of values near 0, representing logic 0. For the CMOS family of integrated circuits, the maximum voltage is 5 volts, as shown in the graph. Transitions between the low and high bands occur very quickly, in much less time than the real time between discrete ticks of the digital clock. It is only at these discrete moments that the low/high band level or 0/1 content of a signal really matters.

The basic components of a digital circuit are built out of very small-scale electronic switches; these days, mostly *transistors* built out of layers of semiconductor material. When a switch is open, electrons can flow through and this corresponds to logic 1. When a switch is closed, the flow of electrons stops and this corresponds to logic 0. These two YouTube videos illustrate the construction and operation of MOSFET transistors if you are interested:

http://www.youtube.com/watch?v=v7J_snw0Eng
http://www.youtube.com/watch?v=QO5FgM7MLGg

The core takeaway message is that *no matter how* digital systems are actually physically constructed, we can abstract away from those details and focus, for the purposes of analysis and design, on the logical character of digital systems and their processing of 0s and 1s.

So, welcome to the digital world!! Let's get going, and see logic at work in digital signals and systems.

3.1.2 | DIGITAL/BINARY SIGNALS AND SYSTEMS

The *Binary* or base-2 numbers are just 0 and 1. The word "bit" is a contraction of "binary digit".

A *digital signal* is any binary sequence (of finite or infinite length); that is, any sequence of 0s and 1s.

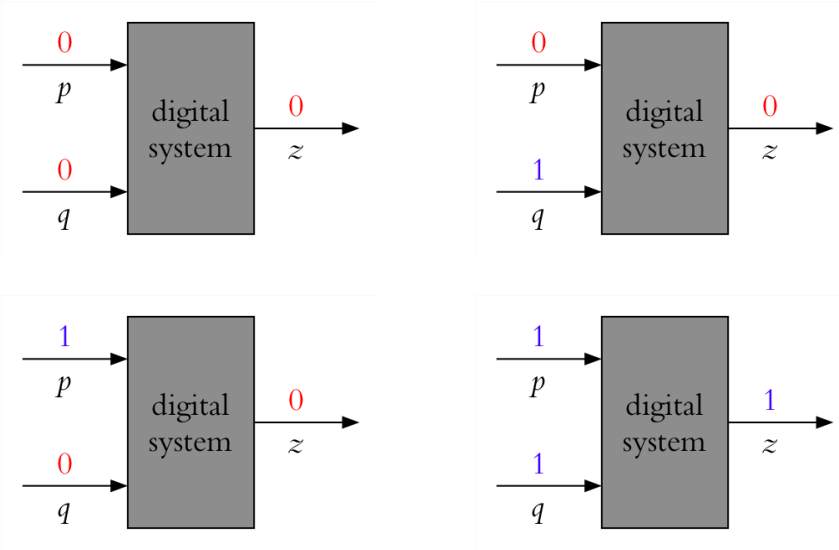
For example, one digital signal:

0 0 1 0 1 1 1 0 0 1 0 0 0 0 1 0 1 ...

Abstractly, a *system* is an abstract machine which accepts inputs and produces outputs.

A *digital system* is any device that accepts digital signals as inputs, and produces digital signals as outputs.

We label input and output signals with atomic proposition symbols, using lowercase letters.



The system above has inputs p and q, and an output z. Thinking of z as a function of, or determined by, the inputs p and q, where have we seen this truth-functional behaviour before?

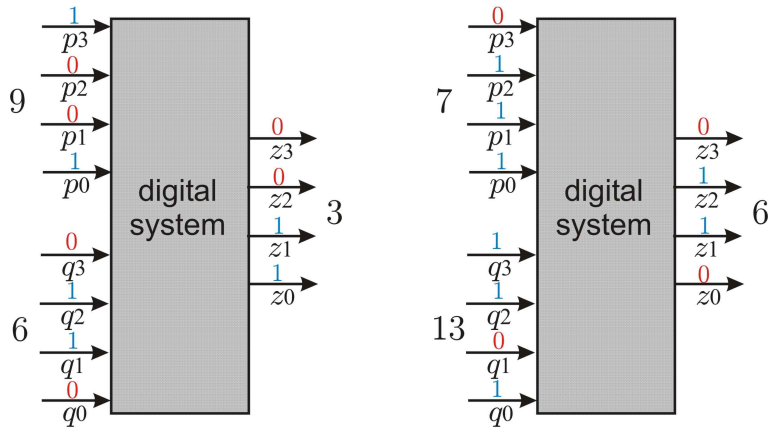
Yes, it is the AND connective: the output z is 1 if and only if both the inputs p and q are 1. So we have the bi-conditional $z \equiv (p \ \& \ q)$ expressing the output z in terms of the inputs p and q. This system is called an *AND gate*; it will be formally defined later in this section, along with *OR gates* and *NOT gates*. Each of these are elementary digital systems whose truth-functional behaviour corresponds exactly to the corresponding connective of propositional logic.

Clearly, content-rich signals like audio, video and image streams need to work with "chunks" of data bigger than 1 bit. A *nibble* is 4-bits; a *byte* is 8-bits; and a *word* is 16-bits, 32-bits, or 64-bits, depending on the width of the memory bus.

Working with 4-bit chunks, nibbles, we can use 4 1-bit signals, and get the binary or base-2 coding of all the whole numbers between 0 and 15. Here, our systematic ordering of truth table rows comes in handy, as we get the 16 rows in numeric order.

n	p3	p2	p1	p0
0	0	0	0	0
1	0	0	0	1
2	0	0	1	0
3	0	0	1	1
4	0	1	0	0
5	0	1	0	1
6	0	1	1	0
7	0	1	1	1
8	1	0	0	0
9	1	0	0	1
10	1	0	1	0
11	1	0	1	1
12	1	1	0	0
13	1	1	0	1
14	1	1	1	0
15	1	1	1	1

Consider a digital system with two 4-bit inputs and one 4-bit output. Decimal base-10 is easier to read than binary base-2, so we include both. Notice that if we had to write out a truth table for this system, it would have 2^8 which is 256 rows in its truth table.

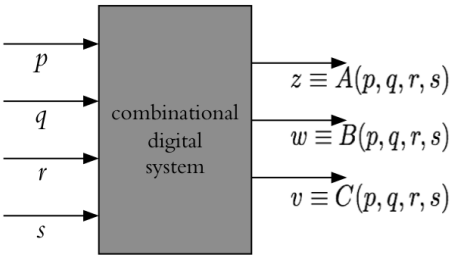


From inputs 9 and 6, this system produces output 3. From inputs 7 and 13, this system produces output 6. From this small sample of 2 out of 256 rows of the truth table, we might guess that the system computes the "magnitude of the difference" between the two 4-bit input numbers.

3.1.3 | COMBINATIONAL DIGITAL SYSTEMS

We will be focusing here on *Combinational* digital systems, which are those that can be built out of the basic components of AND, OR and NOT gates. We will formally introduce these basic components in the next lesson. For our purposes here, we can use an equivalent definition.

A *combinational digital system* is a digital system such that each output can be expressed as a propositional logic formula in terms of the inputs (using only AND, OR and NOT).



In our sample, output z can be characterised by a formula A(p, q, r, st) built out of the atoms p, q, r, and s. Likewise for outputs *w* and *v*. We can in fact allow the material conditional and biconditional connectives as well, since propositional formulas containing these can always be re-written as logically equivalent formulas containing only AND, OR and NOT.

A basic task is to take a natural language specification of system behaviour, and translate that into propositional formulas describing the output signals as a function of the input signals.

A car alert is to be activated exactly when the ignition is turned on and either a door is open or an occupied seat does not have a seat belt buckled.

Let the output signal for the alert be z, with z taking value 1 when the alert is active. Input signal p is 1 when the ignition is turned on; input q is 1 when all car doors are closed; and input r is 1 when all occupied seats have their seat belts buckled.

Which (one or more) of these propositional formulas correctly describe z as a function of p, q and r:

(a) $z \equiv (p \,\&\, (q \vee r))$ **(b)** $z \equiv (p \,\&\, (\sim q \vee \sim r))$

(c) $z \equiv ((p \,\&\, q) \vee (p \,\&\, r))$ **(d)** $z \equiv ((p \,\&\, \sim q) \vee (p \,\&\, \sim r))$

(a) $z \equiv (p \,\&\, (q \vee r))$

(b) $z \equiv (p \,\&\, (\sim q \vee \sim r))$

(c) $z \equiv ((p \,\&\, q) \vee (p \,\&\, r))$

(d) $z \equiv ((p \,\&\, \sim q) \vee (p \,\&\, \sim r))$

It is ~q which means that a car door is open, and ~r means that an occupied seat does not have a seat belt buckled. So the correct answers are **(b)** and **(d)**. They are equivalent because of the *distribution* of AND over OR.

Alternative task: from the functional description of the car alert system, complete the truth table for the output z as a function of p, q, r.

Solution:

p	q	r	z
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	1
1	0	1	1
1	1	0	1
1	1	1	0

Combinational digital systems are in fact a sub-class of digital systems. A crucial characteristic of combinational systems is that they are *memoryless*. This is in contrast with a bigger class called *Sequential* digital systems. These systems contain memory, to keep track of past values of input signals. Roughly speaking, sequential systems are combinational systems together with delay components, feedback loops, and one or

more digital clocks. A basic delay component allows the holding of 1 bit of data for 1 tick of a digital clock.

COMBINATIONAL DIGITAL SYSTEMS: memoryless. The current value of each output signal is a function of the current values of the input signals – and does *not* depend on past values of input signals. In this course, **Logic: Language & Information 1**, we focus on combinational systems.

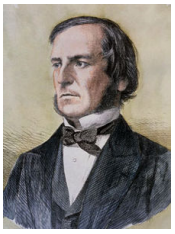
SEQUENTIAL DIGITAL SYSTEMS: contain memory via combinational and *delay* components in *feedback loops*, plus *digital clock*. Memory registers are used to keep track of past values of input signals, with values in memory recorded by internal state signals, together with clock input signal to keep time. In the next course, **Logic: Language & Information 2**, we introduce sequential systems and their analysis using more expressive logics.

JOBS DONE BY COMBINATIONAL SYSTEMS: There is plenty to keep us occupied just with combinational digital systems, as there are a great many tasks that can be performed by combinational components alone. You may have encountered the term *codec* – which is shorthand for encoding-decoding – perhaps when changing the format of a digital video file. Pressing a button on a key pad and translating that in a 4-bit code is another example of encoding. Selection and distribution of digital signals can all be done using combinational components. Binary arithmetic is another important task that can be done combinationally: addition, subtraction, multiplication, and more, with some further binary coding of negative integers and real numbers up to a fixed level of precision. We will come back to some of these tasks during the following sections §3.2, §3.3 and §3.4, and in the practice and graded quizzes.

- **Encoding and Decoding** of digital signals. Tech term: **codec** = en**co**ding-**dec**oding. E.g. of encoding: buttons on mobile phone: press button "**9**"on keypad, and 4-bit encoder produces 1001 as output.
- **Selecting and Distributing** digital signals. Tech term: a multiplexer (MUX) is a data selecter, and a demultiplexer (DEMUX) is a data distributer.
- **Binary arithmetic:** operations of addition, subtraction, multiplication, and more, with binary coding of negative numbers and finite-precision real numbers.
- **Comparison and classification:** for example, input two 4-bit binary numbers $a = a3\ a2\ a1\ a0$ and $b = b3\ b2\ b1\ b0$, and output 0 or 1 to questions "$a > b$?", "$a = b$?" or "$a < b$?" via three output signals.

3.1.4 | HISTORY OF DIGITAL SYSTEMS

Here are some of the significant figures in the early history of digital systems.

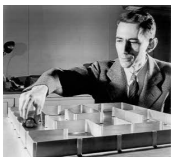


George Boole (1815–1864): founder of modern propositional logic. Symbolic methods of algebra applied to logic. 1847: *The Mathematical Analysis of Logic*;
1854: *An Investigation of the Laws of Thought*.

(IMAGE CREDIT: *George Boole in color* by Author Unknown (c. 1860) [Public domain], via *Wikimedia Commons*. http://en.wikipedia.org/wiki/George_Boole.)

Charles Sanders Peirce (1839-1914): truth tables, not-or/nor (Peirce arrow); as early as 1895, used Boolean algebra and propositional logic to analyse relay and switching circuits.

(IMAGE CREDIT: *Charles Sanders Peirce* by Author Unknown (c. 1900) [Public domain], via *Wikimedia Commons*. http://en.wikipedia.org/wiki/Charles_Sanders_Peirce.



Claude Elwood Shannon (1916-2001): Founder of the Digital Revolution.
1937: *A Symbolic Analysis of Relay and Switching Circuits* (MIT master's thesis): reinvented and elaborated Peirce's idea of using Boolean algebra and propositional logic to analyse relay and switching circuits.
1948: *A Mathematical Theory of Communication*: established the field of *information theory*, a branch of applied mathematics within which one can precisely formulate what is means for a digitized signal to adequately and correctly capture the information content of an analog source signal, and what is means to reconstruct an analog signal out of a digital stream.

(IMAGE CREDIT: *Claude Shannon* and his electromechanical mouse called "Theseus". Reprinted with permission of Alcatel-Lucent USA Inc. http://www.landley.net/history/mirror/pre/shannon.html.

Alan Turing (1912-1954): his 1936 paper formulating concepts of algorithm and computation provided essentially a blue-print for stored memory general-purpose computers.

Victor Shestakov (1907-1987): proposed analysis of relay and switching circuits using Boolean algebra in 1935, predating work of Shannon, but not published until 1941.

Akira Nakashima (1908–1970): series of papers 1934–1938 on switching circuits, with some anticipation of Shannon's work.

Further highlights in the history of digital systems:

- 1937: first binary adder.
- 1947: first point-contact transistor.
- 1958: first integrated circuit.
- 1969: birth of ARPANET, precursor to internet.
- 1991: first digital mobile phones.
- 2008: first memristor produced.

A *memristor* is a new type of electronic component that is smaller than and less energy-hungry than a transistor. A memristor can be configured to be *both* a memory bit storage device *and* a logic gate, with the logic gate functioning as the *conditional* or *material implication* connective. So memristors can implement both sequential and combinational digital systems. Commercial production of memristors is still some years away, but their potential is significant.

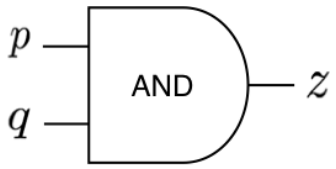
3.2 | Logic gates and logic circuits

3.2.1 | The basic logic gates

In this section, we look inside the "black box" of combinational digital systems. We will get to see the familiar propositional connectives of AND, OR and NOT in a different light. Instead of being the "glue" that connects propositions, the logic gates of digital systems are elementary systems in their own right, defining an input-output relationship on digital signals. Logic gates are the basic *building blocks* out of which complex combinational digital circuits are constructed. Make sure you are clear on the truth tables for AND, OR and NOT, from §1.2.2, as we don't repeat them here.

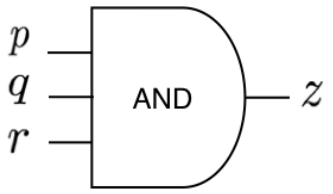
An *AND gate* is a memoryless digital system with two or more inputs and one output, with the property that the output has value 1 if and only if each of the inputs have value 1.

The symbol for an AND gate is characterised by the *rounded* output end (on the right) and the *straight* input side (on the left).



$$z \equiv (p \,\&\, q)$$

z ≡ (p & q)



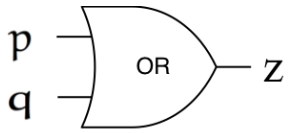
$$z \equiv (p \,\&\, q \,\&\, r)$$

z ≡ (p & q & r)

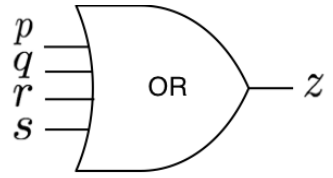
In the notation of Boolean Algebra standard in the Digital Systems literature, A & B is written in product form, as A · B or just AB, since AND is the logical *product* operation.

An *OR gate* is a memoryless digital system with two or more inputs and one output, with the property that the output has value 1 if and only if at least one of the inputs have value 1.

The symbol for an OR gate is characterised by the *pointy* output end (on the right) and the *curved* input side (on the left).



z ≡ (p ∨ q)



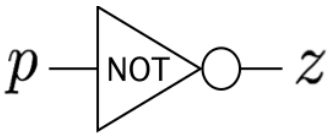
$$z \equiv (p \vee q \vee r \vee s)$$

z ≡ (p ∨ q ∨ r ∨ s)

In the notation of Boolean Algebra standard in the Digital Systems literature, A ∨ B is written in product form, as A + B, since OR is the logical *sum* operation.

A *NOT gate* or *inverter* is a memoryless digital system with one input and one output, with the property that the output has value 1 if the input has value 0, and the output has value 0 if the input has value 1.

The symbol for a NOT gate is characterised by a triangle pointing to the right with an inversion *bubble* on output end (on the right).



$$z \equiv \sim p$$

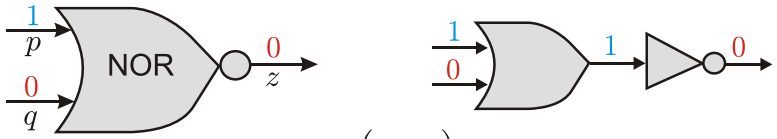
z ≡ ~p

In the notation of Boolean Algebra standard in the Digital Systems literature, ~A is written as A′ or $\overline{A}$.

The combinational digital systems are exactly those that can be constructed out of AND, OR or NOT gates.

3.2.2 | MORE LOGIC GATES

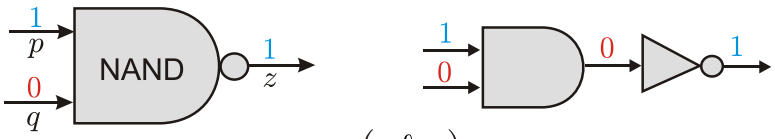
There are several other elementary logic gates of interest.



$$z \equiv \sim(p \vee q)$$

A *NOR gate* is a memoryless digital system with two or more inputs and one output, with the property that the output has value 1 if and only if each of the inputs have value 0. Equivalently, the output of a NOR gate has value 0 if and only if at least one of the inputs have value 1.

The NOR operation is historically known as the Peirce arrow (Peirce: 1881).



$$z \equiv \sim(p \& q)$$

A *NAND gate* is a memoryless digital system with two or more inputs and one output, with the property that the output has value 1 if and only if at least one of the inputs have value 0. Equivalently, the output of a NAND gate has value 0 if and only if each of the inputs have value 1.

The NAND operation is historically known as the Sheffer stroke (Sheffer: 1913).

In fact, all the classical propositional connectives (&, ∨, ~, ⊃ and ≡) can be expressed in terms of NAND alone, or NOR alone.

With (A ↓ B) meaning ~(A ∨ B) for NOR, we have:

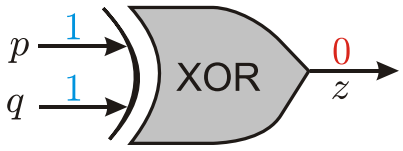
$$\begin{aligned} \sim A &\equiv (A \downarrow A) \\ (A \,\&\, B) &\equiv ((A \downarrow A) \downarrow (B \downarrow B)) \\ (A \vee B) &\equiv ((A \downarrow B) \downarrow (A \downarrow B)) \end{aligned}$$

With (A|B) meaning ~(A & B) for NAND, we have:

$$\begin{aligned} \sim A &\equiv (A|A) \\ (A \vee B) &\equiv ((A|A)|(B|B)) \\ (A \,\&\, B) &\equiv ((A|B)|(A|B)) \end{aligned}$$

This means we can build *any combinational digital system* using just one kind of logic gate or one kind of Lego-like constructor block.

XOR EXCLUSIVE-OR GATE: One further combinational gate is **EXCLUSIVE OR**, written XOR:



$$z \equiv \sim(p \equiv q)$$

An *XOR gate* is a memoryless digital system with two inputs and one output, with the property that the output has value 1 if and only exactly one of the inputs has value 1.

The XOR gate has several equivalent descriptions in propositional logic:

$$\begin{aligned} z &\equiv \sim(p \equiv q) \\ z &\equiv (p \,\&\, \sim q) \vee (\sim p \,\&\, q) \end{aligned}$$

XOR gates are useful because they behave essentially the same as *binary addition*, the natural + operation on binary numbers.

Consider a combinational digital system with three inputs p, q and r, and one output z, that behaves like a 3-input XOR gate. The output z is 1 if and only if exactly one out of three of the inputs has value 1. Which (one or more) of these propositional formulas correctly describe z as a function of p, q and r. It will be useful to make us of a truth table for the system:

p	q	r	z
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	0
1	0	0	1
1	0	1	0
1	1	0	0
1	1	1	0

(a) $z \equiv \sim(p \equiv \sim(q \equiv r))$

(b) $z \equiv (p \vee q \vee r)$

(c) $z \equiv ((p \;\&\; \sim q \;\&\; \sim r) \vee (\sim p \;\&\; q \;\&\; \sim r) \vee (\sim p \;\&\; \sim q \;\&\; r))$

(d) $z \equiv ((\sim(p \equiv q) \;\&\; \sim r) \vee (\sim p \;\&\; \sim(q \equiv r))$

The correct answers are:
(c) $z \equiv ((p \;\&\; \sim q \;\&\; \sim r) \vee (\sim p \;\&\; q \;\&\; \sim r) \vee (\sim p \;\&\; \sim q \;\&\; r))$
(d) $z \equiv ((\sim(p \equiv q) \;\&\; \sim r) \vee (\sim p \;\&\; \sim(q \equiv r))$
(a) and **(b)** are both wrong because they both give z is 1 when all three inputs are 1.

3.2.3 | CIRCUIT DIAGRAMS

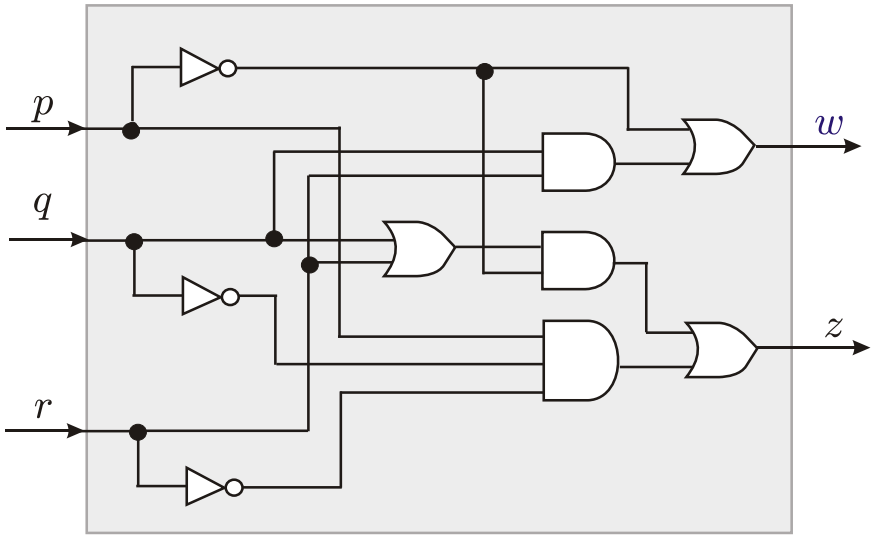
We have now introduced the basic systems, the logic gates. Next, we need to set out some rules for putting together these basic building blocks to create bigger, more complex systems.

- Input and output signal wires are labelled with signal names (we'll use p, q, r, z, *w*, *v*, etc, as for atomic propositions).
- Basic constructors: AND, OR, NOT gates (or NAND only, or NOR only).
- Connect output of one gate with inputs of further gates.
- Can "branch" a signal wire in *feed-forwards* direction, sending same signal as input to 2 or more other components; branch shown

with bold ●. But no *feedback loops* allowed here (come back for **Logic: Language & Information 2** and sequential digital systems).

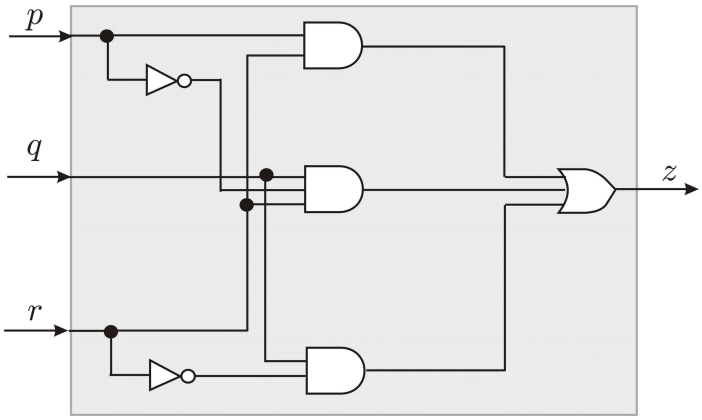
- From a propositional logic formula characterising an output signal in terms of input siganls, of the form $z \equiv A(p, q, r, \ldots)$, we get a recipe for constructing a circuit diagram for the system. Start at atomic propositions for inputs, and build up.
- The layout in 2-dimensions of a circuit diagram provides a template for a physical implementation of the device. The signal lines show where to put physical wires and the logic gates are typically implemented using *analog* transistor devices. Regardless of how logic gates are physically implemented, *space matters*: we want and need to design circuits that have a minimal number of gates and minimal-sized gates in their number of inputs. This will be the content of the last section on combinational digital systems.

MULTI-OUTPUT CIRCUITS: The basic gates are all 1-output systems, but when composed together according to the circuit diagram construction rules, it is easy to create systems with multiple outputs. Using feed-forward branching, any component can have its output branched and sent via other gates to separate outputs.

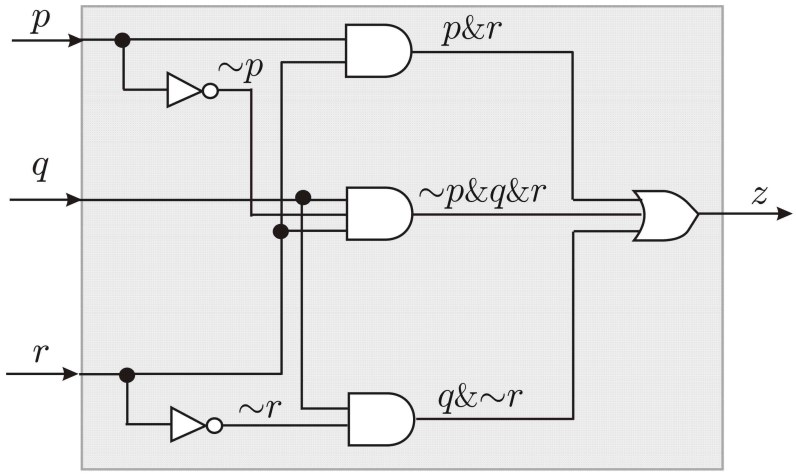


$$z \equiv (p\&{\sim}q\&{\sim}r)\vee({\sim}p\&(q\vee r))$$
$$w \equiv {\sim}p \vee (q\&r)$$

READING CIRCUIT DIAGRAMS: Here, we are interested in developing the skill of *reading* circuit diagrams for combinational systems, in order to extract a logic formula for each output expressing it in terms of the inputs. A systematic way of doing it is by labelling each of the intermediate signal wires with appropriate logic formulas, in order to capture the build-up to the output. Let's start with an example.

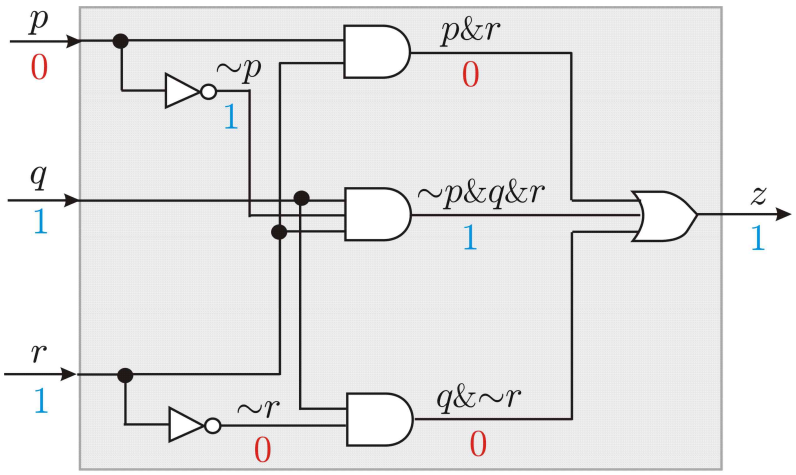


Task: Reading the circuit diagram above, express the system output z as a propositional logic formula in terms of inputs p, q and r, and label all intermediate signal wires with appropriate logic formulas.



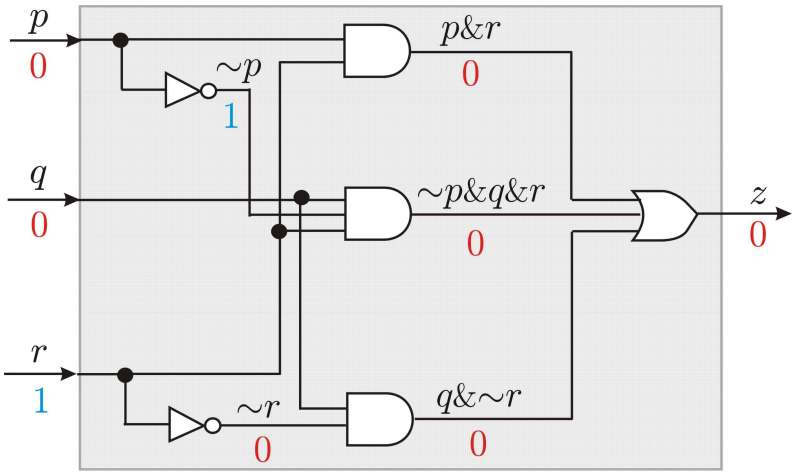
Answer: $z \equiv (p\&r)\vee(\sim p\&q\&r)\vee(q\&\sim r)$

By assigning some bit-values to the input signals p, q and r, we can see them propagate through the circuit, using the intermediate signal formulas, to end up with the final bit-value for the output z. Here, the (p, q, r) input (0, 1, 1) results in z output 1.



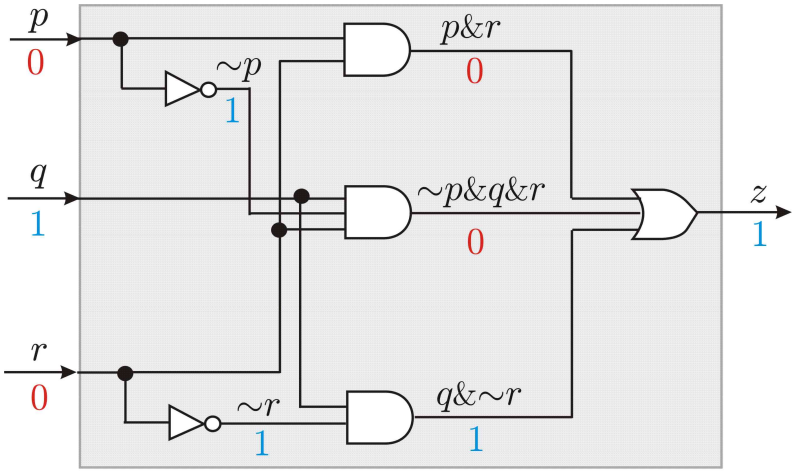
Answer: $z \equiv (p\&r)\vee(\sim p\&q\&r)\vee(q\&\sim r)$

While the (p, q, r) input (0, 0, 1) results in z output 0.



Answer: $z \equiv (p\&r)\vee(\sim p\&q\&r)\vee(q\&\sim r)$

And the (p, q, r) input (0, 1, 0) results in z output 1.



Answer: $z \equiv (p\&r)\vee(\sim p\&q\&r)\vee(q\&\sim r)$

Now it is your turn to practice reading a circuit diagram.

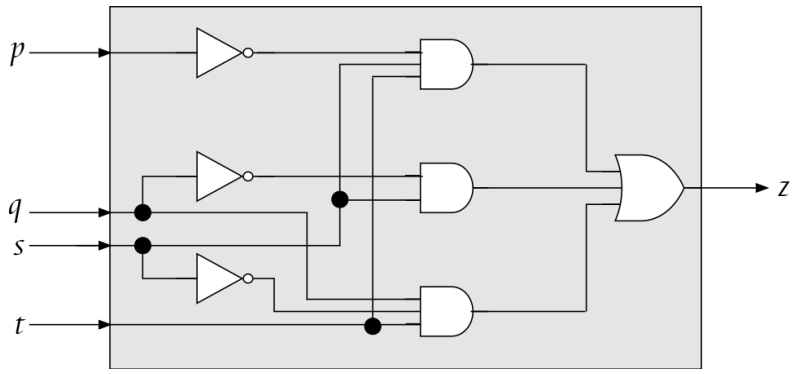
Task: Reading the circuit diagram below, which propositional logic formula correctly expresses the dependence of the output z on the input signals p, q, s and t:

(a) $z \equiv ((p \;\&\; \sim s \;\&\; t) \vee (\sim q \;\&\; \sim s) \vee (q \;\&\; s \;\&\; t))$

(b) $z \equiv ((\sim p \;\&\; s \;\&\; \sim t) \vee (\sim q \;\&\; s) \vee (\sim q \;\&\; s \;\&\; t))$

(c) $z \equiv ((\sim p \;\&\; s \;\&\; t) \vee (\sim q \;\&\; s) \vee (q \;\&\; \sim s \;\&\; t))$

(d) $z \equiv ((p \;\&\; s \;\&\; t) \vee (\sim q \;\&\; s) \vee (q \;\&\; \sim s \;\&\; t))$



Notice that each of the options have the right *shape*: requiring a 3-input OR gate preceded by three AND gates, with 3-inputs, 2-inputs and 3-inputs, respectively.

The correct answer is:

(c) $z \equiv ((\sim p \;\&\; s \;\&\; t) \vee (\sim q \;\&\; s) \vee (q \;\&\; \sim s \;\&\; t))$

The top AND gate gives $(\sim p \;\&\; s \;\&\; t)$; the middle 2-input AND gate gives $(\sim q \;\&\; s)$; and the bottom AND gate gives $(q \;\&\; \sim s \;\&\; t)$.

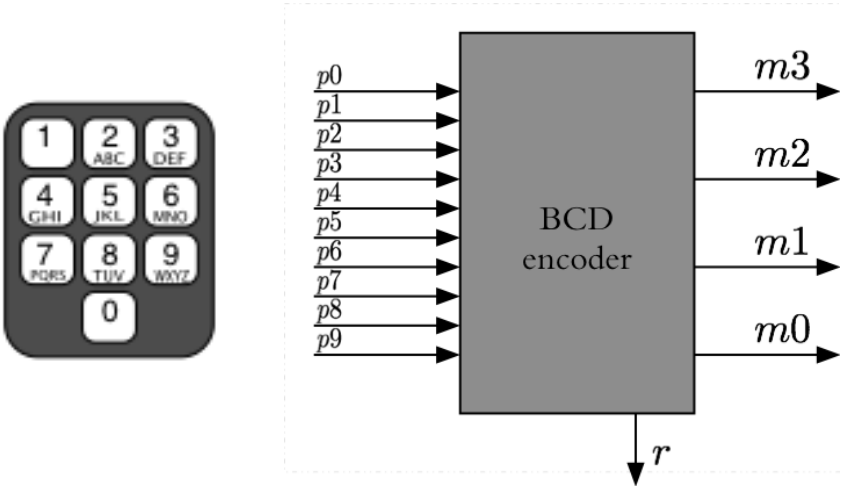
3.3 | TRUTH TABLES, LOGIC FORMULAS AND LOGIC CIRCUITS

In this section, we will examine several different *representations* of combinational digital systems – functional descriptions, truth tables, logic formulas and logic circuit diagrams – and how to transform from one representation to another. We begin with an example of an encoder system.

3.3.1 | DESIGN EXAMPLE: BCD ENCODER SYSTEM

Our first design example is for a very common digital system that we encounter every time we press a button on the keypad on our mobile phone. A BCD or *Binary Coded Decimal* encoder has the job of

outputting the 4-bit binary code for a decimal digit i between 0 and 9 when input i is the *only* active input. This corresponds to cleanly pushing the button for decimal digit i – without fumbling and also hitting another button on the keypad.



Functional description: the BCD encoder system has 10 inputs for decimal digits, p0, p1,p2, p3, p4, p5, p6, p7, p8, p9, and outputs 4-bit m = m3 m2 m1 m0 and a 1-bit error signal r such that: if exactly *one* on the inputs pi is active (value 1) then r is 0 and q is the 4-bit binary value of the decimal i, while if either *zero* or *two or more* of the inputs are active, then r is 1 and m = 0000.

With a total of 10 inputs, the full truth table for the BCD encoder system would have 2^{10} which is 1024 rows in its truth table. However, the output is *non-error* with r = 0 for only 10 out of 1024 of those rows – namely the ones shaded grey in the table below – that correspond to exactly one out of 10 of the decimal digit inputs pi being active. In those 10 rows, the 4-bit output m is the 4-bit binary coding of the decimal number i, for i from $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$.

p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	r	m3	m2	m1	m0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0	1
1	1	×	×	×	×	×	×	×	×	1	0	0	0	0
1	×	1	×	×	×	×	×	×	×	1	0	0	0	0

In all the other 1014 rows of the table, the error output r is 1 and the 4-bit output m is 0000. There is row 0 in which all of p0, p1, up to p9 are 0, which corresponds to *none* of the 10 keypad buttons being pressed, and then there are another 1013 rows of the table in which 2 or more of the pi inputs are 1, which corresponds to two or more of the 10 keypad buttons being pressed. We have indicated some of these *error cases* in the last two rows shown, using an × as a "don't care" symbol, meaning it can be either 0 or 1, it doesn't matter; the output is the same either way. Using the "don't care" notation, the 2nd last row actually covers all the 256 cases where both p0 and p1 are active, while the last row covers all the 256 cases in which both p0 and p2 are active – and of course, there is overlap, such as those cases in which all 3 of p0, p1 and p2 are active. We can similarly describe all the other combinations of two or more out of the 10 of the pi inputs being active.

In the BCD encoder system, it is clear that the output depends crucially on certain *combinations* of input values, namely those that describe when exactly one out of 10 of the decimal digit inputs pi are active, which corresponds to exactly one out of 10 of the keypad buttons being cleanly pressed. Let's introduce some intermediate signals, s0, s1, s2, s3, up to s9, which will be the outputs of 10-input AND gates. Signal s0 is given by the conjunction of p0 with ~p1, ~p2, and continuing up to ~p9, so that s0 is 1 exactly when p0 is the one and only decimal input active. Likewise, s1 is 1 exactly when p1 is the one and only of the 10 inputs active; s2 is 1 exactly when p2 is the one and only of the 10 inputs active; and so on up to s9.

$$
\begin{aligned}
s0 &\equiv (p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s1 &\equiv (\sim p0 \,\&\, p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s2 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s3 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s4 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s5 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s6 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s7 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, p7 \,\&\, \sim p8 \,\&\, \sim p9)\\
s8 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, p8 \,\&\, \sim p9)\\
s9 &\equiv (\sim p0 \,\&\, \sim p1 \,\&\, \sim p2 \,\&\, \sim p3 \,\&\, \sim p4 \,\&\, \sim p5 \,\&\, \sim p6 \,\&\, \sim p7 \,\&\, \sim p8 \,\&\, p9)
\end{aligned}
$$

Equipped with these intermediate signals s0, s1, s2, up to s9, we can set about determining a logic formula to characterise each of the 4 output bits, m3, m2, m1 and m0.

Start with the highest-order bit, m3. We want to characterise when m3 is true – when it has value 1. Looking at the truth table below, looking down the *column* for m3, we can see that m3 is 1 exactly when combination s8 is true or combination s9 is true, meaning keypad buttons 8 or 9 have been cleanly pressed. These are highlighted in dark gray. So we can conclude that m3 if and only if $(s8 \vee s9)$. In a circuit for the system, m3 is the output of a 2-input OR gate with inputs s8 and s9, which are in turn the outputs of 10-input AND gates.

	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	r	m3	m2	m1	m0
s0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
s2	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
s3	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
s4	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
s5	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
s6	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0
s7	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1
s8	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
s9	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

$m3 \equiv (s8 \vee s9)$ $\quad$ $m2 \equiv (s4 \vee s5 \vee s6 \vee s7)$

Next, the second-highest-order bit, m2. Here, we see from the truth table above that m2 is 1 exactly when one of keypad buttons 4, 5, 6, or 7 are cleanly pressed, as shown in the rows of the table shaded in light gray. So we get m2 if and only if $(s4 \vee s5 \vee s6 \vee s7)$. In the logic circuit, m2 is the output of a 4-input OR gate with inputs s4, s5, s6 and s7.

Continuing with the second-lowest-order bit, m1, we see from the truth table below that m1 is 1 exactly when one of the keypad buttons 2, 3, 6, or 7 are cleanly pressed, as shown in the rows of the table below shaded in dark gray. So we get m1 if and only if $(s2 \vee s3 \vee s6 \vee s7)$. In the logic circuit, m1 is the output of a 4-input OR gate with inputs s2, s3, s6 and s7.

	p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	r	m3	m2	m1	m0
s0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
s1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
s2	0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
s3	0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
s4	0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
s5	0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
s6	0	0	0	0	0	0	1	0	0	0	0	0	1	1	0
s7	0	0	0	0	0	0	0	1	0	0	0	0	1	1	1
s8	0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
s9	0	0	0	0	0	0	0	0	0	1	0	1	0	0	1

$m1 \equiv (s2 \vee s3 \vee s6 \vee s7)$ $\quad$ $m0 \equiv (s1 \vee s3 \vee s5 \vee s7 \vee s9)$

Finally, the lowest-order bit m0. We see from the truth table above that m0 is 1 exactly when one of the *odd-numbered* keypad buttons is cleanly pressed: 1, 3, 5, 7, or 9, as shown in the rows of the table above shaded in light gray. So m0 if and only if $(s1 \vee s3 \vee s5 \vee s7 \vee s9)$, and in a circuit, m0 is the output of a 5-input OR gate with inputs s1, s3, s5, s7 and s9.

The remaining output is the 1-bit error signal r which is active (has value 1) in 1014 out of 1024 rows of the truth table. Conversely, r has value 0 on 10 out of 1024 rows, namely those rows described by s0, s1, s2, up to s9. So r is 1 if and only if it is *not* the case that one of the ten

s_i signals is 1. Hence in a circuit for the system, we can use a 10-input OR gate followed by a NOT gate to produce the output r.

p0	p1	p2	p3	p4	p5	p6	p7	p8	p9	r	m3	m2	m1	m0
0	0	0	0	0	0	0	0	0	0	1	0	0	0	0
1	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0	1	0	0	0	0	0	0	0	0	0	0	0	0	1
0	0	1	0	0	0	0	0	0	0	0	0	0	1	0
0	0	0	1	0	0	0	0	0	0	0	0	0	1	1
0	0	0	0	1	0	0	0	0	0	0	0	1	0	0
0	0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	0	0	0	1	0	0	0	0	0	1	1	0
0	0	0	0	0	0	0	1	0	0	0	0	1	1	1
0	0	0	0	0	0	0	0	1	0	0	1	0	0	0
0	0	0	0	0	0	0	0	0	1	0	1	0	0	1
1	1	×	×	×	×	×	×	×	×	1	0	0	0	0

$$r \equiv \sim(s0 \vee s1 \vee s2 \vee s3 \vee s4 \vee s5 \vee s6 \vee s7 \vee s8 \vee s9)$$

3.3.2 | FROM TRUTH TABLE TO LOGIC FORMULA

The method used above in the BCD encoder example to derive logic formulas characterizing output signals can be generalized to arbitrary combinational digital systems represented by a truth table.

METHOD: given a truth table column for output z depending on inputs p_1, p_2, up to p_n (in a truth table with 2^n-many rows):

1. Look down column for output z and identify each of the rows in which output is 1.
2. For each such row, write down the size-n conjunction of inputs or negated inputs which uniquely describes that row: if input p_i is 0 on that row, include $\sim p_i$ in conjunction; if input p_i is 1 on that row, include p_i in conjunction.
3. Output z is equivalent to the disjunction of all these row-conjunctions; if there are m-many rows in which output z is 1, then it will be a size-m disjunction.

In this and the next lesson, when we refer to rows of a truth table by *number*, we use the decimal translation of the binary code for the input combination of the row. So for the 8-row truth table here, the rows are numbered 0, 1, 2, up to 7, with 3-bit binary codes 000, 001, and so on up to 111 for row 7, going down the table in standard order.

Consider the combinational system with three inputs p, q and r and one output z whose truth table is as follows:

row	p	q	r	z
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

Which of these formulas correctly describe output z as a function of inputs p, q and r:

(a) $z \equiv ((\sim p \,\&\, \sim q \,\&\, \sim r) \vee (\sim p \,\&\, q \,\&\, \sim r) \vee (\sim p \,\&\, q \,\&\, r))$
(b) $z \equiv ((\sim p \,\&\, \sim q \,\&\, r) \vee (\sim p \,\&\, q \,\&\, \sim r) \vee (p \,\&\, q \,\&\, r))$
(c) $z \equiv ((\sim p \,\&\, \sim q \,\&\, \sim r) \vee (\sim p \,\&\, q \,\&\, \sim r) \vee (p \,\&\, q \,\&\, r))$
(d) $z \equiv ((\sim p \,\&\, \sim q \,\&\, r) \vee (\sim p \,\&\, q \,\&\, \sim r) \vee (\sim p \,\&\, q \,\&\, r))$

The other three answers each have at least one of their disjuncts with an error.

(d) $z \equiv ((\sim p \,\&\, \sim q \,\&\, r) \vee (\sim p \,\&\, q \,\&\, \sim r) \vee (p \,\&\, \sim q \,\&\, r))$

The formula which correctly describes output z as a function of inputs p, q and r:

row	p	q	r	z	
0	0	0	0	0	
1	0	0	1	1	$(\sim p \,\&\, \sim q \,\&\, r)$
2	0	1	0	1	$(\sim p \,\&\, q \,\&\, \sim r)$
3	0	1	1	1	$(\sim p \,\&\, q \,\&\, r)$
4	1	0	0	0	
5	1	0	1	0	
6	1	1	0	0	
7	1	1	1	0	

3.3.3 | DISJUNCTIVE NORMAL FORM AND DNF CIRCUITS

When we apply our method to go from a truth-table output column to a logic formula characterising the output in terms of the inputs via row-conjunctions, the resulting formula is always of the same *shape* or *form*.

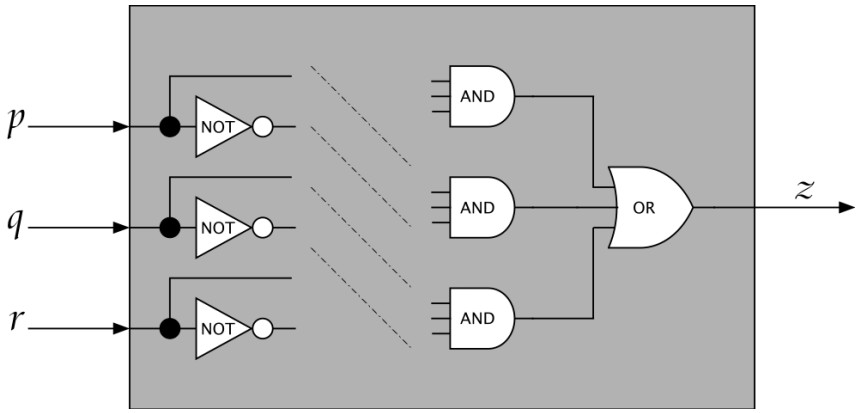
A logic formula A is in *disjunctive normal form* (DNF) if and only if A is a disjunction of conjunctions of *literals*, where a literal is either an atomic proposition or the negation of an atomic proposition; here, we allow size-1 disjunctions or conjunctions consisting of a single formula.

As is common in an inter-disciplinary subject like logic, one needs to be aware of differing terminology used in different disciplines for the same concepts. DNF is also called sum-of-products (SOP) form in the digital systems and Boolean algebra literature.

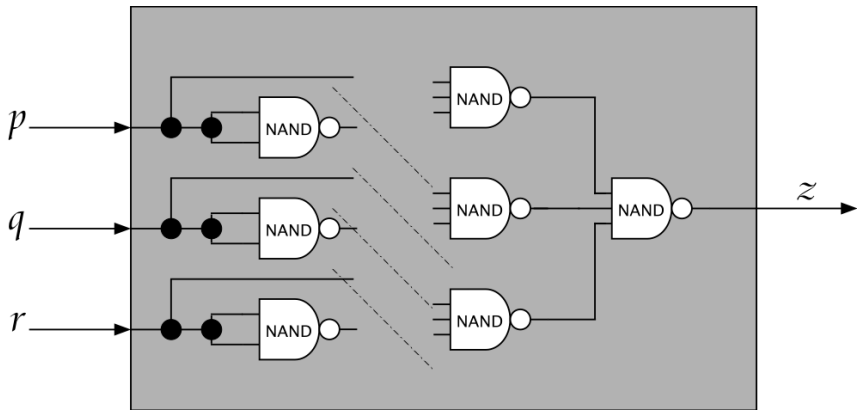
Fact: For every propositional logic formula A, there exists at least one (but usually many) formulas B such that B is in DNF and A is logically equivalent to B (that is, $A \equiv B$ is a tautology).

This fact can be proved using the method described above going from an input-output truth table to a logic formula by taking the disjunction of the row-conjunctions in which the output is true: the result is the canonical DNF for each output. In general, for a given propositional logic formula, there will be several different DNFs that are all logically equivalent.

When $z \equiv A(p, q, r)$, where $A(p, q, r)$ is in DNF and the circuit implements the formula $A(p, q, r)$, then the circuit diagram has a standard two-level structure.



A circuit in standard DNF (or SOP) form can be easily transformed into a NAND-only circuit by simply replacing all the gates with NAND gates.



Recall the digital system from the last exercise on deriving a logic formula characterisation of output from a truth table. With three rows of the table in which the output z was 1, we got three disjuncts, each of which were size-3 conjunctions:

row	p	q	r	z
0	0	0	0	0
1	0	0	1	1
2	0	1	0	1
3	0	1	1	1
4	1	0	0	0
5	1	0	1	0
6	1	1	0	0
7	1	1	1	0

$(\sim p \;\&\; \sim q \;\&\; r)$
$(\sim p \;\&\; q \;\&\; \sim r)$
$(\sim p \;\&\; q \;\&\; r)$

$$z \equiv ((\sim p \;\&\; \sim q \;\&\; r) \vee (\sim p \;\&\; q \;\&\; \sim r) \vee (\sim p \;\&\; q \;\&\; r))$$

This formula can in fact be simplified to a smaller DNF:

Remember there is a list of useful tautologies at the end of Chapter 1.

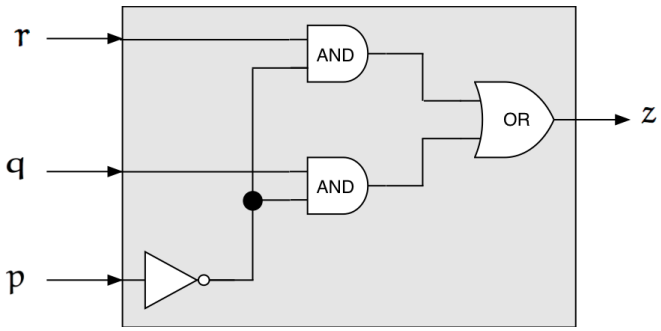
$$z \equiv ((\sim p \;\&\; q) \vee (\sim p \;\&\; r))$$

using a sequence of known tautologies, particularly the distribution of & over $\vee$.

A logic formula A is a *minimal DNF* if and only if A is in DNF and there does *not* exist a logic formula B such that B is in DNF, $B \equiv A$ is a tautology, and *either* B has fewer disjuncts than A, *or* B has the same number of disjuncts as A but fewer total number of literals than A.

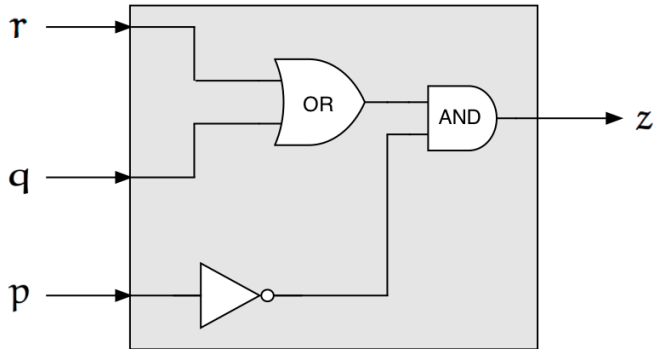
In general, minimal DNFs are not unique: there can be several different DNF's that are equally as "small".

We need to be clear about what a minimal DNF offers. Again, go back to the example system from the exercise. The DNF formula $((\sim p \;\&\; q) \vee (\sim p \;\&\; r))$ is a minimal DNF, so it has a minimal DNF circuit consisting of a 2-input OR gate fed by two 2-input AND gates.



$z \equiv ((\sim p \;\&\; q) \vee (\sim p \;\&\; r))$

However, if we apply the tautology expressing the distribution of $\vee$ over &, we can further re-express the formula as $(\sim p \;\&\; (q \vee r))$, resulting in a circuit with only one AND gate and one OR gate, as drawn below. So yes, this circuit is smaller in gate-count, but it is *not* in DNF form. Among other things, this non-DNF circuit *cannot* be readily transformed into a NAND-only circuit.



$z \equiv (\sim p \;\&\; (q \vee r))$

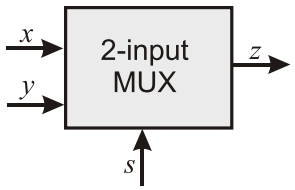
3.3.4 | DESIGN EXAMPLE: 2-IN MULTIPLEXOR (MUX) SYSTEM

We have now seen four different ways of representing a combinational digital system: a functional description, a truth table, a logic formula and a circuit diagram. We'll finish off section §3.3 by looking at a different type of combinational system, and use it to illustrate these four different representations.

A *multiplexor* – or MUX for short – is a system that *selects* between data input signals based on the value of a select control input, and then feeds that signal value through to the system output. When you switch your television set between different input signals, such as your digital TV receiver, your DVD player, and your desktop computer, you are choosing values for select inputs in a multiplexor. The opposite of a MUX is a DEMUX or *de-multiplexor*, which takes a single data input signal and distributes it to one of several outputs, depending on the value of one or more select inputs.

Functional description: the 2-in MUX system takes as input two *data input* signals x and y, together with a *select input* signal s, and produces as output one data signal z such that $z \equiv x$ if s is 0 and $z \equiv y$ if s is 1.

Block structure:



Truth table: We can complete the truth table directly from the functional description. When the select bit s is 0, then $z \equiv x$. This means that in the four rows where s is 0, we copy into the z column whatever value x has on that row.

row	x	y	s	z	
0	0	0	0	0	$z \equiv x$
1	0	0	1	0	
2	0	1	0	0	$z \equiv x$
3	0	1	1	1	
4	1	0	0	1	$z \equiv x$
5	1	0	1	0	
6	1	1	0	1	$z \equiv x$
7	1	1	1	1	

When the select bit s is 1, then $z \equiv y$. This means that in the other four rows where s is 1, we copy into the z column whatever value y has on that row.

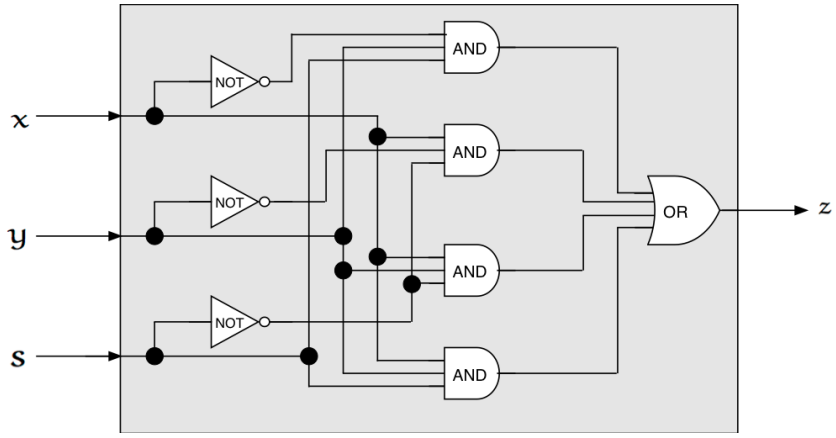
row	x	y	s	z	
0	0	0	0	0	
1	0	0	1	0	$z \equiv y$
2	0	1	0	0	
3	0	1	1	1	$z \equiv y$
4	1	0	0	1	
5	1	0	1	0	$z \equiv y$
6	1	1	0	1	
7	1	1	1	1	$z \equiv y$

DNF from truth table: We can now apply our method for deriving a DNF from a completed truth table, by focusing on the rows of the table in which the output z has value 1.

row	x	y	s	z	
0	0	0	0	0	
1	0	0	1	0	
2	0	1	0	0	
3	0	1	1	1	(~x & y & s)
4	1	0	0	1	(x & ~y & ~s)
5	1	0	1	0	
6	1	1	0	1	(x & y & ~s)
7	1	1	1	1	(x & y & s)

$$z \equiv ((\sim x \,\&\, y \,\&\, s) \vee (x \,\&\, \sim y \,\&\, \sim s) \vee (x \,\&\, y \,\&\, \sim s) \vee (x \,\&\, y \,\&\, s))$$

DNF circuit: A DNF formula gives a recipe for constructing a standard form DNF circuit for the system.



So, we have *one* logic circuit design in DNF form for the 2-in MUX system. But is it the *smallest* DNF circuit that will do the job of a 2-in MUX? Can we find another DNF circuit design that has *fewer* logic gates, and/or *smaller* logic gates, with fewer inputs?

To start answering these questions, let's look closely at the canonical DNF formula derived from the truth-table. Using instances of known tautologies, we can derive the following sequence of logical equivalences:

$$\begin{aligned}
z &\equiv (\sim x \,\&\, y \,\&\, s) \vee (x \,\&\, \sim y \,\&\, \sim s) \vee (x \,\&\, y \,\&\, \sim s) \vee (x \,\&\, y \,\&\, s) \\
z &\equiv (x \,\&\, y \,\&\, \sim s) \vee (x \,\&\, \sim y \,\&\, \sim s) \vee (x \,\&\, y \,\&\, s) \vee (\sim x \,\&\, y \,\&\, s) \\
z &\equiv (x \,\&\, \sim s \,\&\, y) \vee (x \,\&\, \sim s \,\&\, \sim y) \vee (y \,\&\, s \,\&\, x) \vee (y \,\&\, s \,\&\, \sim x) \\
z &\equiv (x \,\&\, \sim s \,\&\, (y \vee \sim y)) \vee (y \,\&\, s \,\&\, (x \vee \sim x)) \\
z &\equiv (x \,\&\, \sim s) \vee (y \,\&\, s)
\end{aligned}$$

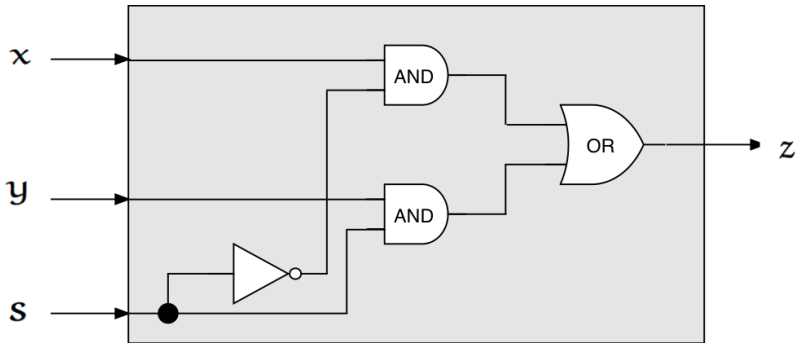
First, the original DNF formula can be re-arranged in the *order* of *disjuncts*, so that we first group together two row-conjunctions that are the same except for differing on y versus ~y, and then group together two row-conjunctions that are the same except for differing on x versus

~x. Pairs of row-conjunctions that differ only on one input are called *complementary*.

The next step is to re-arrange the order of the *conjuncts* within the *row-conjunctions*, to make clear that the first complementary pair of row-conjunctions are the same on x and ~s, while the second complementary pair are the same on y and s.

Finally, we can apply the tautology expressing the distribution of OR over AND, followed by another tautology expressing the cancellation of sub-formulas of the form (B ∨ ~B), and we end up with z equivalent to the much *simpler* DNF form (x & ~s) ∨ (y & s).

In the next section §3.4, we'll be able to see why this is a *minimal* DNF for the 2-in MUX system, and more generally, how we can systemically exploit this idea of *cancelling* and *simplifying* complementary pairs of row-conjunctions.



if we go back to the original functional description, it is easier to see the origins of this final, minimal DNF for the 2-in MUX system. Rephrased: the output signal z will be 1 exactly when either input x is 1 and ~s is 1, or input y is 1 and s is 1. The resulting minimal DNF circuit has an outer-level 2-input OR gate and two 2-input AND gates at the inner-level.

3.4 | MINIMIZING LOGIC CIRCUITS USING K-MAPS

We want a general and mechanical method for finding a *minimal DNF circuit* for a combinational system, having a two-level structure with:

- a minimal number of inputs to the OR gate at the outer level, and
- minimal-sized AND gates at the inner level.

3.4.1 | KARNAUGH MAP METHOD FOR DNF

The **Karnaugh-Veitch mapping method** is a systematic method for deriving a *minimal DNF* from the truth table for a propositional logic formula or combinational system.

It is a graphical (drawing-based) method that is practically limited to six inputs – but in this short introduction, we will only use Karnaugh-maps (K-maps) for at most four inputs.

HISTORY: 1952-53: E. W. Veitch and M. Karnaugh: develop graphical minimization method suitable for up to 6 atoms.
1956-57: philosopher W.V. Quine and mathematician E.J. McCluskey develop algorithm for finding minimal DNFs for any number of atoms; still implemented in electronic design automation (EDA) tools.

To begin with, a K-map should be thought of as a compact truth table – as just a graphical rearrangement. Take an 8-row truth table, such as the one for the 2-in MUX. We take those 8 rows and re-arrange them in an 8-cell 4-by-2 grid. Just for now, ignore the strange ordering of the cells in the 4-by-2 grid. As in a truth table, the cells are labelled according to their input combination. 2-bit codes for inputs x and y label the four *rows* of the grid, and 1-bit values for input s label the two *columns* of the grid. The bit-value entries *inside* the grid cells are the values for output z. In the truth table, output z is 1 in four rows: those numbered 3, 4, 6 and 7. highlighted in gray. In the K-map, there are likewise 4 cells containing bit-value 1: the cells 3, 4, 6 and 7, which correspond to 3-bit x y s codes 011, 100, 110 and 111.

8-row table for 2-in MUX

row	x	y	s	z
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	1
5	1	0	1	0
6	1	1	0	1
7	1	1	1	1

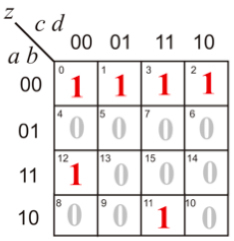
8-cell K-map for 2-in MUX

Take a 16-row truth table for s combinational system with four inputs a, b, c and d, and one output z. We take those 16 rows and rearrange then into a 16-cell 4-by-4 grid. Keep ignoring the strange ordering for now; we will come back to it. As in a truth table, the cells are labelled according to their input combination. 2-bit codes for inputs a and b label the 4 *rows* of the grid, and further 2-bit codes for inputs c and d label the 4 *columns* of the grid. Again, the bit-value entries *inside* the grid cells are the values for output z. In the truth table, output z is 1 in six rows: those numbered 0, 1, 2, 3, 11, and 12. In the K-map, there are likewise six cells containing bit-value 1: the cells 0, 1, 2, 3, 11, and 12.

16-row truth table

row	a	b	c	d	z
0	0	0	0	0	1
1	0	0	0	1	1
2	0	0	1	0	1
3	0	0	1	1	1
4	0	1	0	0	0
5	0	1	0	1	0
6	0	1	1	0	0
7	0	1	1	1	0
8	1	0	0	0	0
9	1	0	0	1	0
10	1	0	1	0	0
11	1	0	1	1	1
12	1	1	0	0	1
13	1	1	0	1	0
14	1	1	1	0	0
15	1	1	1	1	0

16-cell K-map for same



At this stage, we can see that a K-map contains all and only the information contained in the corresponding truth table. It is just a rather perculiar *re-ordering* and *re-arrangement* of a truth table.

This very peculiar ordering of cells in a K-map is designed so that any two adjacent cells reference rows of the truth table that differ only on one atom.

The *Gray code* order on 2-bits is the cyclic order:

$$00 \to 01 \to 11 \to 10 \to 00.$$

It has the property that any pair of adjacent 2-bit values will differ only on one of those two bits. This code is used to index the size-4 side of a K-map grid.

In the 2-dimensional K-map, we have several different ways in which cells can be *adjacent*: there are:

- *vertically adjacent* cells,
- *horizontally adjacent* cells,
- *wrap-around vertically adjacent* cells, and
- *wrap-around horizontally adjacent* cells.

The Karnaugh method to find minimal DNFs involves drawing *loops* around 1s in the K-map. We start by identifying all the 1s in the K-map – just like identifying all the 1s in the output column of the truth table. We then try to *cover* all the 1s in the K-map using *loops* of 2, 4, 8 or 16 *pairwise-adjacent* cells containing 1. *Loops* or groupings of pairwise-adjacent cells must be of size a power of 2, so 1 (which is 2^0) is OK, as well as 2, 4, 8 or 16. The last resort is a loop of size 1.

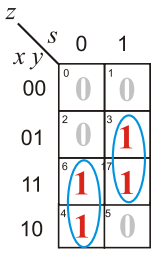
The *Karnaugh map* or *K-map* method to find minimal DNFs requires:

- all 1s to be covered by some loop, of size 1, 2, 4, 8, or 16;
- always use as large a loop as possible; and
- use as few loops as possible to cover all 1s.

Looping is a graphical way of describing simplification of "complementary pairs" of truth table row-conjunctions that differ only on one atom. A size-1 loop will be the best available if there is a cell with a 1 that has no adjacent neighbouring cells with a 1.

- the number of loops = the number of inputs to OR gate;
- each loop corresponds to an AND gate;
- size of loop is inversely related to size of AND gate.

Let's go back to the 8-cell 4-by-2 K-map for the 2-in MUX system, and see how the minimal DNF shown at the end of section §3.3 can be obtained by looping those four 1s in the K-map.



2-in MUX system:

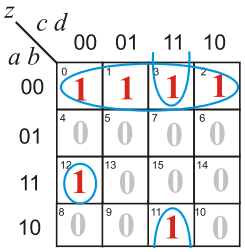
$$z \equiv (x \,\&\, \sim s) \vee (y \,\&\, s)$$

(i) loop of cells 6 (x y s-code 110) and 4 (x y s code 100) simplifies $(x \,\&\, y \,\&\, \sim s) \vee (x \,\&\, \sim y \,\&\, \sim s)$ to $(x \,\&\, \sim s)$ by cancelling complementary y's; and
(ii) loop of cells 3 (x y s-code 011) and 7 (x y s code 111) simplifies $(\sim x \,\&\, y \,\&\, s) \vee (x \,\&\, y \,\&\, s)$ to $(y \,\&\, s)$ by cancelling complementary x's.

Notice that we could also have looped together cells 6 and 7, but that does not add anything: it would be an *extra* loop that isn't needed to cover all the 1s.

So the resulting minimal DNF circuit for the 2-in MUX system has an outer 2-input OR gate and at the inner level, two 2-input AND gates, as we claimed at the end of section §3.3.

Now let's go back to the 16-cell 4-by-4 K-map example. This one has six cells containing 1s, and we can cover them using a total of three loops.



4-input example system:

$$z \equiv ((\sim a \,\&\, \sim b) \vee (\sim b \,\&\, c \,\&\, d) \vee (a \,\&\, b \,\&\, \sim c \,\&\, \sim d))$$

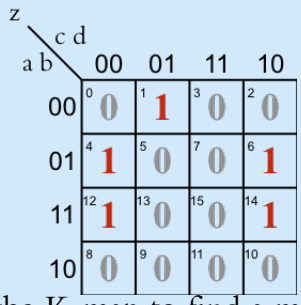
(i) size 4 loop of cells 0, 1, 3, and 2 simplifies to $(\sim a \,\&\, \sim b)$ by cancelling both c's and d's;

(ii) size 2 vertical wrap-around loop of cells 3 and 11 simplifies $(\sim a \,\&\, \sim b \,\&\, c \,\&\, d) \vee (a \,\&\, \sim b \,\&\, c \,\&\, d)$ down to $(\sim b \,\&\, c \,\&\, d)$ by cancelling a's; plus

(iii) size 1 loop of cell 12 – because cell 12 has no adjacent cells containing a 1 – gives $(a \,\&\, b \,\&\, \sim c \,\&\, \sim d)$, corresponding to a b c d-code 1100.

The resulting minimal DNF circuit for this 4-input system has an outer level 3-input OR gate, and three AND gates at the inner level: one 2-input, one 3-input, and one 4-input.

Now it is time for your to get some practice looping K-maps. Consider another system with four inputs a, b, c and d, and one output z.



Task: use loops on the K-map to find a minimal DNF formula expressing output z in terms of inputs a, b, c and d.

A minimal DNF formula expressing output z in terms of inputs a, b, c and d is:

$$z \equiv (b \mathbin{\&} \sim d) \vee (\sim a \mathbin{\&} \sim b \mathbin{\&} \sim c \mathbin{\&} d)$$

The size 4 loop of cells 4, 6, 12 and 14 cancels out a's and c's, and corresponds to the conjunction $(b \mathbin{\&} \sim d)$.
Cell 1 is lonely, with no adjacent cells containing 1s, so is covered by last resort loop of size 1. This cell has the a b c d code 0001 so it corresponds to the conjunction $(\sim a \mathbin{\&} \sim b \mathbin{\&} \sim c \mathbin{\&} d)$.

We do need to be clear than the Karnaugh-Veitch mapping method (or more generally, the Quine-McCluskey algorithm) only concerns DNF circuits, and not arbitrary logic circuits. It does *not* guarantee that there are no smaller circuits (with a smaller total number of gates) that do the same job but are not in DNF form.

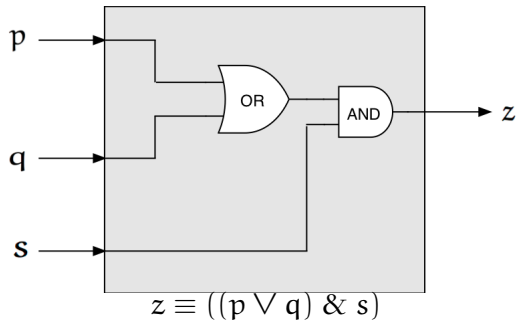
Similar to an example we saw in Section §3.3, the formula

$$(p \mathbin{\&} s) \vee (q \mathbin{\&} s)$$

is a minimal DNF, as is easy to demonstrate with the 4-by-2 K-map having three 1s in cells 3, 5 and 7, which is covered by two size-2 loops: one of cells 3 and 7, and the other of cells 7 and 5. Hence the circuit as shown below, consisting of a 2-input OR gate fed by two 2-input AND gates, is a minimal DNF circuit.

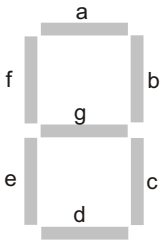
$$z \equiv ((p \mathbin{\&} s) \vee (q \mathbin{\&} s))$$

But we can readily find a non-DNF circuit with fewer gates.

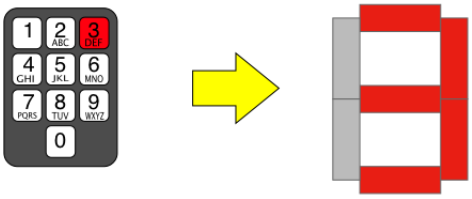


3.4.2 | DESIGN EXAMPLE: BCD-TO-7-SEGMENT DECODER

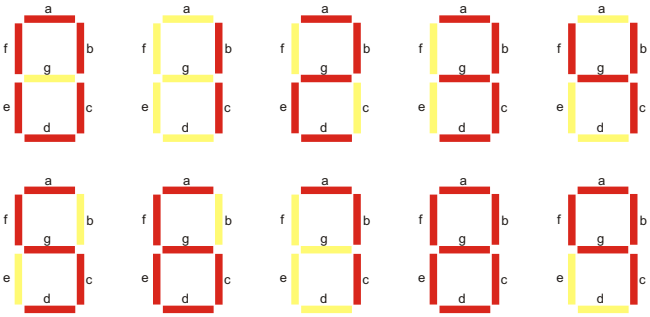
Let's get some more practice with K-maps via a common combinational system, a *decoder* or *display* system. We want to display on a 7-segment device any one of the decimal digits: $0, 1, 2, 3, 4, 5, 6, 7, 8, 9$. These sort of 7-segment displays are usually made of LCDs (Liquid Crystal Displays) or LEDs (Light Emitting Diodes).



For example, suppose we hooked it up the BCD encoder from Section §3.3.1, which turned a keypad decimal digit input into a 4-bit binary code. Pressing the keypad button 3 should then result in lighting up the 5 out of 7 segments that form the shape of the symbol for 3.

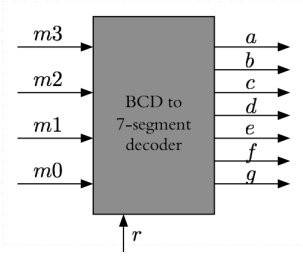


For all 10 decimal digits, here are all the patterns needed to form those symbols on a 7-segment display.



Notice that the patterns are all distinct. It is only decimal 8 that lights up all 7 segments; all the others have at least one segment that is not lit up. Notice also that we have to agree on the pattern for 1, choosing to align to the right of the display. The 7 segments are labelled a, b, c, d, e, f and g. The two segments needed to be lit up for the symbol 1 are b and c.

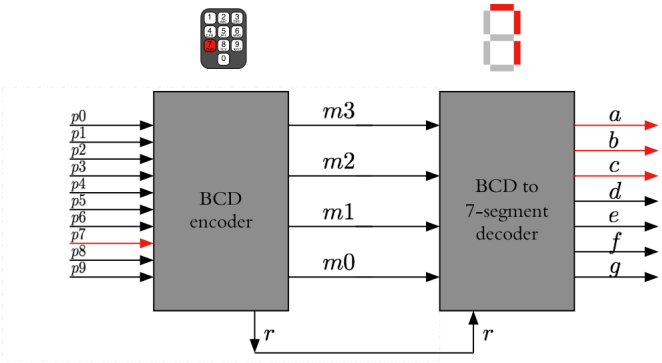
Block structure:



Functional description: the BCD-to-7-segment decoder system takes as input a 4-bit binary code m3 m2 m1 m0, together with control input r, and returns as output a 7-bit code a b c d e f g which describes how to "light up" the 7-segment display by turning on or leaving off the various display segments.

- If the 4-bit input is BCD, in range $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$, *and in addition*, the control input r is 0, then the intended lighting up of the display segments is to be done.
- Otherwise, if the input is in range $\{10, 11, 12, 13, 14, 15\}$ or if the control input r is 1, then none of the 7 segments are to be turned on.

Decimal keypad to 7-segment display CODEC circuit:
By combining the BCD encoder from Section §3.3.1 with this BCD-to-7-segment decoder, we could then simply create a decimal keypad to 7-segment display CODEC circuit by connecting outputs of the first with inputs of the second.



For example, if button 7 on the keypad is the only button cleanly pressed, then input signal p7 will be the only input active for the BCD

encoder. The resulting 4-bit output code m=0111 and the error output r = 0. Then the BCD-to-7-segment decoder will activate outputs a, b and c, to light up the display to form the symbol 7. If the keypad is mishandled and two or more buttons are pressed, then the BCD encoder will produce an error output r = 1, which would then be passed to the BCD-to-7-segment decoder, resulting in no segments being lit up.

Truth table for BCD-to-7-segment decoder:

decimal	m3	m2	m1	m0	r	a	b	c	d	e	f	g
0	0	0	0	0	0	1	1	1	1	1	1	0
1	0	0	0	1	0	0	1	1	0	0	0	0
2	0	0	1	0	0	1	1	0	1	1	0	1
3	0	0	1	1	0	1	1	1	1	0	0	1
4	0	1	0	0	0	0	1	1	0	0	1	1
5	0	1	0	1	0	1	0	1	1	0	1	1
6	0	1	1	0	0	1	0	1	1	1	1	1
7	0	1	1	1	0	1	1	1	0	0	0	0
8	1	0	0	0	0	1	1	1	1	1	1	1
9	1	0	0	1	0	1	1	1	1	0	1	1
10	1	0	1	0	×	0	0	0	0	0	0	0
11	1	0	1	1	×	0	0	0	0	0	0	0
12	1	1	0	0	×	0	0	0	0	0	0	0
13	1	1	0	1	×	0	0	0	0	0	0	0
14	1	1	1	0	×	0	0	0	0	0	0	0
15	1	1	1	1	×	0	0	0	0	0	0	0
	×	×	×	×	1	0	0	0	0	0	0	0

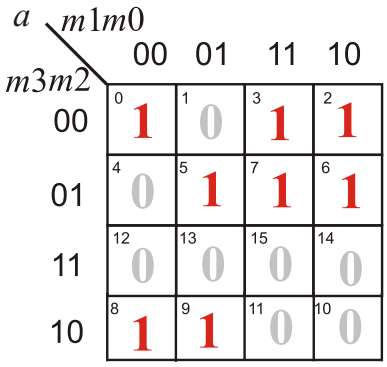
For the intended lighting of decimal values 0, 1, 2, up to 9, as shown on the first 10 rows of the table, you can check that we have the right distribution of 0s and 1s. For example, take decimal 5. That should light up segments a, c, d, f, and g. Go to the truth table and see the output for 5 has 1s under a, c, d, f, and g.

For the remaining rows of the truth table, remember that we use × to mean "don't care" – so the bit-value can be either 0 or 1. When the decimal value of the 4-bit input m is between 10 and 15, we don't care whether the control input r is 0 or 1; either way, the output is all 0s and no segments are lit up. When the control input r is 1, the value of m won't matter, as in all such cases, the output is all 0s and no segments are lit up.

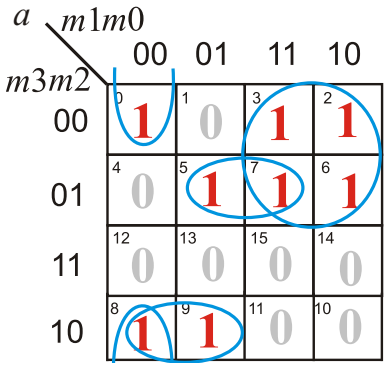
The BCD-to-7-segment decoder gives us 7 examples of 4-by-4 K-maps to play with. For each of the outputs a, b, c, d, e, f, and g, we can use a 4-by-4 K-map to derive a DNF formula for that output in terms of the 4 main input bits, m3, m2, m1 and m0. From the truth table, we can see that additionally, we need to add an extra *conjunct* for the literal ~r *within each conjunction in the DNFs*, in order to express the dependence on the 5th input r.

BCD-to-7-segment decoder: a output

From the truth table column for a:



Looping the eight 1s with four loops:



$$a \equiv (\sim m3 \,\&\, m1 \,\&\, \sim r) \vee (\sim m2 \,\&\, \sim m1 \,\&\, \sim m0 \,\&\, \sim r) \vee (\sim m3 \,\&\, m2 \,\&\, m0 \,\&\, \sim r) \vee (m3 \,\&\, \sim m2 \,\&\, \sim m1 \,\&\, \sim r)$$

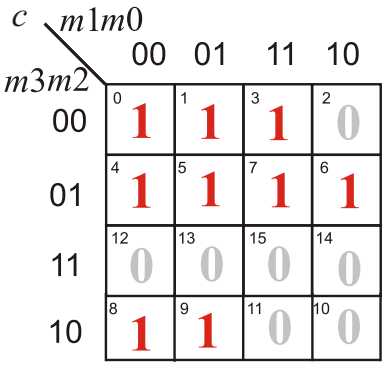
The biggest loop we can get is of size 4, covering cells 3, 2, 7 and 6. For this loop, m2 and m0 are cancelled, and the remaining bit-values for m3 and m1 are 0 and 1, respectively, so combining with the extra conjunct ~r, we get the conjunction (~m3 & m1 & ~r) for this loop.
The size-2 loop of cells 0 and 8 cancels m3, and the resulting conjunction is (~m2 & ~m1 & ~m0 & ~r).
The size-2 loop of cells 5 and 7 cancels m1, and the resulting conjunction is (~m3 & m2 & m0 & ~r). This size-2 loop around cells 5 and 7 is better than a size-1 loop around cell 5 on its own.
Finally, the size-2 loop of cells 8 and 9 cancels m0, and the resulting conjunction is (m3 & ~m2 & ~m1 & ~r). Again, this size-2 loop around cells 8 and 9 is better than a size-1 loop around cell 9 on its own.

Again, notice that it is OK to cover some cells by two loops. Here, that is the case for cells 7 and 8. This is fine because it is resulting in larger loops.

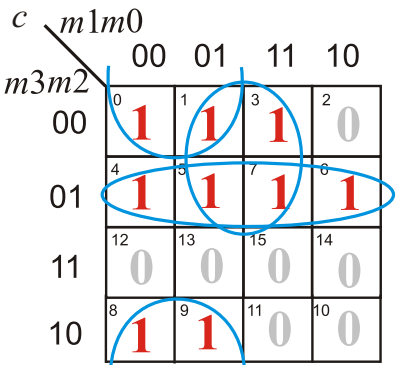
Our minimal DNF gives a circuit for output a with an outer-level 4-input OR gate, and at the inner level, one 3-input AND gate and three 4-input AND gates. This minimal DNF is certainly not unique. We could have replaced the loop around cells 0 and 8 with a different one around cells 0 and 2, but the sizes would all be the same.

BCD-to-7-segment decoder: c output

From the truth table column for c:



Looping the nine 1s with three loops:



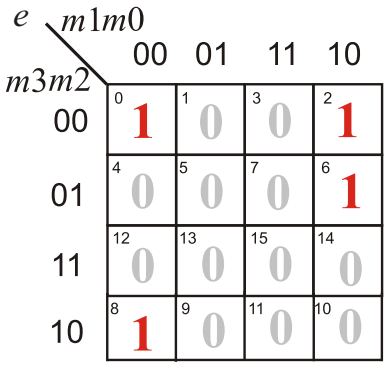
$$c \equiv (\sim m2 \,\&\, \sim m1 \,\&\, \sim r) \vee (\sim m3 \,\&\, m0 \,\&\, \sim r) \vee (\sim m3 \,\&\, m2 \,\&\, \sim r)$$

For the c output, we can get three size 4 loops in the 4-by-4 K-map for m3 m2 vs m1 m0.
The first, the vertical wrap-around of cells 0, 1, 8 and 9, sees both m3 and m0 cancelled, and the bit values for the remaining m2 and m1 are both 1, so combining with the extra conjunct ~r, we get the conjunction (~m2 & ~m1 & ~r) for this loop.
The second size 4 loop, of cells 1, 3, 5 and 7, cancels both m2 and m1, and the bit values for m3 and m0 are 0 and 1, respectively, so combining with the extra conjunct ~r, we get the conjunction (~m3 & m0 & ~r) for this loop.
The third size 4 loop, of cells 4, 5, 6 and 7, cancels both m1 and m0, and the bits for m3 and m2 are 0 and 1, respectively, so combining with the extra conjunct ~r, we get the conjunction (~m3 & m2 & ~r) for this loop.

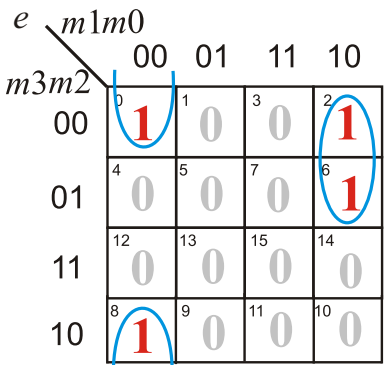
Again, notice that it is OK to cover some cells by two loops. Here, that is the case for cells 1, 5 and 7. This is fine because the result is larger loops.

Our minimal DNF gives a circuit for output c with an outer-level 3-input OR gate, and at the inner level, three 3-input AND gates. This minimal DNF happens to be unique.

BCD-to-7-segment decoder: *e* output
From the truth table column for e:



Looping the four 1s with two loops:



$$e \equiv (\sim m2 \,\&\, \sim m1 \,\&\, \sim m0 \,\&\, \sim r) \vee (\sim m3 \,\&\, m1 \,\&\, \sim m0 \,\&\, \sim r)$$

For the e output, we can get two size 2 loops in the 4-by-4 K-map for m3 m2 vs m1 m0.
The first, the vertical wrap-around of cells 0 and 8, sees m0 cancelled, and the bit values for m3, m2 and m1 are all 0, so combining with the extra conjunct ~r, we get the conjunction (~m3 & ~m2 & ~m1 & ~r) for this loop.
The second size 2 loop, of cells 2 and 6, cancels m2, and the bit values for m3, m1 and m0 are 0, 1 and 0, respectively, so combining with the extra conjunct ~r, we get the conjunction (~m3 & m1 & ~m0 & ~r) for this loop.

Our minimal DNF gives a circuit for output c with an outer-level 2-input OR gate, and at the inner level, two 4-input AND gates. This minimal DNF also happens to be unique.

3.4.3 | MINIMIZING MULTI-OUTPUT CIRCUITS

In the BCD-to-7-segment decoder system above, we used K-maps to derive minimal DNF characterisations of each output *separately*, without considering whether or not we could find any loops in the K-maps – corresponding to AND gates in the circuit diagram – that are *common* to several outputs. Each time an AND gate is shared by two outputs, there is a saving of one AND gate in the circuit for the whole system.

For example, from studying the truth table for the BCD-to-7-segment decoder system, we can see that for six of the seven outputs, namely a, b, c, d, f and g, there is a common requirement for the conjunction:

```
m3 & ~m2 & ~m1 & ~r
```

corresponding to the loop of cells 8 and 9 in the 4-by-4 K-map for m3 m2 vs. m1 m0. Four of the outputs, a, d, f and g, can share a 4-input AND for this common conjunction. The outputs b and c also need the m3-complementary conjunction (as above, but with ~m3 instead of m3), so these two outputs can share a 3-input AND gate for the conjunction:

```
~m2 & ~m1 & ~r
```

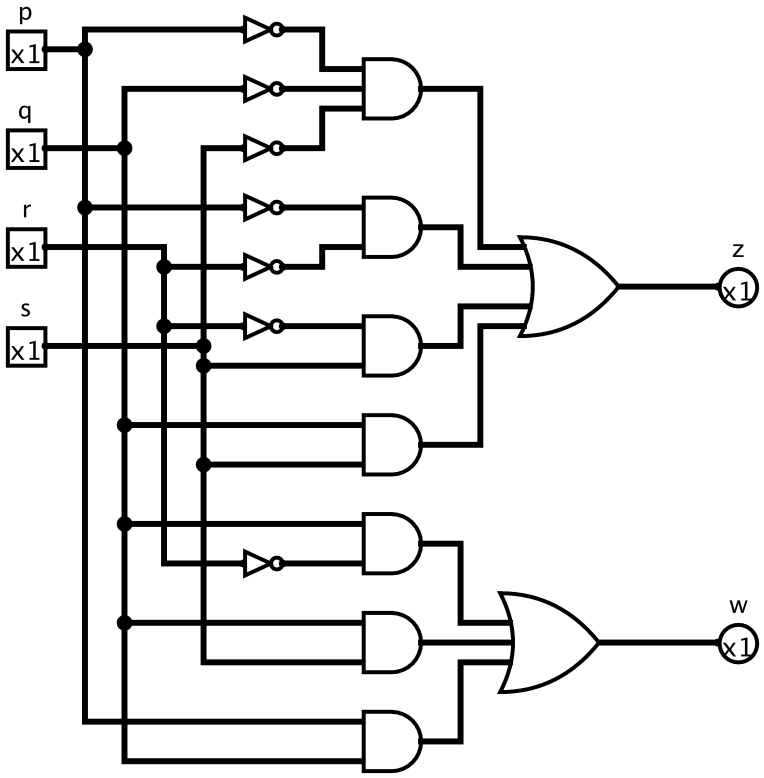
corresponding to the loop of cells 0, 1, 8 and 9 in the 4-by-4 K-map.

When measuring the *size* of a circuit diagram, we will use first, the number of AND gates, and second, the *total input-count* for the set of AND gates, which is the result of adding up the input-size of each of the gates in the set.

We say a circuit diagram is a *minimal DNF circuit* for a combinational system if and only it is in standard DNF form and every other circuit diagram in standard DNF form that implements the same system has at least as many AND gates, and those AND gates have a total input-count at least as large as that of the given circuit diagram.

Equivalently, a circuit diagram in standard DNF form is *not* a minimal DNF circuit for a combinational system if and only if there is another circuit diagram in standard DNF form implementing the same system that either has a smaller number of AND gates, or has the same number of AND gates but the total input-count of those AND gates is smaller than that of the given circuit diagram.

Notice that there is some subtlety here. First, it is possible to have each output of a 2-output system separately expressed as a minimal DNF formula in terms of the inputs, but the resulting 2-output circuit built according to those formulas to fail to be minimal DNF because of a failure to make optimal use of AND gates that can be shared by both outputs.

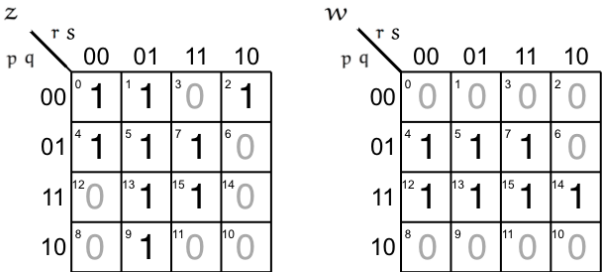


For a simple example, consider a combinational system with input signals p, q, r and s, and output signals *z* and *w*, which has a circuit diagram as above. This circuit has 7 AND gates and a total input-count of 15 across those AND gates. (In the circuit diagram, the "×1" label merely indicates these are 1-bit signals, and is an artefact of the software used to generate the diagram.)

Direct from the circuit diagram, we can read off separately two DNF formulas for *z* and *w*, namely:

$$z \equiv (\sim p \,\&\, \sim q \,\&\, \sim s) \vee (\sim p \,\&\, \sim r) \vee (\sim r \,\&\, s) \vee (q \,\&\, s)$$
$$w \equiv (q \,\&\, \sim r) \vee (q \,\&\, s) \vee (p \,\&\, q)$$

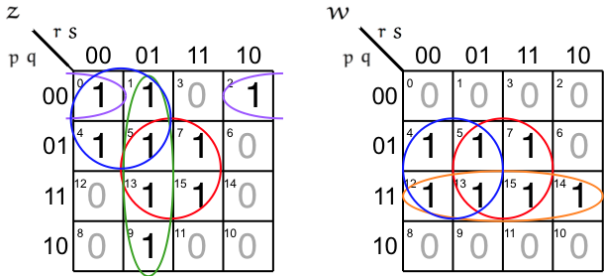
From these DNF formulas, we can then fill in the 1s in the K-maps for the two outputs.



In the DNF for *z* from the given circuit, conjunction (~p & ~q & ~s) corresponds to 1s in cells 0 and 2; (~p & ~r) corresponds to 1s in cells 0, 1, 4 and 5; (~r & s) corresponds to 1s in cells 1, 5, 9 and 13; and (q & s) corresponds to 1s in cells 5, 7, 13 and 15. In the DNF for *w*

from the circuit diagram, conjunction (q & ~r) corresponds to 1s in cells 0 and 2; (q & s) corresponds to 1s in cells 5, 7, 13 and 15; and (p & q) corresponds to 1s in cells 12, 13, 14 and 15.

Looking at the circuit diagram and the DNF formulas, we can easily see the AND gate for (q & s) is *common* to the two outputs, so we can improve this circuit by dropping one copy of the (q & s) AND gate. For the remaining copy, say the one closest to the z output, we branch the output wire of that AND gate and send it as an input to the OR gate for w. The result will be a circuit in standard DNF form that has only 6 AND gates with a total input-count of 13. Using a pair of K-maps, we can show that this is in fact the best that can be done, so is a minimal DNF circuit for the system.



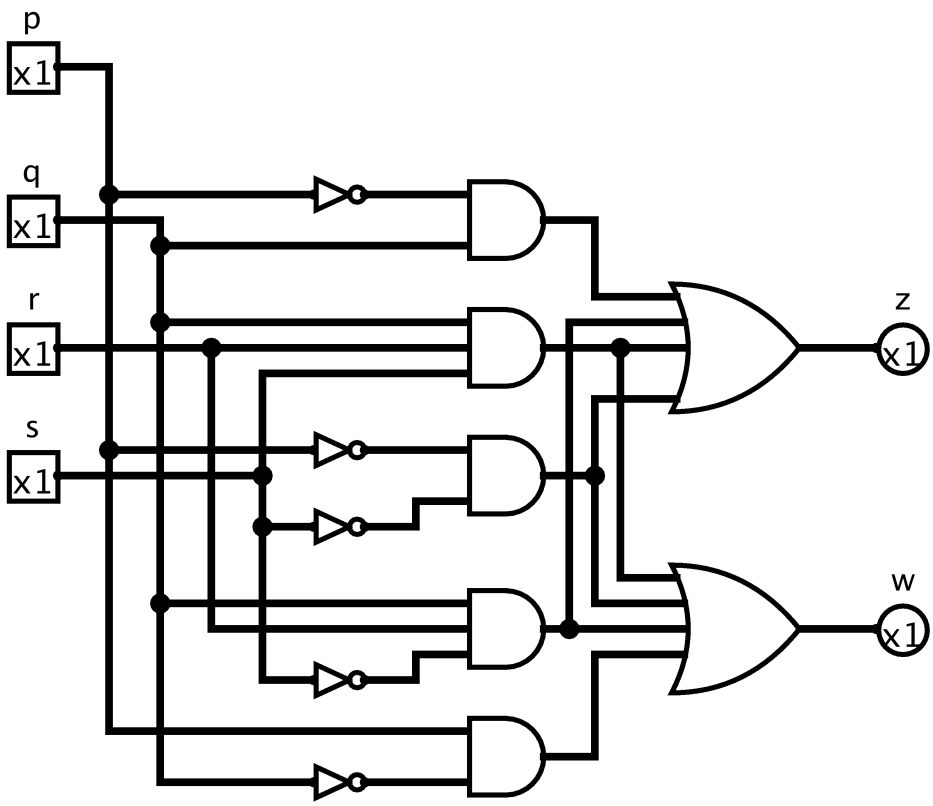
loops (total of 6)	output(s)	conjunctions	AND gates
red loop (cells 5, 7, 13, 15)	z and w	(q & s)	2-input
blue loop (cells 0, 1, 4, 5)	z	(~p & ~r)	2-input
green loop (cells 1, 5, 9, 13)	z	(~r & s)	2-input
purple loop (cells 0, 2)	z	(~p & ~q & ~s)	3-input
blue loop (cells 4, 5, 12, 13)	w	(q & ~r)	2-input
green loop (cells 12, 13, 14, 15)	w	(p & q)	2-input
total input-count			13

Putting these together, we get the same minimal DNF formulas for each output:

$$\begin{aligned} z &\equiv (q \,\&\, s) \vee (\sim p \,\&\, \sim r) \vee (\sim r \,\&\, s) \vee (\sim p \,\&\, \sim q \,\&\, \sim s) \\ w &\equiv (q \,\&\, s) \vee (q \,\&\, \sim r) \vee (p \,\&\, q)\,. \end{aligned}$$

There are no larger loops possible within these K-maps, and the loop for (q & s) is the only one that can be shared between z and w. So a circuit with 6 AND gates and a total input-count of 13 across those AND gates is the best that can be done, and gives a minimal DNF circuit.

Now it is your turn.



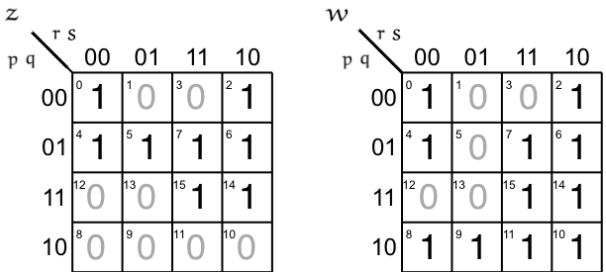
Consider the circuit diagram above for a combinational system with input signals p, q, r and s, and output signals z and w. The circuit is in standard DNF form. It has a total of 5 AND gates with a total input-count of 12 across those AND gates.
Identify which one of the following statements is correct.

(a) The circuit diagram above is a minimal DNF circuit for this combinational system.

(b) The circuit diagram above is not a minimal DNF circuit for this system because there is another circuit for the same system with less than 5 AND gates.

(c) The circuit diagram above is not a minimal DNF circuit for this system because there is another circuit for the same system which has 5 AND gates but a total input-count for its AND gates of less than 12.

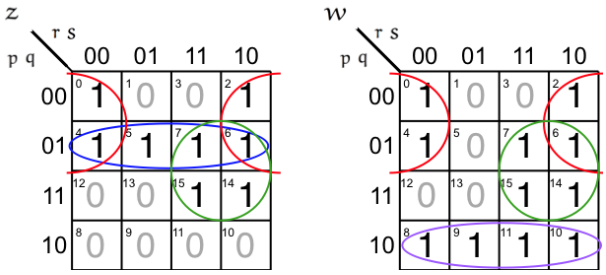
To answer this question, first read the circuit diagram to complete K-maps for the two outputs of this system (there are some blank K-maps at the end of this chapter).



Direct from the circuit diagram, we get the above pattern of 1s and 0s for the system, with the DNF formulas:

$$z \equiv (\sim p \,\&\, q) \vee (q \,\&\, r \,\&\, s) \vee (\sim p \,\&\, \sim s) \vee (q \,\&\, r \,\&\, \sim s)$$
$$w \equiv (q \,\&\, r \,\&\, s) \vee (\sim p \,\&\, \sim s) \vee (q \,\&\, r \,\&\, \sim s) \vee (p \,\&\, \sim q)\,.$$

Taking the largest loops as possible, and sharing loops where possible, we get the following:



The correct answer is **(b)** . There is a minimal DNF circuit with 4 AND gates and a total input-count of 8.

loops (total of 4)	output(s)	conjunctions	AND gates
green loop (cells 6, 7, 14, 15)	z and w	(q & r)	2-input
red loop (cells 0, 2, 4, 6)	z and w	(~p & ~s)	2-input
blue loop (cells 4, 5, 6, 7)	z	(~p & q)	2-input
purple loop (cells 8, 9, 10, 11)	w	(p & ~q)	2-input
total input-count			8

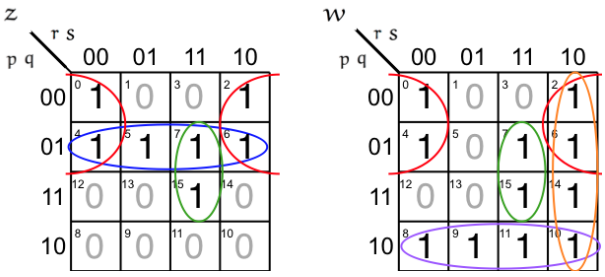
Putting these together, the DNF circuit is described by:

$$z \equiv (q \,\&\, r) \vee (\sim p \,\&\, \sim s) \vee (\sim p \,\&\, q)$$
$$w \equiv (q \,\&\, r) \vee (\sim p \,\&\, \sim s) \vee (p \,\&\, \sim q)\,.$$

There are no larger loops possible, and there is no other sharing of loops to compare, so we can conclude that the DNF circuit described by these loops is a minimal DNF circuit.

There is a bit more subtlety in this topic. It is possible to have a minimal DNF circuit for a multi-output system where, because of sharing of AND gates across multiple outputs, one or more of the outputs have a DNF formulas describing their part of the circuit which fails to be a *minimal* DNF formula.

Consider the combinational system with with input signals p, q, r and s, and output signals z and w, that is the same as the previous system *except that* output z has value 0 on p q r s input 1 1 1 0; this is cell 14 in the K-map for z.



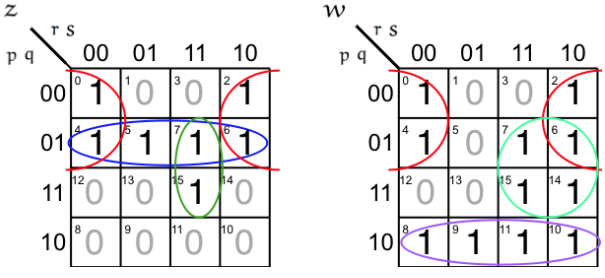
We claim this joint looping of the two K-maps together gives us a minimal DNF circuit with 5 AND gates and a total input-count of 11.

loops (total of 5)	output(s)	conjunctions	AND gates
red loop (cells 0, 2, 4, 6)	z and w	$(\sim p \,\&\, \sim s)$	2-input
green loop (cells 7, 15)	z and w	$(q \,\&\, r \,\&\, s)$	3-input
blue loop (cells 4, 5, 6, 7)	z	$(\sim p \,\&\, q)$	2-input
orange loop (cells 2, 6, 10, 14)	w	$(r \,\&\, \sim s)$	2-input
purple loop (cells 8, 9, 10, 11)	w	$(p \,\&\, \sim q)$	2-input
total input-count			11

The green loop shared by z and w is *not* the optimal choice if we consider the two outputs separately. In the K-map for w, we could do better with a size 4 loop of cells 6, 7, 14 and 15, and in the K-map for z, we would separately take the green size 2 loop of cells 7 and 15. The trade-off for output w is that, if we do use the green size 2 loop of cells 7 and 15 in common with output z, then in order to cover the 1 in cell 14, we still need to include an extra loop: a size 4 loop of cells 2, 6, 10 and 14 will work, but equally good is a size 4 loop of cells 10, 11, 14 and 15. Putting together the AND gates in the list above, we get the DNF circuit description:

$$\begin{aligned} z &\equiv (\sim p \,\&\, \sim s) \vee (q \,\&\, r \,\&\, s) \vee (\sim p \,\&\, q) \\ w &\equiv (\sim p \,\&\, \sim s) \vee (q \,\&\, r \,\&\, s) \vee (r \,\&\, \sim s) \vee (p \,\&\, \sim q) \end{aligned}$$

To make the case that the circuit so described in *minimal* DNF, compare the 2-output circuit created when, instead, we look for minimal DNF formulas by doing the K-map looping separately for each of the outputs z and w. The result is a DNF circuit which also has 5 AND gates with a total input-count of 11 – no smaller in size, but no larger either.



loops (total of 5)	output(s)	conjunctions	AND gates
red loop (cells 0, 2, 4, 6)	z and w	$(\sim p \,\&\, \sim s)$	2-input
green loop (cells 7, 15)	z	$(q \,\&\, r \,\&\, s)$	3-input
blue loop (cells 4, 5, 6, 7)	z	$(\sim p \,\&\, q)$	2-input
aqua loop (cells 6, 7, 14, 15)	w	$(q \,\&\, r)$	2-input
purple loop (cells 8, 9, 10, 11)	w	$(p \,\&\, \sim q)$	2-input
total input-count			11

Putting these together, we get an alternative minimal DNF circuit description:

$$\begin{aligned} z &\equiv (\sim p \,\&\, \sim s) \vee (q \,\&\, r \,\&\, s) \vee (\sim p \,\&\, q) \\ w &\equiv (\sim p \,\&\, \sim s) \vee (q \,\&\, r) \vee (p \,\&\, \sim q)\,. \end{aligned}$$

For this system, the minimal size for a DNF circuit is 5 AND gates with a total input-count of 11, and there are at least three equally optimal, minimal DNF circuits available; two we have described in detail, and one has an output characterised by a non-minimal DNF formula.

Simplification of DNF (or SOP) formulas by hand using K-maps is only practical for small systems with a small number of inputs. The core idea of K-map simplification – matching and cancelling complementary pairs – is generalised within the *Quine-McCluskey algorithm* which finds minimal DNFs for propositional logic formulas for any number of atoms. This core algorithm is still implemented in electronic design automation (EDA) software tools.

We have just touched the surface in this introduction to digital systems. In particular, we have mostly seen small "hand-drawn" systems with 16-row truth tables or circuit diagrams that fit on less than a standard page. To progress further, to design the sort of circuits we see inside laptop computers or tablet devices, it is essential to make use of a *hardware description language* such as VHDL or VERILOG, in conjunction with EDA tools. A hardware description language provides yet another way to describe digital systems – in addition to truth tables, functional descriptions, circuit diagrams and K-maps – that is optimised for use with design automation software. For example, the last combinational system discussed above could be described in VHDL as follows:

```
1  entity  4-in-2-out bool  is
2      port  (p, q, r, s : in bit;
3             z, w : out bit);
4  end  4-in-2-out bool;
5
6  architecture  example_DNF  of  4-in-2-out bool  is
7  begin
8      z <= (not p  and not s)
9           or (q and r and s)
10          or (not p  and  q);
11     w <= (not p  and  not s)
12          or (q and r)
13          or (p  and  not q);
14 end example_DNF;
```

FURTHER LEARNING

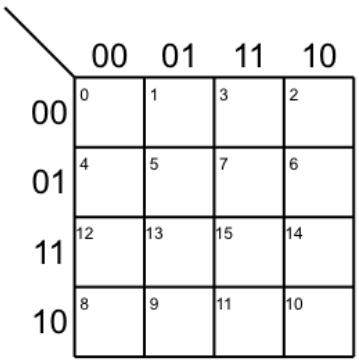
The aim of this chapter is to give a basic introduction to combinational digital systems and their foundation in propositional logic. In the sequel **Logic: Language and Information 2**, there is a further applications section on digital systems that will delve deeper into the topic, including introducing *sequential digital systems* with memory, delay components, and feedback loops, and the analysis of such systems in *predicate logic* and its extensions.

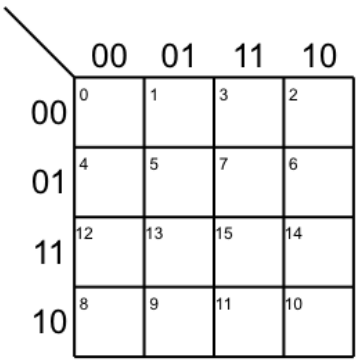
For those with a deeper interest in digital systems, there is a wealth of textbooks and online resources available. These include:

- Burch, Carl. *Logisim*; free open-source software (GPL: Gnu General Public License) providing an educational tool for designing and simulating digital logic circuits; http://www.cburch.com/logisim/ (Accessed 5 Mar, 2014).
- Chandrakasan, Anantha. 6.111 *Introductory Digital Systems Laboratory*, Spring 2006. (MIT OpenCourseWare: Massachusetts Institute of Technology); http://ocw.mit.edu/courses/electrical-engineering-and-computer-s 6-111-introductory-digital-systems-laboratory-spring-2006 (Accessed 5 Mar, 2014). License: CC BY-NC-SA
- Srinivasan, S. *Digital Circuits and Systems*, eeUniversity, EE-Times, India; video lecture series; http://www.eetindia.co.in/VIDEO_CAT_800040.HTM (Accessed 5 Mar, 2014).
- Crisp, John. *Introduction to Digital Systems* (Elsevier, 2000); widely available textbook.

BLANK TRUTH TABLES AND K-MAPS

row						
0	0	0	0	0		
1	0	0	0	1		
2	0	0	1	0		
3	0	0	1	1		
4	0	1	0	0		
5	0	1	0	1		
6	0	1	1	0		
7	0	1	1	1		
8	1	0	0	0		
9	1	0	0	1		
10	1	0	1	0		
11	1	0	1	1		
12	1	1	0	0		
13	1	1	0	1		
14	1	1	1	0		
15	1	1	1	1		





PHILOSOPHY: VAGUENESS

4

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

- The difference between vagueness (where a meaning isn't determined precisely enough for us to be able to decide every case) and ambiguity (a word having several diferent meanings).
- The *sorites paradox* concerning vague concepts.
- The *supervaluational*, *epistemicist*, and *fuzzy logic* approaches to vague concepts.

SKILLS

- Distinguish the premises and conclusion in a sorites paradox argument, and give examples of these arguments.
- Use three-valued truth tables to give an account of how three-valued Łukasiewicz logic treats vague information.
- Use infinitely-valued truth table rules to give an account of how infinite-valued Łukasiewicz logic treats vague information.
- Explain how different treatments of vagueness (supervaluations, epistemicism and fuzzy logic) treat vague expressions, and give reasons for and against these approaches.

Two-valued Boolean, classical propositional logic makes the assumption that every proposition is either *true* (1) or *false* (0). But the world is not just filled with sharp borderlines between categories. The world is *vague*. Statements are not all clearly true or clearly false. When we look at a continuously varying strip of colour, like this:

Point to parts of the strip from left to right, at the left end we are happy to say that it is *red*—that proposition 'this looks red' is *true*. At the right end we are happy to say that it is *not* red—the proposition 'this looks red' is *false*. But I cannot spot *where* the statements flip from *true* to *false*. There seems to be no sharp borderline between the *red* and the *non-red*. This is one example of the phenomenon of *vagueness*. In this chapter,

VAGUENESS is but one example of many that puts pressure on the assumption of bivalence. Other examples are the PARADOXES (*This very sentence is false.*), AMBIGUITY (*I will meet you at the bank.*), PRESUPPOSITION FAILURE (*Have you stopped cheating on your exams?*), FUTURE CONTINGENTS (*There will be a sea battle tomorrow.*), and other phenomena besides these. Books have been written on each of these topics.

we'll take a look at the *philosophical* assumption that every proposition is either *true* or *false* using the problem of vagueness as our focus.

> BIVALENCE is the assumption that every proposition is either *true* or *false*. That is, that every proposition takes one of these *two values*.

Bivalence is closely related to the *law of the excluded middle*, the fact that $p \vee \sim p$ is a tautology. As you will see, a logical system can satisfy the law of the excluded middle without assuming bivalence. The reverse can happen, too.

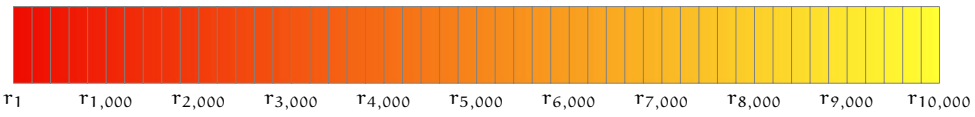
Our question will be this: should we look for an alternative logical system that does not make the bivalence assumption? Or can the bivalence assumption survive in the face of vagueness? In either case, we should have a better understanding owhat in involved in making the bivalence assumption when we consider closely the phenomenon of *vagueness*.

4.1 | THE SORITES PARADOX

Let's sharpen up our understanding of vagueness by looking at the *sorites paradox*, so called from the Greek word 'soros' for 'heap.' (The sorites paradox was considered first with grains of sand making a heap. 10,000 grains of sand is enough to make a heap. If 10,000 grains of sand is enough to make a heap, then so is 9,999 (one grain of sand doesn't make the difference between heap and non-heap), … if 2 grains of sand is enough to make a heap, then so is 1. Therefore, 1 grain of sand is enough to make a heap. We will focus on the colour case in these notes. But what we say here will equally apply there.)

Let's focus on our strip, shading from red to yellow. Let's imagine it divided up neatly into 10,000 little patches.

I can't do 10,000 here in this document, but I can do 50. The important feature of using a *large number* is that we want the patches to be close enough in colour that adjacent patches look *indistinguishable* in colour to you and to me.



For each patch, numbered from patch 1 at the fire-engine-red leftmost end, to patch 10,000 at the canary-yellow rightmost end, let's consider the specific statement

- r_i: *patch* i *looks red*.

If you like, make this statement as specific as you like. For example: "Patch i looks red to me today, when examined under standard lighting conditions."

This collection of statements, from r_1 to $r_{10,000}$ forms the heart of the sorites paradox. Here is why.

- r_1 seems true.
- $r_{10,000}$ seems false.
- Each $r_i \supset r_{i+1}$ seems true.

It's worth spending a little bit of time explaining why each of these things seems to be the case. First, r_1 seems true because we started the

strip off with the colour red. This leftmost patch on the strip looks red to me if *anything* looks red to me. Second, $r_{10,000}$ seems false, since the strip shades all the way to yellow, which is *nothing* like red. Thirdly, although I think you might be able to tell the difference between the colours of some adjacent pairs of the 50 patches here, if you imagine each of these 50 patches split into 200 subdivisions, the differences between these tiny patches will be so miniscule that they go beyond the human eye's ability to distinguish. We choose the number of patches to be large enough that you cannot distinguish any difference in colour between patch $5,000$ and patch $5,001$, or between patch 1 and patch 2, or between any adjacent pair of patches. If this is the case, then each formula $r_i \supset r_{i+1}$ is true, because if one patch looks red, then any patch which looks indistinguishable from it must look red, too, whether either of the patches look red or not. This applies equally to the left end of the strip, the right end, and all points in between. There seems to be no drastic cutoff between the red and the non-red.

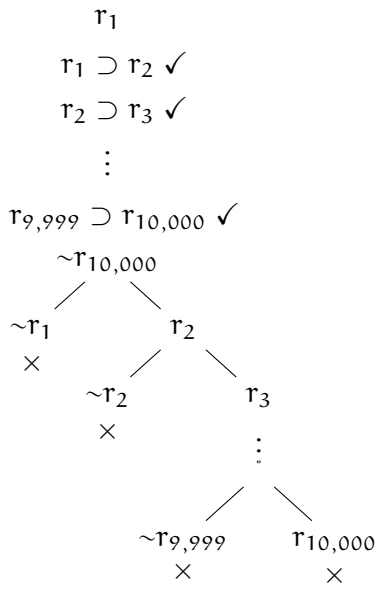
Why? Because if patch i looks red and patch $i+1$ does not look red, then the patches *are* distinguishable, aren't they?

All of this so far seems common sense, at least when you consider it like this. And perhaps those three judgements (r_1 is true; $r_{10,000}$ is false; each $r_i \supset r_{i+1}$ is true) *are* common sense. (We say '*perhaps*' here because we've used are not common or everyday concepts in setting this up. In particular, '$\supset$' is the logician's material conditional, and we we have taken care to precisely specify $10,000$ different statements for our consideration. This is not 'common sense' in any straightforward sense of the words.) The *paradox* arises when we consider one final piece of the puzzle.

- $r_1, r_1 \supset r_2, r_2 \supset r_3, \ldots, r_{9,999} \supset r_{10,000} \vdash r_{10,000}$

Indeed, logic tells us that this argument is valid. I wouldn't advise doing a truth table to check it. $10,000$ atomic propositions means $2^{10,000} \approx 2\times10^{67}$ rows of a truth table, which would take some time to complete. On the other hand, a proof tree for this argument not too difficult to survey.

To compare, current estimates say that the universe is approximately 4.35×10^{17} seconds old.



A proof if this magnitude is too large for us to display in full. If it is written at the scale where processing a conditional (and closing the associated formulas) takes around 3cm, the entire proof tree would take approximately $10,000 \times 3\text{cm} = 30\text{km}$. That would be *rather large*, but you could survey a proof of this size in *much* less time than estimated age of the universe, and a computer (or frankly, your *phone*) could construct a proof of this size in a flash.

So, according to the account of logical validity we've studied in the previous part of this course, the argument form is *valid*. This is the problem. This is the *sorites paradox*.

Before we go any further, it will be helpful to define our vocabularly clearly, so we know what we're talking about. First, the sorites *argument*.

> A SORITES ARGUMENT is an argument of the form
>
> $$p_1, p_1 \supset p_2, p_2 \supset p_3, \ldots, p_{n-1} \supset p_n \text{ therefore } p_n$$
>
> in which the initial premise p_1 seems true, the conclusion p_n seems false, and each of the transition premises $p_i \supset p_{i+1}$ seems true, because of *vagueness*, the presence of borderline cases.

A person 1 microsecond old is young. If a person n microseconds old is young, so is a person $n+1$ microseconds old. Therefore a person 3×10^{15} microseconds old is young. (3×10^{15} microseconds is a little over 95 years.)

We don't assume that the sorites argument has 10,000 premises. The 'n' is a placeholder for a number. There are sorites arguments with many more and there are sorites arguments with many fewer premises, too. The crucial feature of sorites arguments is that the premises must all seem true and the conclusion seem false.

That is the sorites *argument*. Now for the *paradox*.

> The SORITES PARADOX: In a sorites argument,
>
> (*a*) the premises *seem true*, and
>
> (*b*) the argument *seems valid*, but
>
> (*c*) the conclusion *seems false*.

This is the *paradox*. An argument with true premises and a false conclusion is not valid. We cannot have all of the things that *seem* to be the case *actually* be the case.

It seems that any response that attempts to make sense of the phenomenon of the sorites paradox will fall into one of the following four categories.

None of this is to reject the possibility of mixed strategies, which say give one response to one particular sorites argument, and another response to another one. These categories are intended to sort out the possible responses to a single given sorites argument.

1. *Logic does not apply* to vague expressions.
2. Logic does apply, and the *argument is not valid*.
3. Logic does apply, the argument is valid but *one of the premises is false*.
4. Logic does apply, the argument is valid and sound, and therefore *the conclusion is true*.

According to OPTION 1, the question of the validity or invalidity of the sorites argument does not arise. There is no contradiction in the para-

dox because we should not be applying logic to vague expressions like 'looks red,' 'is young' or 'is a heap.'

According to OPTION 2, logic applies, but that the sorites argument (despite what classical logic says) is actually *invalid*. We will explore this option in the next section, where we consider *revising* our account of logic to have more than two truth values.

Any remaining options will have to say that logic applies and the sorites argument is *valid*. This comes in two forms. Either the premises are all true, or not.

According to OPTION 3, logic applies, the sorites argument is valid, but the premises are not all true. This means that the conclusion can still be rejected.

Finally, the last remaining option, OPTION 4, grants that logic applies, the argument is valid and the premises are all true. It follows that the *conclusion* of the sorites argument is true, too—that is, that the yellow end of the strip of colour is red, that a 100 year old person is young, that 1 grain of sand makes a heap, and that chickens have existed for billions of years.

This last example involves the sorites paradox concerning the vagueness of species boundaries. The transition premises here are to the effect that the immediate ancestor of a chicken is also a chicken. The transition between chicken and non-chicken does not occur sharply in one generation.

With this map of the territory of different responses to sorites arguments, we can chart where will explore in the rest of this chapter. All of these options have been explored by philosophers, seeking a justifiable and coherent account of vagueness. OPTION 4 is the most extreme. We will not consider it further here—we leave it on the table as an option of last resort, since if we could avoid having to say that yellow things are red, that everyone is young, that one grain of sand makes a heap, and that chickens have always exsisted, that would be a plus.

We will also not turn to OPTION 1, though this has a longer and nobler heritage. Some hardy souls (like Gottlob Frege and Bertrand Russell, in the early development of logic in the 20th Century) held that the everyday language of speakers like us is so riven with vagueness, with inconsistency and other flaws that there is no hope for logic to genuinely and safely apply when it comes to our everyday talk. Logic only applies to what is expressed in a rigorous, precise language. This view has some merits—or at least it would have if we had some way of grasping what such a precise language might be, if we had some sense that we could express things about a full range of human concerns in such a language. But we have no such confidence that a language with precisely given meaning and sharply defined boundaries could do such a thing beyond the domain of pure mathematics. A language without vagueness is nothing like the languages that you or I speak, and we seem to do quite well in reasoning—and judging arguments to be valid or invalid—in a langage riven with vagueness. Yes, there are difficulties in knowing what to do at the borderlines of vague concepts, but to say that logic only applies to precise expressions is to throw the baby out with the bathwater. We will leave OPTION 1 here and spend the rest of our time with the remaining two options.

Try these on for size: Provide sharp definitions with no vagueness for terms which do the job of words like *chicken, inflation, teenager, affection, New Delhi, neutron star, gene, anger, impressionist, sonata, art deco, elegantly, ...* Or explain how we could live the kinds of lives we do without having concepts like those.

In the next section we will explore OPTION 2—and revise classical logic with an aim to better deal with vagueness, and in the section after

that, we will explore OPTION 3, keeping classical logic and attempting to understand vagueness from the point of view of two-valued semantics.

4.2 | MANY VALUED LOGICS

1. ~~*Logic does not apply* to vague expressions.~~
2. Logic does apply, and the *argument is not valid*.
3. ~~Logic does apply, the argument is valid but *one of the premises is false*.~~
4. ~~Logic does apply, the argument is valid and sound, and therefore *the conclusion is true*.~~

Let's are explore OPTION 2. The simplest and most straightforward way to revise our picture of logic is to say that some statements are *neither true nor false*. Let's go back to our strip, and instead of thinking of 10,000 different propositions, let's just focus on 8, evenly spaced from left to right.

r_1 r_2 r_3 r_4 r_5 r_6 r_7 r_8

When I look at this strip on my computer screen, it eems to me that the first two patches to the left (1 and 2) are clearly red and the last five (4 to 8) are clearly not red (they are orange or yellow). But the patches marked 3 and 4 seem to *me* to be borderline cases.

Maybe they don't seem like that to you. Your eyes, your printer or your computer screen, and your concept *red* may each differ from mine. Adjust things accordingly.

So, if I mark 1 under those statements that I take to be clearly and definitely true, and 0 for those statements I take to be clearly and definitely false, I get this:

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
1	1			0	0	0	0

This is the question for us: What should go in those two gaps? What truth values should we assign to r_3 and r_4?

This three-valued logic is due to the Polish logician, Jan Łukasiewicz. (1878–1956). It is sometimes called $Ł_3$, Łukasiewicz's three valued logic.

4.2.1 | THREE VALUED LOGIC

The first answer we'll consider to this question is that if a statement is not determinately true (1) and not determinately false (0), then it should get a third truth value, which is neither 1 nor 0. Because we conceive of this truth value as poised between 0 and 1, we will use '$\frac{1}{2}$' to denote our third truth value. Łukasiewicz proposed three valued 'truth tables' for this logic, keeping the traditional connectives of two valued classical logic, but interpreting them in a new way. The truth table for negation is the simplest.

ŁUKASIEWICZ THREE-VALUED TRUTH TABLE FOR NEGATION:

A	~A
0	1
$\frac{1}{2}$	$\frac{1}{2}$
1	0

If a formula receives one of the traditional values, 0 or 1, its negation is defined in the usual way. The negation of a proposition with value $\frac{1}{2}$ also receives the value $\frac{1}{2}$. If we point at something which is not clearly red and not clearly non-red, then the claim '*that's red*' and the claim '*that's not red*' both receive the intermediate truth value, $\frac{1}{2}$. Our strip might be understood in this way:

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
1	1	$\frac{1}{2}$	$\frac{1}{2}$	0	0	0	0
$\sim r_1$	$\sim r_2$	$\sim r_3$	$\sim r_4$	$\sim r_5$	$\sim r_6$	$\sim r_7$	$\sim r_8$
0	0	$\frac{1}{2}$	$\frac{1}{2}$	1	1	1	1

As statements range from 1 to 0, through $\frac{1}{2}$, their negations range from 0 to 1, taking exactly the same stop through $\frac{1}{2}$. On this view, when we don't assign 0 or 1 to a proposition, we also don't assign 0 or 1 to its negation, either.

To interpret conjunction and disjunction, it is most useful to picture the truth values as *ordered* from least true to most true, like this:

1

$\frac{1}{2}$

0

This idea, of truth values as coming in an *order* has been influential in more that just three valued logic. It does not even require that the order be 'linear' like it is here: more complex families of values can be understood in terms of truth coming in degrees [6].

Given this order from less true to more true, we can understand conjunction and disjunction. The conjunction of two statements takes the *least true* among the values. A conjunction is as false as either of the conjuncts is false, or as true only as *both* of the disjuncts are true. So, a conjunction of a 1 with a $\frac{1}{2}$ is only $\frac{1}{2}$. A disjunction is as true as *either* of the disjuncts are true, or as false as both of the disjuncts are false. So a disjunction of a 0 with a $\frac{1}{2}$ is $\frac{1}{2}$. The tables for conjunction and disjunction can be written like this:

ŁUKASIEWICZ THREE-VALUED TRUTH TABLES FOR CONJUNCTION AND

DISJUNCTION:

&	0	$\frac{1}{2}$	1
0	0	0	0
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$
1	0	$\frac{1}{2}$	1

$\vee$	0	$\frac{1}{2}$	1
0	0	$\frac{1}{2}$	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
1	1	1	1

Notice that they agree with the tables for the classical two-valued logic at the extremes (when the inputs are either 1 or 0), and whenever $\frac{1}{2}$ is involved, the conjunction of two values is their minimum, and the disjunction is their maximum. We can construct truth tables for formulas in the usual way, except tables now rows numbered have powers of *three* rather than powers of two.

Here are truth tables for $p \vee \sim p$ and $p \,\&\, \sim p$:

p	p	$\vee$	$\sim$	p	p	&	$\sim$	p
0	0	1	1	0	0	0	1	0
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
1	1	1	0	1	1	0	0	1

Notice that $p \vee \sim p$ is not true in every row, and $p \,\&\, \sim p$ is not false in every row. If p gets the intermediate value of $\frac{1}{2}$ then both $p \,\&\, \sim p$ and $p \vee \sim p$ also get that value. If we keep the convention that a tautology is a formula always assigned the value 1, then $p \vee \sim p$ is not a tautology in Łukasiewicz's three-valued logic. If we keep the convention that a contradiction is a formula always assigned the value 0, then $p \,\&\, \sim p$ is not a contradiction in Łukasiewicz's three-valued logic.

FOR YOU: Verify that $A \,\&\, B$ always has the same value as $B \,\&\, A$; that $A \vee B$ always has the same value as $B \vee A$; that $A \,\&\, (B \,\&\, C)$ always has the same value as $(A \,\&\, B) \,\&\, C$; that $A \vee (B \vee C)$ always has the same value as $(A \vee B) \vee C$; and that that $A \,\&\, (B \vee C)$ always has the same value as $(A \,\&\, B) \vee (A \,\&\, C)$.

You can do these with long truth tables (especially for the formulas involving A, B and C—they will take $3 \times 3 \times 3 = 27$ rows!), or you can use the fact that the values of $A \,\&\, B$ and $A \vee B$ are the minimum and maximum respectively of the values of A and of B. So, the value of $A \,\&\, B$ is the minimum of the values of A and of B and that's the same as the value of $B \,\&\, A$. (The same goes for the $\vee$ case—replace 'minimum' by 'maximum'.) The value of $A \,\&\, (B \,\&\, C)$ is the minimum of the value of A and the minimum of the values of B and of C. But this is just the minimum of the values of A, B and C and this is the same as the value of $(A \wedge B) \wedge C$. (The same goes for the $\vee$ case—replace 'minimum' by 'maximum' again.) Finally, the value of $A \wedge (B \vee C)$ is the smaller of the value of A (on the one hand) and the larger of the values for B and for C. This is the same as the larger of these two values: the minimum of A and B, and the minimum of A and C—which is the value of $(A \,\&\, B) \vee (A \,\&\, C)$.

Recall that in classical two-valued logic, $p \supset q$ is equivalent to $\sim p \vee q$. Let's look at the truth table for $\sim p \vee q$. This table would have 9 rows, which is a little long to fit on the page here. Instead, I will present it in a square 3×3 table like I have done for the tables for & and $\vee$.

$\sim p \vee q$	0	$\frac{1}{2}$	1
0	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1
1	0	$\frac{1}{2}$	1

Read the table like this: to find the value of $\sim p \vee q$, look up the *row* labelled with the value of p and the *column* labelled with the value of q. So, if $i(p) = \frac{1}{2}$ and $i(q) = 0$, then the value of $\sim p \vee q$ is $\frac{1}{2}$: that's the $\frac{1}{2}$ you find the $\frac{1}{2}$-row and the 0-column. This looks just like the table for $\vee$, except that the rows are flipped around—since the first disjunct is $\sim p$, not p.

Do you notice anything special with the truth table for $\sim p \vee q$, when compared with that for $p \supset q$ in classical two-valued logic? One thing I notice is that $\sim p \vee p$ is not a tautology. That means, if we take $\sim p \vee q$ to mean something like "if p then q," then it's not a tautology—it's not always true—to say "if p then p." It only gets the value $\frac{1}{2}$ when p gets the value $\frac{1}{2}$. But if I point to a spot in the colour strip in the $\frac{1}{2}$-zone, and say of this patch: "if this is red, it's red," then according to this logic—*if* that "if p then q" is to be understood as $p \supset q$—then I have said something neither true nor false. This strikes many as the wrong way to go. After all, I am very tempted to say, when pointing at patches 3 and 4 "I don't know if this counts as red or not, but I do know that if *this* is red (r_4) then so is *that* (r_3)", but r_3 an r_4 both get the value $\frac{1}{2}$, and the $\sim p \vee q$ conditional gives this the value $\frac{1}{2}$ too. It also struck Łukasiewicz as an incorrect account of the conditional, and he

proposed this alternative table, which differs from the table for $\sim p \vee q$ only in the central spot:

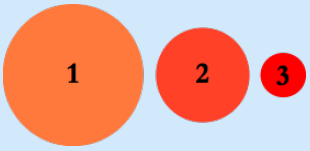
$\supset$	0	$\frac{1}{2}$	1
0	1	1	1
$\frac{1}{2}$	$\frac{1}{2}$	1	1
1	0	$\frac{1}{2}$	1

This table now gives $p \supset p$ the value 1 always, whatever the value received by p. Let's check that $p \supset q$ is equivalent to $\sim q \supset \sim p$—that these two formulas always have the same truth value. For this, I will use a 9-row truth table, so you can see at least *one* large three-valued truth table. Noiice that the two columns for the main connectives are identical.

$p \supset q$ IS EQUIVALENT TO $\sim q \supset \sim p$

p	q	p	$\supset$	q	$\sim$	q	$\supset$	$\sim$	p
0	0	0	1	0	1	0	1	1	0
0	$\frac{1}{2}$	0	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	1	0
0	1	0	1	1	0	1	1	1	0
$\frac{1}{2}$	0	$\frac{1}{2}$	$\frac{1}{2}$	0	1	0	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$
$\frac{1}{2}$	1	$\frac{1}{2}$	1	1	0	1	1	$\frac{1}{2}$	$\frac{1}{2}$
1	0	1	0	0	1	0	0	0	1
1	$\frac{1}{2}$	1	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	$\frac{1}{2}$	0	1
1	1	1	1	1	0	1	1	0	1

FOR YOU: Here are three circles, numbered 1, 2, 3.



Let r_i be the statement: *circle* i *is red*, and let l_i be the statement: *circle* i *is large*.

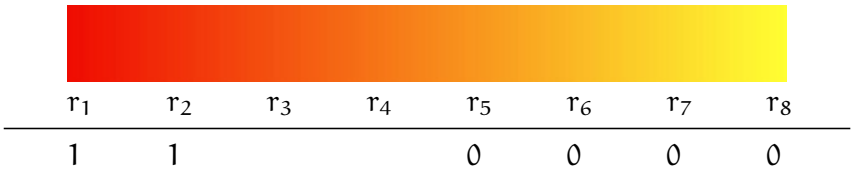
Let's suppose that $i(l_1) = i(r_3) = 1$, $i(l_2) = i(r_2) = \frac{1}{2}$ and $i(l_3) = i(r_1) = 0$.

What are the values of $i(r_2 \supset r_3)$ and $i(r_2 \supset l_2)$? Do these values strike you as *correct*, given what these statements mean?

ANSWER: $i(r_2 \supset r_3) = 1$, and $i(r_2 \supset l_2) = 1$ too. For $r_2 \supset r_3$, this seems right to me, because circle 3 is *more* red than circle 2, so it's true that if circle 2 is red, so is circle 3. On the other hand, $r_2 \supset l_2$ doesn't seem quite as clear-cut. Is it true that if circle 2 is red then it's large? What is the connection between the redness and size of circle 2? It seems that the conditional is true just because the antecedent and consequent have value $\frac{1}{2}$.

The three-valued logic of Łukasiewicz is useful not only in giving an account of what how 'gaps' between truth and falsity might work in the case of vagueness. Łukasiewicz thought it would be of use in giving an account of 'future contingents'—statements which are not determinately true and not determinately false, because they are about an unsettled future. There are things which are *settled* to be true in virtue of history up until the present moment (1), things *settled* to be false by the same means (0) and things not yet fixed—the *open future*. Statements about the open future, in this account, are given the value $\frac{1}{2}$ [8].

It must be said that while Łukasiewicz's three valued logic has genuine appeal for the account of truth value gaps (and the open future, as well as other areas where gaps between truth and falsity are desirable), it is only a *start* when it comes to vagueness. Most people who wish to reject classical logic do not find the jump to *three* truth values particularly natural, when it comes to our strip of colours.



Just as it seems difficult in the two-valued picture to locate the gap between truth and falsity, it seems just as difficult to precisely locate a border between the 1 region and the $\frac{1}{2}$ region, and just as difficult to locate the boundary between the $\frac{1}{2}$ region and the 0 region.

Instead of looking for a sharp division where there doesn't seem to be one, a natural option is to take truth values to truly come in *degrees*. After all, if redness comes in degrees (colours at the left of the strip are *redder* than those to the right), then if the claim that something is red is true if and only if that thing is red, it seems that it makes sense to think of *truth* as coming in degrees too. That's what we'll explore in the next section.

4.2.2 | INFINITELY VALUED LOGIC

DEGREES OF TRUTH: We can expand our picture by allowing truth values from 0 to 1, where 1 is true, 0 is false, and every number between 0 and 1 are intermediate truth values.

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
1	1	0.7	0.3	0	0	0	0

We could fill in our trip of values like this, with 1 for determinate truth, 0 for determinate falsity, and intermediate numbers for truth degrees between these two extremes.

It should be immediately obvious that we cannot use truth tables for truth degrees—formulas can take any of an infinite number of possible values. But we can still think of an interpretation as assigning truth values to formulas, and we can think of the interpretation of complex formulas as determined by the interpretation of their component parts. The interpretation for negation, conjunction and disjunction is straightforward.

- $i(\sim A) = 1 - i(A)$.
- $i(A \,\&\, B) = \min(i(A), i(B))$ — the minimum, the *smaller* of the two values $i(A)$ and $i(B)$.
- $i(A \vee B) = \max(i(A), i(B))$ — the maximum, the *larger* of the two values $i(A)$ and $i(B)$.

FOR YOU: What values can $A \,\&\, \sim A$ take in an interpretation? What values can $A \vee \sim A$ take?

ANSWER: $A \,\&\, \sim A$ can take any value between 0 and $\frac{1}{2}$. $A \vee \sim A$ can take any value between $\frac{1}{2}$ and 1. If $i(A)$ is between 0 and 1, the smaller of the alues of A and $\sim A$ is between 0 and $\frac{1}{2}$, so $i(A \,\&\, \sim A)$ is that value, between 0 and $\frac{1}{2}$. The larger of the values is between $\frac{1}{2}$ and 1, so $i(A \vee \sim A)$ is that value, between $\frac{1}{2}$ and 1.

CONDITIONALS AND TRUTH DEGREES: Conditionals are most interesting in the presence of truth degrees. There are many different ways one could interpret conditionals. Łukasiewicz interprets them in this way:

- If $i(A) \leqslant i(B)$ then $i(A \supset B) = 1$
- If $i(A) > i(B)$ then $i(A \supset B) = 1 - i(A) + i(B)$

The rules mean this: if B is at least as true A, then the conditional $A \supset B$ is completely true. If, on the other hand, B is *less* true than A, then the conditional $A \supset B$ falls below 100% truth to the same extent that B drops below A. That is, $i(A \supset B) = 1 - i(A) + i(B) = 1 - (i(A) - i(B))$. That means, for example, that the only way that the conditional $A \supset B$ can be determinately false is if $i(A) = 1$ and $i(B) = 0$. If, for example, $i(A) = \frac{2}{3}$ and $i(B) = \frac{1}{2}$ then $i(A \supset B) = 1 - \frac{2}{3} + \frac{1}{2} = \frac{6}{6} - \frac{4}{6} + \frac{3}{6} = \frac{5}{6}$.

FOR YOU TO TRY: Consider the formula $(A \;\&\; (A \supset B)) \supset B$. What values can it take?

FOR YOU: The argument from A and $A \supset B$ to conclusion B *is not* absolutely valid. If $i(A) = \frac{1}{2}$ and $i(B) = 0$ then $i(A \supset B) = \frac{1}{2}$ so $\frac{1}{2} \leqslant i(A), i(A \supset B)$ but $\frac{1}{2} \not\leqslant i(B)$.

WHEN IS AN ARGUMENT VALID? The important thing when it comes to diagnosing the sorites paradox is to understand the status of the sorites *argument*. What can we say about the argument? Now that we have so many truth degrees, there are many different things we could say about the status of an argument. One option is to do what we have done in the two-valued case. We can say that an argument is valid iff whenever the premises are true, the conclusion must be, too. But now, we have many different options for what counts as 'true.' There is absolute 100% truth, but there is also truth in the ense of crossing some slightly less strict threshold. If we think of all of the values between the threshold t and 1 as being good enough to count as *true* then we have a kind of criterion for validity.

- An argument from Σ to B is t-*valid* iff whenever $i(A) \geqslant t$ for each A in Σ then $i(B) \geqslant t$ too.
- Think of t as the threshold to count as true.
- If the argument from Σ to A is t-valid we'll write '$\Sigma \vDash_t A$.'
- An argument is *absolutely valid* iff it is t-valid for every threshold value t.

It's a fact that an argument rom Σ to A is absolutely valid if and only if for every interpretation i, the value $i(A)$ conclusion is at least as great as the value of the least of the premises. (If this criterion is satsified, then clearly if the premises are all over a threshold t so is the conclusion. If this criterion is not satisfied on some interpretation, then let t be the least of the values of the premises, then the argument is indeed not t-valid.)

In the case of an infinite number of premises, we just need the *infimum* of the premises—the greatest lower bound.

FOR YOU: Is the argument from A and $A \supset B$ to conclusion B absolutely valid?

It can take any value from $\frac{1}{2}$ to 1. In particular, if $i(A) = \frac{1}{2}$ and $i(B) = 0$ then $i(A \supset B) = \frac{1}{2}$ and so, $i(A \;\&\; (A \supset B)) = \frac{1}{2}$ too. So $i(A \;\&\; (A \supset B) \supset B) = \frac{1}{2}$. There is no way it can take any value lower than $\frac{1}{2}$.

BACK TO OUR STRIP Now return to the sorites strip and let's look at the form of the sorites argument. If we select six statements r_1 to r_6 where $i(r_1) = 1$, $i(r_2) = 0.8$, $i(r_3) = 0.6$, $i(r_4) = 0.4$, $i(r_5) = 0.2$ and $i(r_6) = 0$.

$$r_1, r_1 \supset r_2, r_2 \supset r_3, r_3 \supset r_4, r_4 \supset r_5, r_5 \supset r_6 \nvDash_{80\%} r_6$$

Notice the premises are all 80% true or greater. $i(r_1) = 1$, while $i(r_1 \supset r_2) = 1 - 1 + 0.8 = 0.8$, $i(r_2 \supset r_3) = 1 - 0.8 + 0.6 = 0.8$, etc. While the conclusion is true only to degree 0. The argument is invalid at the threshold of 80%. The premises are all true to a very high degree, while the conclusion is not true to that degree at all. In other words

Small errors can add up!

In general, each of the transition conditionals $r_i \supset r_{i+1}$ is very close to 100% true, because the truth values from left to right never drop down in a dramatic way: the steps from truth to falsity are gradual. So the conditionals are all true to a very high degree. However, the conclusion is very much less true—in this case, it is completely false. The argument is *invalid* because the premises are all highly true but the conclusion is completely false.

This is the *fuzzy logic* analysis of the sorites argument. The response to the sorites paradox in this form is that the premises all seem true—and all are true to a high degree. The conclusion seems false—and is false to a very high degree. But the argument is actually *not* valid, despite the verdict of classical logic.

4.2.3 | QUESTIONS CONCERNING TRUTH DEGREES

There is a great deal to like in this analysis. In a sorites argument, the premises do seem true and the conclusion seems false, and this analysis respects that. It provides a novel account of validity which allows for the sorites argument to be invalid. However, some questions remain to be answered by anyone who would like to take this route:

- Are the connective rules *correct*? We simply *stated* the connective rules, for conjunction, disjunction, negation and the conditional. Are they the correct rules? Where do they come from? How would we know if we had the correct rules? With more degrees of truth there are many other possible rules a connective could satisfy. Are *these* the right ones? What makes them correct?
- Are there *too many borderlines*? We criticised two and three valued logic by saying that it forces us to find the particular borderline

between the red and the non-red; or between the red and the gappy section, and between the gappy section and the non-red. But infinitely many truth degrees really bring us infinitely many borderlines. Now we need to find the spot where it passes 50%, as well as the spot where it's 30% and any other number, too. If every statement has a truth value in between 0 and 1, what makes a borderline statement have truth value $0.141592654\cdots$ rather than $0.141592653\cdots$? What does the difference berween these values amount to in nature?

- Do all truth degrees *compare*? If the claim "this is red" has a truth value between 0 and 1, and the claim that "that is large" also has a truth value in this range, then either the one is truer than the other, or they have exactly the same truth value. But why should the degree that *one* thing is *red* compare precisely to the degree that *another* is *large*? There seems to be nothing in nature that makes it that these degrees of truth should be able to be compared with each other. It is one thing to say that truth values come in degrees. It is another to say that those degrees are neatly ordered in exactly the same way as the numbers between 0 and 1.
- Isn't *modus ponens*, the inference from A and $A \supset B$ to B valid, if *anything* is valid? The crucial step in this defusing of the sorites paradox is the judgement that the sorites argument is invalid. However, the argument is just a repeated *modus ponens* (the inference from A and $A \supset B$ to B). This inference is about as fundamental to logic as anything is (it is hard to imagine reasoning using 'if … then …' in your vocabulary without appealing to this inference). Yet, Łukasiewicz's logic (either in its three valued or its infinitely valued versions) takes his inference to be invalid. Doesn't this count as a mark against the *logic*, if you cannot give a diagnosis as to why the inference is invalid, and how one could live without it?

Any view of vagueness and the sorites paradox that responds by taking what we have called OPTION 2—that logic applies to the sorites argument, the premises of sorites arguments can be true without the conclusion following, and that the argument is invalid—must involve answering questions like these. Łukasiewicz's approach to truth values beyond 0 and 1 is not the only way to develop OPTION 2. There are many other ways to go beyond classical propositional logic. But to do so is to prompt questions like these, important questions about a logical system is to be understood and applied.

The approaches in is last section have explored life beyond two-valued classical propositional logic. This is not the only response to the sorites paradox. As we have seen, there is another option you can take—you can attempt to take what we have called OPTION 3, and keep fidelity with our two-valued semantics, but understand it in a new way.

4.3 | RETAINING CLASSICAL LOGIC

1. ~~*Logic does not apply* to vague expressions.~~
2. ~~Logic does apply, and the *argument is not valid*.~~
3. Logic does apply, the argument is valid but *one of the premises is false*.
4. ~~Logic does apply, the argument is valid and sound, and therefore *the conclusion is true*.~~

Let's revisit the strip with our original truth values filled in for those patches I could determine as red or as not red. This time, instead of filling in the remaining positions with values other than 0 or 1, we'l consider filling them in with 0 and 1, but in all the different possible combinations. Here is the result:

r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
1	1	0	0	0	0	0	0
1	1	1	0	0	0	0	0
1	1	1	1	0	0	0	0
1	1	0	1	0	0	0	0

If you look at each of these evaluations, you'll see that *one* of them stands out as unacceptable. The last row looks *very* strange. According to this row, patch 3 is not red (r_3 is false), but that patch 4 (which is *less* red than patch 3) *is* red (r_4 is true). This is clearly a violation of what we mean by 'red,' so I will leave that evaluation out, and what we are left the three remaining evaluations, presented in the following table:

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
i_1:	1	1	0	0	0	0	0	0
i_2:	1	1	1	0	0	0	0	0
i_3:	1	1	1	1	0	0	0	0

The interpretations i_1, i_2 and i_3 are somehow all equally acceptable—we call them the APPROPRIATE VALUATIONS.

> In a case of vagueness, and APPROPRIATE VALUATION takes to be *true* everything we know to be true, *false* everything we know to be false. (sTypically in cases of vagueness and borderline cases, there is more than one appropriate valuation.)

None of the appropriate valuations is a description that fits precisely and completely with what we know to be true. On the other hand, none of them violates what we know to be true about what is red and what is not. They agree with the values we could determine (those

are the red zeros and ones) and they fill in the rest in any way that is acceptable. This picture, of the world (and the way our language relates to the world) as being represented by a *range* of different evaluations, will be central in our understanding of how to make vagueness cohere with classical logic.

4.3.1 | SUPERVALUATIONS

OPTION 3A: BORDERLINES—SUPERVALUATIONS There are a number of different ways of interpreting these valuations i_1, i_2 and i_3 in this case—and the range of valuations found in other vagueness cases. The first approach that we'll examine today goes like this:

- The world is modelled by *all* of the appropriate valuations.
- Statements are either *super-true*, or *super-false*, and in the *gap*.
  - A statement is super-true if it is true in *all* of the appropriate valuations.
  - A statement is super-false if it is false in *all* of the appropriate valuations.
  - A statement is in the gap if it true in some appropriate valuations and false in others.

The result is a three valued logic, not unlike Łukasiewicz's three valued logic. However, the result is *not* really as much of a revision from classical logic as a genuine three-valued logic.

- Two-valued *tautologies* are super-true in all supervaluations.

This fact is straightforward to demonstrate. If a formula A is a tautology in two-valued logic, then it is true in every interpretation whatsoever, including all of the appropriate ones.

So, for example, $r_3 \vee \sim r_3$ is a tautology—and is super-true according to the appropriate valuations in our example, even though r_3 is in the gap and is not super-true or super-false. (Every formula of the form $A \vee \sim A$ is super-true whatever the status of A.) This means that we cannot provide a truth table for disjunction in the way that we can in Łukasiewicz's three valued logic. If we represent super-truth by 'T', super-falsity by 'F', and the gap by 'N' (neither true nor false) then the most we can fill out in the table the most we can do is this:

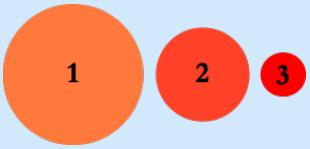
$\vee$	F	N	T
F	F	N	T
N	N		T
T	T	T	T

The gap in the middle can be filled with either N or T, depending on what formulas are used. If p is N (true in some appropriate valuations, false in others), then $p \vee p$ is N (it is true in the same valuations as p). It's a disjunction of two Ns that makes an N. But $p \vee \sim p$ is T (it's true

in all valuations). It's a disjunction of two Ns that makes a T. This means that disjunction is not 'truth functional' when it comes to the values T, N and F—that is, its truth value of the disjunction doesn't depend on the truth values of its constituents. This is the same is true for conjunction:

&	F	N	T
F	F	F	F
N	F		N
T	F	N	T

FOR YOU: Here are three circles, numbered 1, 2, 3.



Let r_i be the statement: *circle* i *is red*, and let l_i be the statement: *circle* i *is large*.

Suppose there are four acceptable valuations all make make l_1 and r_3 *true* and l_3 and r_1 *false*, and r_2 and l_2 can be *true* or *false*, independently.

What is the status of the following four formulas (r_2, $r_2 \vee {\sim}r_2$, $r_2 \vee l_2$, $r_2 \supset r_3$) are super-true (T), in the gap (N) or super-false (F). Do these values strike you as *correct*, given what these statements mean?

ANSWER: r_2 is in the gap (N), $r_2 \vee {\sim}r_2$ is super-true (T), $r_2 \vee l_2$ is in the gap (since it is true in the valuation where either r_2 or l_2 is true, and false in the valuation where r_2 and l_2 is false), and $r_2 \supset r_3$ is true, since r_3 is true in every valuation.

On the supervaluation view, each of these valuations count as different possible ways the vague concepts can be interpreted. No matter what 'red' means, every patch is either red or not red. On the other and, the claim that a patch on the border is red is neither true nor false, because the border can be drawn on either side of that patch.

SUPERVALUATIONS AND THE SORITES That is one way to understand vagueness from within the framework of classical two-valued logic—vague borderlines are places where we have freedom in our concepts, where items can be counted equally as 'in' or 'out'. Let's see what this says about the sorites argument. This argument form is *valid* according to classical logic, and so, it's valid here, because what we have is classical

logic under another guise. The argument has this shape:

$$r_1, r_1 \supset r_2, \ldots, r_i \supset r_{i+1}, \ldots, r_j \supset r_{j+1}, \ldots, r_{9{,}999} \supset r_{10{,}000}$$

If there is more than one acceptable valuation, one where r_1 to r_i is true, and another where it is r_1 to r_j that are true, then the premises are not all true—they are not all super-true, at least. In one valuation, $r_i \supset r_{i+1}$ is false, because the transition in this valuation from red to non-red occurs at patch i. In another, $r_i \supset r_{i+1}$ is true, but $r_j \supset r_{j+1}$ is false, since the transition occurs at patch j. In every valuation, *some* premise is false, because the transition from true to false occurs somewhere. However, it is not the case that any premise is super-false, since no premise is false in every valuation. So what we have is the following:

- For every valuation in the supervaluation, there is a borderline, and a false premise $r_i \supset r_{i+1}$.
  - but for each valuation, the false premise is different.
- The conclusion is super-false, and the premises are either super-true or in the gap.
- The argument is valid, but it's not the case that all premises are super-true.
- The general statement "there is a spot which is red and the next one isn't" is super-true too.

It's not the case that the premises are all *true*—they're not all super-true. However, they they are true in *almost all* of the valuations.

4.3.2 | EPISTEMICISM

One issue with the account of truth assumed in supervaluationism is the fact that the statement "there's a patch where it's red and the next one isn't" counts as super-true—since in every acceptable valuation there is a patch which is red and the next one is not—while the ultimate picture of reality divides the scene into those things which are red, those which are not, and those which are in the gap. Truth-according-to-the-theory says that reality is *precise* and *sharp* (it *says* there is no gap between truth and falsity—it says that there is a sharp transition point) while it is *incomplete* (it does not pin down where that transition point is). There is a tension in this view. There is another way to read the same formal features (the same collection of acceptable valuations) but interpret them differently. This view does not have the same tensions that are had by supervaluations. (It will have *different* tensions and difficulties

OPTION 3B: BORDERLINES—EPISTEMICISM On this view, the three acceptable valuations do not equally represent the way the world is. Rather, they represent what we can *know* about the world. We don't *know* which of i_1, i_2 and i_3 best represents the world, but we know that it's

one of them. This is what makes epistemicism deserve that name—it is a view of vagueness that connects it most squarely with talk of our *knowledge*. On this view, when we consider the strip of colours:

'Episteme' is the Greek word for knowledge.

	r_1	r_2	r_3	r_4	r_5	r_6	r_7	r_8
i_1:	1	1	0	0	0	0	0	0
i_2:	1	1	1	0	0	0	0	0
i_3:	1	1	1	1	0	0	0	0

- The acceptable valuations are now understood as ways the world could be *as far a we can know*.
- The world is *one* of the valuations…but we don't know which.
- So, the world outstrips our knowledge of it.
- Our possible knowledge is modelled by all of the valuations.
- Statements are *knowable* and *refutable*, and *undecidable*.
- All tautologies are knowable.

On this view every patch is either red or not red—(the statement $r_3 \vee \sim r_3$ is true, no matter which patch we are considering), and indeed, the statement r_3 is either true or not true—but we don't know which, because we don't know if the world is like i_1 or i_2 or i_3.

EPISTEMICISM AND THE SORITES So, when confronted with the sorites paradox, the epistemcist says the following things:

$$r_1, r_1 \supset r_2, \ldots, r_i \supset r_{i+1}, \ldots, r_j \supset r_{j+1}, \ldots, r_{9{,}999} \supset r_{10{,}000}$$

- The argument is valid (it is valid in classical logic), but one of the premises is false.
- There *is* a borderline between the red and the non-red, but we don't (and *can't*) know what it is.

Why *can't* we know? It's because of how we learn the concept of 'red' with our visual system and the bounds of the abilities of our visual system to discriminate colours.

As before, this leaves as many qustions as it answers. Some of the questions that are asked are as follows.

- There's a point which is red and its neighbour isn't, *despite looking indistinguishable*. How can that be? Doesn't this contradict the idea that
- (What decides that *this* valuation models the world?)
- Is this still *vagueness*, when there is a point which is red and the next one isn't?

There is much current research in the philosophy of logic around topics of vagueness and the way to understand truth values and logical interpretations. A good guide to the topic is Timothy Williamson's book *Vagueness* [14], and his book *Knowledge and its Limits* is a good discussion of an account of knowledge and its logic, and applications to different areas of philosophy [15].

4.4 | OTHER TOPICS IN THE PHILOSOPHY OF LOGIC

The topic of vagueness is not the only topic connecting logic and philosophy. There are many others that are worth exploring. We have focussed on looking at one of the fundamental assumptions in the model theory of logic—the status of the truth values in the truth table technique, and how they are to be understood in the presence of vagueness. There are other philosophical questions in and around logic. These take at least two different forms.

- Questions from *Philosophy*–issues in metaphysics, epistemology, philosophy of language.

- Questions from *Logic* itself—what is the right logical system (is Łukasiewicz's logic correct, or is classical two-valued logic correct? What about many of the other logical systems? How do we tall which is correct?) Is there just one correct logical system or are there many [4]? When we give an account of logical consequence, do models come first in the definition, and then proofs come along as a way to give an account of what is valid given the prior definition from models, or should proofs come first and models trail along after? Do proofs come first or models?

These, and more questions like them, form the landscape of applications of logic to philosophy—and *vice versa*. It will be a rich relationship between these two disciplines for years to come.

LINGUISTICS: IMPLICATURE AND IMPLICATION

5

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

- Two kinds of inferences which can be drawn from premises: *entailment* (which can be understood in terms of valid argument) and *implicature*: understand these and the differences between them.
- Grice's Cooperative Principle, the maxims of conversation (*Quantity*, *Quality*, *Relevance*, *Manner*), and the way that implicatures are generated on the basis of them (assumption that speaker is observing the maxims, or else assumption that speaker is flouting or violating a maxim).
- Non-truth-conditional aspects of the meanings of natural language connectives in comparison to the connectives of propositional logic, and how these can be accounted for by a theory of implicature and the pragmatics of cooperative discourse.

SKILLS

- Decide on the basis of the tests we have looked at whether an inference is an entailment or an implicature and provide evidence for your decision.
- Identify which maxims of conversation are involved in the generation of specific implicatures, and how those implicatures are generated.
- Use the theory of implicatures to give an account of how individual connectives are *used* in concrete situations may differ from a straightforward reading of the truth table interpretations of those connectives.

5.1 | LOGIC AND LINGUISTICS

WHAT DO LINGUISTS DO? Linguistics is the study of *language* in *all* its aspects. There are plenty of different areas of language to study.

- *Phonetics* and *phonology* — sounds in language production.
- *Morphology* — morphemes, the individual units in languages.

We gratefully acknowledge the generous assistance of our colleague, Lesley Stirling, who developed lectures in UNIB10002 from which these notes are derived.

- *Syntax* — the rules of formation of linguistic complexes, like sentences.
- *Semantics* — how words and sentences get their meanings
- *Pragmatics* — the way language is *used* in different contexts, and how use contributes to interpretation
- *Discourse analysis* — the analysis of larger texts and discourses
- ...

This study is rich and complex and varied. Logic connects with language in a number of ways, to a great degree in syntax, semantics, and to a lesser extent to pragmatics and discourse analysis. One view about the relationship between logic and language that has grown in prominence in the second half of the twentieth century arises out of what has come to be known as the "Chomsky Hypothesis"

Noam Chomsky
(1928-)

CHOMSKY'S HYPOTHESIS (1957):
Natural languages can be described as formal systems.

This is a plausible view, given a wide enough understanding of what counts as a formal system. The thesis doesn't mean to say that natural languages are like the language of propositional logic. No, that is altogether too simple and straightforward. The languages that we speak, on the other hand, are so complex that a range of different techniques must be used to model them effectively.

"There is in my opinion no important theoretical difference between natural language and the artificial language of logicians; indeed I consider it possible to comprehend the syntax and semantics of both kinds of languages within a single natural and mathematically precise theory."

— Richard Montague, "Universal Grammar" 1970 [10]

It's important to understand the difference between how linguists approach language to how philosophers and other logicians approach language. Linguists are much more *empirical*.

"Linguists are like vacuum cleaners! Philosophers are rather like black holes.

Philosophers react to every theory by constructing arguments against it.

Linguists react to every theory by taking it in and using it to ex-

plain some of their millions of examples." — The philosopher David Kaplan in conversation with linguist Barbara Partee, quoted in Partee, "Reflections of a formal semanticist as of Feb 2005"

But when one develops an empirical science, one needs *data* upon which to base your theory. Where is the data of linguisics found?

Barbara Partee

- Brought *linguistics* and *formal logic* together.
- Developed and promoted *Montague semantics*.
- Helped establish connetions between *formal syntax* and *formal semantics*.

AN EMPIRICAL SCIENCE NEEDS DATA

- *Corpora* of natural language examples
  - Books
  - Newspapers
  - The Internet

There are many different sources of language data to trawl, especially in the age of the internet and the explosion of large data sources. However, public written and recorded data is only one form of language data available to us. Another important form of information for linguistics is found in the judgements of native speakers. These come in various forms:

- Native speaker intuitions
  - about *grammaticality*
    *Who did a book about impress you?
    *This sentence no verb.
  - about *meaning* and *meaning relations*
    I saw relations in my underwear.
    I gave her cat food.

Linguists will often star a sentence to indicate that it is somehow defective. In this case, the starred sentences are ungrammatical, though in some sense, understandable.

They can be intuitions about whether language expressions are grammatically correct or not (such as the two starred sentences there). More interesting for us are the intuitions we have about how individual sentences are to be interpreted. For example, the claim about seeing elephants in your underwear can be interpreted in an least two ways. First, the elephants may be majestic beasts found in the Afrian plains, and I may have seen them, while wearing underwear, on Safari in Africa. On the other hand, it could have been a stuffed toy elephant, which for some reason was hiding inside my own underwear.

I saw elephants
(*while I was*) in my underwear

I saw elephants
(*which were*) in my underwear

Similarly, the claim "I gave her cat food" could have various meanings. It could be that I am saying of her cat, that I gave food to that cat. Or it could be of *her* that I gave catfood. In this case, all the words have the same meaning (and we are not interpreting words in different ways on different readings), but it is just a matter of how the words are to be combined in the syntactic structure of the sentence.

I gave (her cat) food

START WITH YOUR OWN JUDGEMENTS ABOUT MEANING It's alwas helpful to *start* with your own judgements about the meanings of words and sentences, but never stop there alone. We want to use our own internal sense of how it is that we are interpreting our own language, but never be captivated by just that sense—if a theory is going to work, it will work not only for the theorist, but for all users of the language.

> "Formalization is an excellent thing in moderation. When there's too little, claims are fuzzy and argumentation is sloppy. But there can be too much formalization, or premature formalization. So one shouldn't hesitate to share ideas in an informal state; looking at things from many points of view may help a good formalization emerge." — Barbara Partee "Reflections of a Formal Semanticist as of Feb 2005"

5.2 | ENTAILMENT AND IMPLICATURE

Our focus in this chapter is the topic of implicatures—the way we can say one thing in order to lead our hearers to believe another. This is an issue in the semantics, the pragmatics of language, and discourse analysis.

– *Phonetics* and *phonology*

- *Morphology*
- *Syntax*
- *Semantics*
- *Pragmatics*
- *Discourse analysis*
- ...

language use is shot through with *inference*. People conclude one thing on the basis of hearing another. Often, these inferences are so idiosyncratic and perculiar to a particular context, that outsiders can find it very difficult to interpret. Here is a conversation Jen and Greg might have about their friend Lesley.

'Lesley' here is our colleague, Dr. Lesley Stirling, a linguist at the University of Melbourne.

Jen: That's Lesley now.
Greg: What is she doing?
Jen: Well, she didn't kick the cat.
Greg: OK, I'll slice the lemon.

What on earth could this discourse mean? There are strange jumps between claims that are made that are very hard to interpret, without particular local knowledge of the situation. Once you know those local facts, it is much easier to interpret what's going on.

HOW WE GET FROM A TO B Here are two of the inference steps used by us in this little discourse. First, we know that Lesley has been out playing tennis, and that when she loses, she usually takes out her frustrations on the cat. So, we make the following inference:

P: Lesley didn't kick the cat.
C: *So*, Lesley won at tennis.

Then, we make another inference, because we know how Lesley likes to celebrate when she wins at tennis.

P: Lesley won at tennis.
C: *So*, Lesley will want a Gin and Tonic.

The inference steps made in this discourse are *not* valid arguments. They have plenty of counterexamples—there are many *possible* situations in which Lesley wins at tennis without wanting a Gin and Tonic. There is nothing that compels one to the conclusion from the premise

on the basis of logic alone. This is a *feature*, and not a *bug*. We are not making *mistakes* when we reason like this. We are exploiting the shared knowledge we have of our circumstances.

ENTAILMENT AND WORD MEANING This kind of shared knowledge can be of various kinds. An important kind of shared knowledge we have with others in conversation is our knowledge of meanings of words in the language we speak. If we make the following inference:

P: There's a cat on the mat.
C: There's an animal on the mat.

Then clearly, this is not merely a *contingent* inference. As a matter of necessity, if a cat is on the mat, an animal is on the mat. This isn't *deductively and logically* valid—it depends on the non-formal connection between the meanings of the words 'cat' and 'animal', but if we add the obvious extra premise to our argument—

P: There's a cat on the mat.
P′: If something is a cat, it's an animal.
C: There's an animal on the mat.

—the result is a valid argument. What's more, the extra premise we added is the kind of thing you can know if you know the meanings of the word 'cat' and 'animal' as we are using them. Here is another example of the same sort of thing:

P: I painted the door red.
P′: Red is a colour.
C: I painted the door a colour.

Without the second premise P′, this would not be a valid argument. With it, the argument is valid. This premise is also something that you *know* when you know the meanings of the words 'red' and 'colour'. This is one of the kinds of things you are expected to know as a dialogue partner—we rely on each other to understand the meanings of the words we're speaking.

Premises are said to ENTAIL a conclusion if the argument from the premises to the conclusion, *augmented with premises true in virtue of word meanings*, are valid.

Entailments are not strictly speaking logically valid, but they come close. The premises necessitate the conclusion, and there are no *counterexamples* to valid arguments in situations where our words work as we mean them to.

Here are some inferences, classified into entailments and non-entailments.

[ENTAILMENT] Madison drives a Toyota Prius.
So, Madison drives a car.

[ENTAILMENT] Charlotte ate the cake, but got better.
So, Charlotte ate something.

[NOT AN ENTAILMENT] Clancy's father didn't cook any supper.
So, Clancy didn't get any supper.

The first and second entailments involve the hidden premises (A Prius is a car; a cake is soemthing). The third argument is clearly not an entailment. Clancy might have made her own supper, or her mother may have made her supper.

FOR YOU: Which of these are entailments?

(*a*) Aragorn is tall, and Gandalf is taller than Aragorn.
So, Gandalf is tall.

(*b*) Charlotte ate the cake, but got better.
So, Charlotte is not sick any more.

(*c*) Yesterday was Tuesday.
So, today is Wednesday.

(*a*) and (*c*) are entailments, with extra premises (anyone taller than a tall person is tall; and, the day after Tuesday is Wednesday, and the day after yesterday is today). (*b*) is not. You can get better and then fall sick again.

IMPLICATURES The philosopher *Paul Grice* (1913-1988) proposed a theory of *conversational implicatures*. An *implicature* is something that we *mean* or *imply* or *suggest* by way of saying *something else*. When you know basic logic, you can get a good understanding of how implicature works, and the different ways we can say what we mean. Let's consider an example in a straightforward dialogue:

A DIALOGUE Consider the following short dialogue. What have you learned from what I have said in answer to your question?

> Q: Do you have any children?
>
> A: I have a son.

At the very least, you've learned that I have a son. And indeed I (Greg), do have a son.

Have you learned that I don't have any other children as well? All I said is that I have a son. I have not said anything about whether I have any other children.

Yet it seems to me that you would have been well within your rights to conclude from what I said that the total number of children I have is one—my son. Is that right? And if it is right, how does this work?

IMPLICATURES An implicature is part of what the speaker *means*, but not part of what is *said*. I have *said*, in this dialogue, that I have a son. Perhaps what I meant (and what I meant for you to conclude) was that I have no other children. Here are some other example implicatures you might make in a dialogue:

> *A*: Can I trust him?
>
> *B*: Well, he's never been convicted of a crime.

From this little interchange, I woul think that the right thing to conclude is that you *can't* trust him. If the best thing you can say for him is that he has never been convicted, this is not much reassurance.

If you were to receive the following recommendation as the only positive claim made in a letter of reference for a candidate for a philosophy professorship, what would you conclude?

> Jones dresses well, and writes grammatically correct English.

I would conclude that Jones is no good at Philosophy. Even though this hasn't been explicitly said, it is what is left out (any praise for Jones' ability at philosophy) spoke volumes.

So, we make implicatures often in discourse. They can be distinguished from entailments by a number of distinguishing features. Implicatures can be *cancelled*. If the reference went on to say 'in addition to dressing well and writing correct English, Jones is the greatest philosopher of the 21st Century', then the reference writer is not *retracting* anything already said, but now the implicature can't be drawn. This *cancels* the implicature.

Implicatures are also context dependent. I would *not* be saying that Jones is no good at philosophy if I said the words "Jones dresses well, and writes grammatically correct English" in a different context, you may draw a very different conclusion. Finally, an implicature can be *reinforced* without it sounding like you are repeating yourself. I could say "Jones dresses well, and writes grammatically correct English, but apart from that, I've got nothing positive to say—I think he's a terrible philosopher."

ENTAILMENTS	IMPLICATURES
Cannot be *cancelled*.	Can be *cancelled*.
I have one daughter. And I don't have any children.	I have one daughter. And I have a son, too.
Hold regardless of context.	Depend on context.
Cannot be *reinforced* without sounding odd	Often are *reinforced*.
I have one daughter. And I have one child.	I have one daughter. And that's my only child.

HOW DO IMPLICATURES WORK? Paul Grice proposed that the *Cooperative Principle* governing the use of language in dialogue is central to the behaviour of implicatures.

COOPERATIVE PRINCIPLE: Make your conversational contribution such as is required, at the stage at which it occurs, by the accepted purpose or direction of the talk exchange in which you are engaged

NOTE: Neither the *cooperative principle* nor the 'maxims' which unpack it are to be taken *prescriptively*. They are *not* rules we are exhorted to follow. These 'maxims' spell out what is involved in cooperation in a discourse.

GRICE'S MAXIM—QUALITY: Try to make your contribution one that is true.

1. Do not say what you believe to be false.
2. Do not say that for which you lack adequate evidence.

GRICE'S MAXIM—QUANTITY: Say what you need to, but no more.

1. Make your contribution as informative as is required (for the current purposes of the exchange).
2. Do not make your contribution more informative than is required.

GRICE'S MAXIM—RELATION (OR RELEVANCE): Be relevant.

1. Contribute to the topic at hand.
2. Don't change topic arbitrarily.

GRICE'S MAXIM—MANNER: Be efficient in what you say

1. Avoid obscurity of expression.
2. Avoid ambiguity.
3. Be brief (*avoid unnecessary prolixity*).
4. Be orderly.

From these maxims, we can see how implicatures are generated. They are generated in a number of different ways, by our assuming that the speaker is observing the maxims, or attempting to understand what is going on when someone is flouting those maxims.

ON THE ASSUMPTION THAT THE SPEAKER IS OBSERVING THE MAXIMS

Q: Is Jeanette home?
A: Her light is on.

She is home, or is likely to be home: *Relevance*

In this case, when we ask if Jeanette is home, the answer (her light is on) is taken (by Relevance) to have some bearing on the question. If it does have bearing on the question, it will be in the affirmative. The light being on is reason to believe that she's home.

Q: Do you have any children?
A: I have one son.

That's my only child: *Quantity*

If I were answering here and complying with the maxim of *quantity*, then I would say enough to answer the question, and no more than is needed. The speaker is keen to know about my children. If I merely say that I have one (when I have more than one), this is saying *less* than in required for the purpose of addressing the issue. So, assuming that I was following the maxim of Quantity, you can assume that I only have one child.

THROUGH AN APPARENT VIOLATION OR FLOUTING OF A MAXIM We can also make sense of cases where conversation partners apparently flout the maxims.

Q: How's your thesis going?
A: It's nice outside.

It's not going well: *Relevance*

In this case, it's clear that the relevance principle has been violated. We reason as follows: she violates relevance, there must be a reason. A plausible reason is that the answer is too painful to mention right now, and changing the subject is warranted. If that's the case, it's because the thesis is not going well.

Miss Jones produced a series of sounds intended to correspond to the *Aria* from *Rigoletto*.

She sang badly: *Manner*

In this case, the maxim of manner has been violated, to say something apparently needlessly long and complex, and to say something less than "she sang the *Aria* from *Rigoletto*." To say that the sounds were intended to correspnod to the Aria is to draw attention to the way it does not count as actually singing it. It is an insult.

5.3 | IMPLICATURE AND THE CONNECTIVES

The previous section gave us some basic concepts concerning implicatures. In this section, we'll see how they can be applied to help understand puzzles concerning the behaviour of the logical connectives.

PROPOSITIONAL CONNECTIVES Here are the familiar propositional connectives we have studied.

Conjunction	&	and
Disjunction	$\vee$	or
Conditional	$\supset$	if …then …
Biconditional	$\equiv$	if and only if
Negation	$\sim$	not

There are little puzzles about each of these connectives. We'll start with the simplest one, conjunction.

EXAMPLE: & AND AND

p	q	p & q
0	0	0
0	1	0
1	0	0
1	1	1

Does the logician's '&' actually *mean* 'and'? Is *this* what 'and' *means*? Or does the concept of conjunction have a meaning that goes beyond this truth table?

VARIOUS USES OF AND IN ENGLISH There is no doubt that we use 'and' in a variety of ways that go beyond what a truth table can capture. Think of the different things you understand from these sentences?

a. We went out and had dinner.
b. We had dinner and went out.
c. We went out. We had dinner.
d. We went out and had dinner, although not in that order.

It's clear that in the use of 'and' in these statements, we have behaviour which goes beyond the simple truth table. How can we exaplain the bwhaviour of 'and' in these sentences? One proposal is to use Grice's maxim of manner:

> Be efficient in what you say:
>
> 1. Avoid obscurity of expression.
> 2. Avoid ambiguity.
> 3. Be brief (*avoid unnecessary prolixity*).
> 4. Be orderly.

When it comes to truth tables, 'p and q' means the same as 'q and p'. However, they are not the same *sentences*. We can choose either for whatver purpose is in hand. Given that we process the words in a sentence in order, it is generally helpful to choose the sentence whose order matches the order in which the events described happens. If I say 'I went out and had dinner' you tend to picture things happening in the order in which you hear them. On this view, that temporal ordering is not an effect of the *meaning* of the proposition (in the sense of the conditions in which the statement is true), but rather, an important side effect of the way we process sentences. In this way, although 'and' does not *mean* 'and then,' the maxim of manner explains why we choose one ordering over another.

CONDITIONALS AND MATERIAL IMPLICATION

- $p \supset q$ is false only if there is an explicit counterexample: if p is true, and q is false.
- Many people find this problematic as an analysis of the meaning of English *if* …*then* sentences.
- It *is* odd to assert a conditional if there is no apparent connection between the antecedent and the consequent.

In view of these problems with the conditional, many have proposed alternative analyses, rejecting the material conditional. However, a Gricean account of when it is appropriate to *use* conditionals may help.

For example, it makes sense to assert

- If the cake was poisoned, then the dog will die.

when the dog has eaten the cake. On the other hand, if I know that the cake was not poisoned (so the antecedent here is false), it is almost certainly inappropriate to assert

- If the cake was poisoned, then the dog is a space alien from Alpha Centauri.

This would be a violation of quantity (saying less than is worthwhile for the purposes of the discussion) and a violation of relevance (talking

about irrelevant matters). In the same way, it would be a violation of quantity or manner to assert

- If the cake was poisoned, then $2 + 2 = 5$.

unless this were a dramatic and flowery way of asserting the claim that the cake wasn't poisoned. Similarly, asserting this

- If it's Tuesday, then the dog will die.

when it is merely materially true in the sense that the dog will not die (today), then this is a violation of quantity and of relevance.

IS THIS ALL THERE IS ABOUT CONDITIONALS? However, some have thought that there is something more to be said about conditionals than these considerations. Perhaps their distinctive behaviour cannot be explained solely in terms of implciatures and Grice's maxims. Consider these cases where we assert conditionals only on the basis of a false antecedent or a true consequent, and compare them with corresponding examples of disjunctions.

— *False antecedent*:
- If it's Tuesday, then the dog will die.
- Either it isn't Tuesday, or the dog will die.

In this case it seems very odd to assert the conditional, and more appropriate to assert the disjunction when you know that one disjunct is true. If I know that it isn't Tuesday, then the disjunction is indeed true, even if it's odd to assert. It seems not only odd to assert the conditional in this circumstance, but to many it sounds *false* in a way that the disjunction does not. The same holds for conditionals with true consequents. If I know that it's Monday, then

— *True consequent*:
- If the cake was poisoned, it's Monday.
- Either the cake wasn't poisoned, or it's Monday.

In this case, too, it seems more appropriate to assert the disjunction than the conditional, even though the truth conditions are the same, if the classical truth tables are to be believed. For these reasons, some think that alternative truth conditions for the conditional should be found.

IMPLICATURES AND PRAGMATICS The theory of implicature is part of a domain of study of meaning which is called 'pragmatics.' Pragmatics deals with language in *use* and the way in which context contributes to our use of words and sentences.

There are a number of other puzzles about connectives which seem to be illuminated by attention to pragmatics.

- English '*or*' often seems to have an *exclusive* rather than an *inclusive* sense ('but not both'): but some have argued that this is again the result of pragmatics.
- What is the meaning of '*but*'? It seems to share the truth conditions of '*and*':

  The sun is shining, but it is cold outside.
  The sun is shining, and it is cold outside.

  However *but* implicates that the second conjunct is unexpected.

SUMMARY AND FURTHER CONSIDERATIONS

- The propositional connectives account quite well for much of the meaning of natural language connectives.
- But, it looks like there are some aspects of the meaning of sentences containing connectives which are not explained by their truth-conditional definitions.
- A theory of *pragmatics* might help us here.
- However, this is an area of debate and on-going work.

COMPUTER SCIENCE: PROPOSITIONAL PROLOG

6

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

These are the ideas and concepts we'll learn in this chapter of the notes.

- Core task of automated reasoning: computing logical consequence.
- Restricting to a fragment of propositional logic: program clauses, logic programs and queries.
- PROLOG notation: program clauses `q :- p_1, p_2, ..., p_m.` or simply `q.`, and queries `?- r_1, r_2, …, r_k.`
- How PROLOG answers queries: the resolution proof rule and resolution refutation sequences.
- Resolution refutations in clausal form, and the Horn clause fragment of propositional logic.
- Incompleteness in PROLOG: when it can crash and not answer.
- Negation as failure in queries: PROLOG answers **true** to `?- \+ q` exactly when it answers **false** to `?- q .`

SKILLS

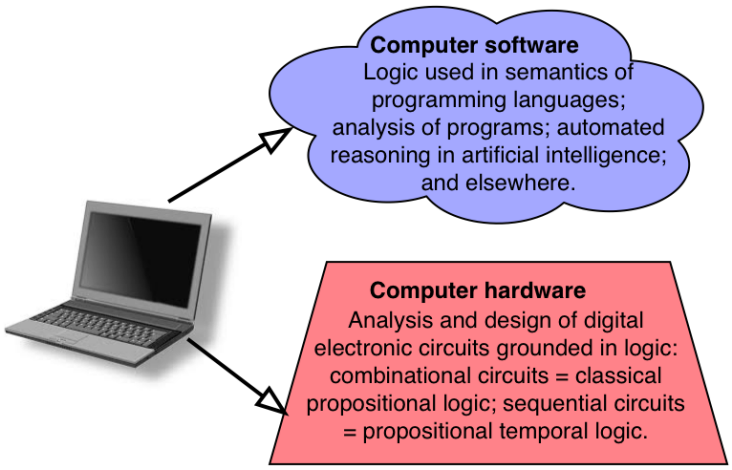
These are the skills you will learn to demonstrate.

- Determine whether or not a logic formula is logically equivalent to a logic program.
- Represent a small knowledge base of information as a logic program, and write queries to answer questions based on that information.
- Determine a resolution refutation sequence demonstrating why PROLOG answers the way it does to a given program and query.
- Identify a resolution refutation sequence translated into the Horn clause fragment of propositional logic.
- Given a PROLOG program and a sequence of queries, determine how PROLOG will respond to each of those queries.
- Use negation in a PROLOG query (carefully!).

6.1 | LOGIC, COMPUTERS, & AUTOMATED REASONING

6.1.1 | LOGIC AND COMPUTERS

Computers have profoundly changed the world over the last 50 years, so an educated person today needs to have at least a basic conceptual understanding of computers, as well as some practical knowledge of how to use them. As everyone knows, computers are based on logic; what is less well understood is the multiple layers which make up that truth.



At the level of physical stuff, the *hardware* of computers is composed of digital electronic circuits. The basic building blocks of digital circuits are AND, OR and NOT gates, which are physical, electronic realisations of the connectives of propositional logic. The Applications Section 3 on Combinational Digital Circuits gives an introduction to this core class of digital circuits and their grounding in logic.

At the level of thinking and doing stuff, the *software* of computers is the means by which we get to *use* the computing capacity of hardware. Logic is fundamental in multiple areas of computer science, the home discipline of software. Logic is central in giving the formal meaning or semantics of commands and declarations in a programming language. Indeed, the idea of a formal language with syntax generated by a grammar first emerged in logic. The analysis and design of programs, database systems, even operating systems - all of these draw on formal logic.

The area of computer science within which logic is *most* visible is that of *automated reasoning*, which falls under the broader area of artificial intelligence. In trying to program a computer to *think logically*, the core job is to compute logical consequence. The *knowledge* available to the system should not be restricted to just the propositions explicitly stored in the computer, but should ideally include all the logical consequences of the stored propositions.

This chapter on Propositional Logic Programming will provide an introduction to the enterprise of automated reasoning, and how to

compute logical consequence in a fragment of propositional logic using a programming language called PROLOG.

First some historical context: the computational aspects of reasoning have a surprisingly long history.

6.1.2 | HISTORY OF AUTOMATED REASONING

A dominant perspective within artificial intelligence and cognitive science is that human mental activity is fundamentally a *computational* process.

An early proponent of this view is the seventeen century English philosopher Thomas Hobbes. He first formulated the radical idea that human reasoning or *ratiocination* is a process of computation, involving a form of addition and subtraction of concepts and propositions.

These days, Hobbes is better known for his political philosophy, notably the view that in the absence of a *social contract* binding us together in a political community, human life in a "state of nature" would be "solitary, poor, nasty, brutish, and short"!

Thomas Hobbes
(1588-1679)

"We must not therefore think that computation, that is, ratiocination, has place only in numbers, as if man were distinguished from other living creatures by nothing but the faculty of numbering; for *magnitude, body, motion, time, degrees of quality, action, conception, proportion, speech and names* [...] are capable of addition and subtraction. Now such things as we add or subtract, that is, which we put into account, we are said to *consider*, *compute*, *reason*, or *reckon*."
The English Works of Thomas Hobbes of Malmesbury; Vol. 1: De Corpore (1655).

In Hobbes' own era, his views on the computational nature of reasoning were strongly endorsed by next in our cast of historical figures, the German mathematician and philosopher, Gottfried Leibniz.

Gottfried Wilhelm Leibniz
(1646-1716)

Leibniz proposed a *characteristica universalis*, a universal symbolic language for science, mathematics and philosophy. Leibniz's central idea was that concepts expressed in natural language are typically complex, derivative concepts, and that by assigning symbols to all primitive, irreducible concepts, one could then formally express complex concepts as *combinations* of symbols for primitive concepts. This was all 200 years before the first formal language presentations of propositional and predicate logic, and in Leibniz's time, logic was more a loose grab-bag of principles.

Leibniz's ambitous dream was that his universal language would be a means by which disputes between persons could be simply resolved by calculating or reasoning.
"This language will be the greatest instrument of reason [...] for when there are disputes among persons, we can simply say: Let us calculate, without further ado, and see who is right." *The Art of Discovery* (1685).

David Hilbert
(1862-1943)

Coming now to the beginning of the 20th century, in the midst of a great surge of progress in formal logic, the German mathematician and logician David Hilbert proposed an meta-mathematical program of research. It was aimed at resolving once and for all a crisis in the foundations of mathematics posed by paradoxes and inconsistencies that emerged in the late 19th century. The program was an attempt to show that *all* of mathematics follows from a correctly chosen finite system of axioms; and that some such axiom system can be shown to be provably consistent.

To establish the consistency of more complex parts of mathematics, one should be able to bootstrap up from simpler areas of mathematics, such as arithmetic on the natural numbers. The consistency proof itself was to be carried out using only what Hilbert called "finitary" methods. The special epistemological character of finitary reasoning then yields the required justification of classical mathematics.

Kurt Gödel
(1906-1978)
[on left, with Einstein]

Hilbert's dream came to a crashing halt only 11 years later, in 1931, at the hands of a then 25 year old fresh PhD, Kurt Gödel. Gödel proved that any formal logic system expressive enough to describe arithmetic on the natural numbers (1) is incomplete if it is consistent; and (2) the consistency of the system cannot be proved within the system itself. Recall that *incompleteness* (the negation of *completeness*) means that there exist true formulas of the language that cannot be proven using the rules of the proof system.

Key ideas from Godel's incompleteness theorem were incorporated into the work of the young English mathematician and logician, Alan Turing, who is the true *founder* of the computer age.

Alan Turing
(1912-1954)

Turing formalised concepts of computation and algorithm with his *Logical Computing Machine* model, subsequently called the *Turing machine* model. Turing's 1936 paper "On Computable Numbers" directly built on Godel's result to show that it is not possible to decide algorithmically whether or not a given Turing machine will "halt", or successfully terminate, on a given input. To prove the result, Turing developed a model of a *universal* Turing machine, which could simulate the computation of any Turing machine on any input. It is the universal Turing machine that served as a blue print for the construction of the first stored-program digital computer developed by John von Neumann in 1946.

After spending World War 2 decrypting German code for the British military, Turing went on to make foundational contributions to the field of artificial intelligence. Turing proposed a "test" of a machine's ability to exhibit intelligent behaviour indistinguishable from that of a human. The *Turing Test* is an an input-output test of a machine's capacity to play the "Imitation Game". Suppose that a machine and a person are separated from a third party, the interrogator, whose job is to ask questions of both the machine and the person in order to determine which is which. The object of the machine is to try to cause the interrogator to mistakenly conclude that the machine is the other person; the object of the other person is to try to help the interrogator to correctly identify the machine. Questions are posed by the interrogator and answered by the machine and the person in text format only.

To pass this test, the machine must be able to sustain a conversation with the interrogator for an extended period of time – five minutes was proffered by Turing – keeping in mind that the interrogator knows that one of the other two participants in the conversation is a machine.

I believe that in about fifty years' time it will be possible to programme computers, with a storage capacity of about 10^9, to make them play the imitation game so well that an average interrogator will not have more than 70 percent chance of making the right identification after five minutes of questioning. ... I believe that at the end of the century the use of words and general educated opinion will have altered so much that one will be able to speak of machines thinking without expecting to be contradicted.
A. M. Turing (1950) Computing Machinery and Intelligence. *Mind* 49: 433-460 (Mind Association, Oxford University Press).

While it is widely accepted that there is not yet artificial intelligence capable of reliably and repeatedly passing the Turing Test, it is arguable that Turing was correct in predicting that by the end of the 20th century, the phrase "machines thinking" would be commonplace. Regrettably, Turing did not get to see the world-changing consequences of his work. He died in 1954 after being prosecuted, and persecuted, for homosexuality.

6.1.3 | COMPUTING LOGICAL CONSEQUENCE

Our interest in the remainder of this chapter is on *logical* reasoning performed by a computer. As we have already noted, in trying to program a computer to "think logically", the core job is to compute logical consequence.

Logical consequence problem:
Given logic formulas $A_1, A_2, \ldots, A_n$ and B,

determine whether or not:

$$A_1, A_2, \ldots, A_n \models B$$

Recall that logical consequence is a *truth-transferring* relationship: it requires that under all ways of valuing the atomic formulas, if A_1, A_2 up to A_n are all true, then B must also be true.

One might think that in propositional logic, this cannot be *too* hard. After all, Gödel-like incompleteness results occur at the next level up, in predicate logic and its extensions. But propositional logical consequence is quite hard enough!

Direct semantic evaluation in propositional logic is very expensive: if there are k-many atomic propositions occurring within the formulas $A_1, A_2, \ldots, A_n$ and B, then there are 2^k-many truth table rows/valuations, so an explicit check of all of these is dumb and impractical. We will see an example with 64 atomic propositions in section §6-3. The number 2^{64} is more than 18 *quintillion*, where a quintillion is 10^{18} of 1,000,000,000,000,000,000. We definitely won't be doing truth tables!

Indeed, there is 1 million US dollars available if you can devise a better method for deciding propositional logical consequence. Specifically, a method that has *polynomial* rather than *exponential* growth in the number of atoms. That problem is equivalent to a fundamental problem in computer science, the **P** versus **NP** problem, about complexity classes, which now also known as the **Millennium Prize problem** [Clay Mathematics Institute: http://www.claymath.org/millenium-problems/p-vs-np-problem]. While still unsolved (and the prize unclaimed), many researchers believe the answer is NO, **P** does not equal **NP**, which means there are no polynomially-bounded methods for computing logical consequence in propositional logic.

As presented in Chapter 2 on propositional logic, the *proof tree method* does give us an alternative to truth tables as a means to establish logical consequence, because the method is both sound and complete:

$$A_1, A_2, \ldots, A_n \models B \quad \textit{if and only if} \quad A_1, A_2, \ldots, A_n \vdash_{\text{PfTr}} B$$

That is, B is a logical consequence of $A_1, A_2, \ldots, A_n$ if and only if a proof tree starting with $A_1, A_2, \ldots, A_n$ and ~B has every branch closed. While this completeness of proof trees is conceptually important, as well as practically useful for evaluating logical consequence "by hand", it is less practically useful for automated reasoning. Remember how much choice you had in applying the tree rules? How making a poor choice could lead to a blow-out in the size of the tree? It is this that makes implementation of proof trees in a computer program rather difficult – although there are implementations out there, usually a variant of proof trees called *tableau* are used instead, and these are beyond the scope of this introductory subject.

To find practically computable classes of logical consequence problem, many researchers have got down the path of shrinking the playing field with various restrictions:

- restrict to a simple fragment of the language;
- use a proof system with *only one rule*.

Instead of working with all possible formulas, we restrict instead to those in a manageable fragment. For the rest of this chapter, we will be dealing with *program clauses* and *goals*, which fall into a class known as *Horn clauses*. The second restriction comes as a follow-on from then first: by using only simple formulas of a known shape, we can get to work in a proof system with *only one rule*, to keep it simple for "dumb machines".

This approach is known as *Logic Programming*. It is a direct, declarative style of computer programming using logic-based languages such as PROLOG or DATALOG. It is declarative in the sense that you write *what* you want to compute, but not how. This is in contrast with *procedural* or *imperative* programming languages such as Java or C, where the program describes explicitly the steps of computation.

There is also a procedural reading of PROLOG program clauses, which we will discuss below, but the declarative versus procedural distinction is still useful.

6.2 | LOGIC PROGRAMMING IN PROLOG

6.2.1 | PROGRAM CLAUSES AND LOGIC PROGRAMS

We begin by formally defining the restricted fragment.

A program clause or definite clause is a formula of one of two kinds:

$$(p_1 \,\&\, p_2 \,\&\, \cdots \,\&\, p_m) \supset q \qquad \text{conditional rule}$$
$$q \qquad \text{fact}$$

where $m \geqslant 0$ and $p_1, p_2, \ldots, p_m, q$ are all atomic formulas.
(Case of $m = 0$ gives facts.)
A logic program $\mathcal{P}$ is a list $A_1, A_2, \ldots, A_n$ of program clauses.
A goal $\mathcal{G}$ is a list $r_1, r_2, \ldots, r_k$ of atomic formulas, with the goal formula the conjunction $B = (r_1 \,\&\, r_2 \,\&\, \cdots \,\&\, r_k)$.

Automated reasoning task in logic programming framework: determine whether or not:

$$\mathcal{P} \models \mathcal{G} \quad \text{that is,} \quad A_1, A_2, \ldots, A_n \models B$$

A knowledge base of information is to be represented by a list of *program clauses*, also known as *definite clauses*. Each of these is either a *fact*

– a single atomic proposition on its own, or else a simple conditional whose consequent is a single atomic proposition, and with antecedent a conjunction of atoms. If there are zero atoms in the antecedent, then the conditional reduces to a fact.

A *logic program* $\mathcal{P}$ is thought of as the *conjunction* of its list of program clauses, and we think of it as a database of information within some knowledge domain. We say a propositional logic formula A is logically equivalent to a logic program exactly when $(A \equiv C)$ is a tautology, where C is a single program clause or a conjunction of two or more program clauses. In establishing that a given formula is equivalent to a logic program, the following logical equivalences will be useful. For any logic formulas $A, A_1, A_2, A_3, B, B_1, B_2$, the following are tautologies:

Remember there is a list of useful tautologies at the end of Chapter 1.

$$\begin{aligned} (A \supset (B_1 \,\&\, B_2)) &\equiv ((A \supset B_1) \,\&\, (A \supset B_2)) \\ (A_1 \supset (A_2 \supset B)) &\equiv ((A_1 \,\&\, A_2) \supset B) \\ (\sim A_1 \vee \sim A_2 \vee \sim A_3 \vee B) &\equiv ((A_1 \,\&\, A_2 \,\&\, A_3) \supset B) \end{aligned}$$

A logic program $\mathcal{P}$ represents our knowledge base in some domain. The logic programming framework allows us to query the *logical consequences* of our knowledge base using formulas of a restricted form. A *goal* $\mathcal{G}$ is just a list of one or more atoms, and the corresponding goal formula is the *conjunction* B of those atoms (or the atom alone if there is only one in the goal).

In the logic programming framework, the automated reasoning task of interest is whether or not a goal $\mathcal{G}$ is a logical consequence of a logic program $\mathcal{P}$. Again, recall this requires a *truth-transferring* relationship: for every way of valuing the atoms, or for every row of the truth table, if a valuation makes all of the program clauses A in $\mathcal{P}$ true, then it also makes the goal formula B true, and hence each of the goal atoms in $\mathcal{G}$ is true under that valuation.

If the answer is YES to a logical consequence question $\mathcal{P} \models \mathcal{G}$? then we know that all the atoms in the goal $\mathcal{G}$ logically follow from the knowledge base $\mathcal{P}$, so we can effectively adjoin the atoms in $\mathcal{G}$ to expand our knowledge base.

First, we will get some practice identifying formulas that can be rewritten as logic programs.

Which one or more of the following propositional formulas can be re-written as a logic program: that is, *logically equivalent* to either a program clause or a conjunction of two or more program clauses. Assume $p, p_1, p_2, p_3, q, q_1, q_2$ are all atomic formulas.

(a) $p \,\&\, (q_1 \vee q_2)$

(b) $\sim p_1 \vee \sim p_2 \vee \sim p_3 \vee \sim p_4 \vee q$

(c) $(p_1 \,\&\, p_2) \supset (q_1 \,\&\, q_2)$

(d) $(p_1 \,\&\, p_2) \supset (q_1 \vee q_2)$

(e) $p_1 \supset (p_2 \supset (p_3 \supset q))$

Given a logic program $\mathcal{P}$ with program clauses $A_1, A_2, \ldots, A_n$, together with a goal $\mathcal{G}$ consisting of atoms $r_1, r_2, \ldots, r_k$ with conjunction $B = (r_1 \,\&\, r_2 \,\&\, \cdots \,\&\, r_k)$, the *refutation* approach to the logical consequence question $\mathcal{P} \models \mathcal{G}?$ is as follows:

> Suppose that each of the program clauses $A_1, A_2, \ldots, A_n$ in $\mathcal{P}$ is true, and at the same time, the *negated* goal formula $\sim B$ is true; then from this collection of data, try to derive a contradiction – to show that it is impossible.

The proof tree method in Chapter 2 also adopts the refutation approach; to test whether or not $A_1, A_2, \ldots, A_n \models B$, we start a tree with A_1, A_2 and so on up to A_n as well as $\sim B$, and then develop all branches and see whether or not they all close with contradictions.

The core automated reasoning task in the logic programming framework are the inference steps taken trying to derive a contradiction from the program clauses $A_1, A_2, \ldots, A_n$ together with the negated goal formula $\sim B$. The class of formulas over which these inference steps will apply is restricted to a fragment called the *Horn clause* fragment of classical logic. Unlike the proof tree method, we only want *one* inference rule.

As we saw in the exercise above, if A is program clause of the form $(p_1 \,\&\, p_2 \,\&\, \cdots \,\&\, p_m) \supset q$, then A is logically equivalent to a disjunction $(\sim p_1 \vee \sim p_2 \vee \cdots \vee \sim p_m \vee q)$. Also notice that a negated goal formula $\sim B$ is logically equivalent to the disjunction $(\sim r_1 \vee \sim r_2 \vee \cdots \vee \sim r_k)$. Both these sorts of formulas fall into a common class.

A logic formula A is called a *clause* if and only if A is a disjunction of *literals*, where a literal is either an atomic formula or the negation of an atomic formula; here, we allow disjunctions of size 1 consisting of a single formula.

A logic formula A is called a *Horn clause* if and only if it is logically equivalent to a clause containing *at most one* positive literal.

With this terminology, we see that:

- program clauses are Horn clauses with *exactly one* positive literal, and
- negated goal formulas are Horn clauses with *zero* positive literals, which means all literals are negative.

Later in the chapter, in **§6.4 How Prolog Answers Queries**, we will come back to the Horn clause fragment and the *resolution proof rule*.

To appreciate that Horn clauses do mark out a proper *fragment* of propositional logic, we need to locate where they sit within the class of all logic formulas.

A logic formula A is in *conjunctive normal form* (CNF) if and only if A is a conjunction of clauses; here, we allow size-1 conjunctions consisting of a single formula.

Hence logic programs are CNF formulas whose clauses contain exactly one positive literal, while negated goal formulas are CNF formulas with only one clause, all of whose literals are negative. If we take the conjunction of a logic program with a negated goal formula (as one does under the refutation approach), then the result is a CNF formula all of whose clauses are Horn clauses.

Let $p_1, p_2, p_3, p_4, q_1, q_2, q_3$ all be atomic formulas. The following are all CNF formulas; only the first two have all their clauses being Horn clauses:

$$(\sim p_1 \vee q_1)$$
$$(\sim p_1 \vee \sim p_2 \vee \sim p_3 \vee q_1) \;\&\; q_2 \;\&\; (\sim q_1 \vee \sim q_2 \vee \sim q_3)$$
$$(p_1 \vee p_2 \vee \sim q_1) \;\&\; (q_2 \vee q_3)$$

The third CNF formula has clauses with two positive literals.

Fact: For every propositional logic formula A, there exists at least one (but usually many) formulas C such that C is in CNF and A is logically equivalent to C (that is, $A \equiv C$ is a tautology).

The notion dual to CNF is *DNF*, *disjunctive normal form*; a formula A is in DNF if and only if A is a disjunction of conjunctions of literals, also allowing size-1 disjunctions and conjunctions. DNF formulas play a key role in Digital Systems (see §3.3.3 above).

We briefly justify the claim that every propositional formula is logically equivalent to some CNF formula. One way to show this is to proceed via a DNFs and use De Morgan equivalences. Start with an arbitrary formula A and suppose its atoms are p_1, p_2, ..., p_n, so its truth table has 2^n-many rows. Each row can be uniquely characterised by a size-n *conjunction* such that, for each $i \in \{1, 2, \ldots, n\}$, the row-conjunction includes the negative literal $\sim p_i$ if p_i has value 0 on that row, and this row-conjunction includes the positive literal p_i if p_i has value 1 on that row. Let D be the disjunction of all the row-conjunctions for rows of the truth table in which A evaluates to 0. Then D is a DNF, and is the *canonical DNF* for the formula $\sim$A. Hence $\sim$D is logically equivalent to the original formula A. To obtain a CNF formula C logically equivalent to A, we can apply a sequence of logical equivalences to $\sim$D, starting with the De Morgan equivalences, to drive the negations inwards to the atoms, and as needed, apply double-negation elimination: $(\sim\sim A \equiv A)$. For example, suppose D is the DNF formula $(p_1 \,\&\, \sim p_2 \,\&\, p_3) \vee (\sim p_1 \,\&\, p_2 \,\&\, p3)$. Then:

$$\begin{aligned} A &\equiv \sim D \\ &\equiv \sim((p_1 \,\&\, \sim p_2 \,\&\, p_3) \vee (\sim p_1 \,\&\, p_2 \,\&\, p_3)) \\ &\equiv (\sim p_1 \vee p_2 \vee \sim p_3) \,\&\, (p_1 \vee \sim p_2 \vee \sim p_3)\,. \end{aligned}$$

With the restriction to clauses containing at most one positive literal, we can see that conjunctions of Horn clauses constitute a fragment of the language of propositional logic. To show that conjunctions of Horn clauses constitute a proper fragment, we only need exhibit a propositional formula that is not logically equivalent to a conjunction of Horn clauses; this formula will do:

$$(p_1 \vee p_2 \vee \sim q_1) \,\&\, (q_2 \vee q_3)\,.$$

6.2.2 | PROGRAM CLAUSES & GOALS IN PROLOG

Having identified the restricted fragment of propositional logic, we now turn to the PROLOG language – and its syntactic idiosyncrasies.

Syntax: A program clause or definite clause is written in propositional PROLOG as:

```
q :- p_1, p_2, ..., p_m.
```

or

```
q.
```

where $m \geqslant 0$ and `p_1, p_2, ..., p_m, q` are all atoms; the case of $m = 0$ gives facts `q.`

The PROLOG language has a number of idiosyncratic features.

- Conditionals are written in *reverse order* compared with propositional formulas. The connective symbol "`:-`" is read as "IF"; the connective is *preceded by* the consequent `q` (called the 'head' of the clause) and *followed by* the list of atoms in the antecedent (the 'body' of the clause). The connective "`:-`" is also called the 'neck', connecting head with body.
- Commas between the antecedent atoms `p_i`'s are read as "AND".
- Simple facts `q.` have no body, just a head.
- Atoms must have names starting with a *lower-case letter*, then followed by upper-case letters, lower-case letters, digits or an underscore _.
- Each program clause must end with a full-stop or period "." .
- A PROLOG program consists of a list of program clauses written in plain text file, with file type ".pl"; that is, a name of form "`progname.pl`".
- The *order* of atoms within a clause, and the order of program clauses in a program, both make a difference in how PROLOG runs (see §6.4 below).

Program clauses in PROLOG are written *declaratively*, with the meaning: `q` is true if all of `p_1, p_2,` up to `p_m` are true, or simply `q` is true in the case of a fact. However, there is also a *procedural* interpretation: we can establish `q` if we first establish `p_1`, and then establish `p_2`, and continuing up to `p_m`, in that order.

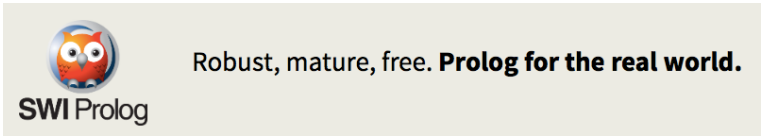
After loading a logic program into a PROLOG interpreter, posing a query with a goal is the means to ask whether or not the goal is a logical consequence of the given logic program.

Syntax: A goal $\mathcal{G}$ is written in propositional PROLOG as:

```
?-  r_1, r_2, ..., r_k.
```

where $k \geqslant 1$ and `r_1, r_2, ..., r_k` are all atoms; that is, a list of atoms after the query prompt "`?-` " ending with a full-stop.

There are numerous implementations of PROLOG around. We use:



http://www.swi-prolog.org

This is a free, open-source PROLOG implementation for MS-Windows, Mac OSX, and Linux. The latest stable release is version 6.6.2: http://www.swi-prolog.org/download/stable (accessed of 9 March 2014).

Nothing depends on the choice of PROLOG implementation, so if you have a reason to prefer an alternative, it's not a problem. Another free, open-source PROLOG interpreter is:

GNU PROLOG

http://www.gprolog.org/

6.2.3 | EXAMPLE PROLOG PROGRAM AND QUERY

Time to get straight into an example. Consider the following bit of weather data expressed in a PROLOG program. It is rather salient to us here as the Australian summers – with high temperatures and high fire danger – have become increasingly challenging over the last few years, evidencing global warming. We are located in Melbourne, the capital of the state of Victoria, and in the central part of the state. There is a little town with the great name of 'Yackandandah', in Northern Victoria, where it is usually hotter than Melbourne. With that preamble, consider the following PROLOG program.

```
/* weather-001.pl */
windy :- melbourne.
windy :- yackandandah.
light_rain :- melbourne.
dry :- yackandandah.
hot35 :- northern_victoria.
mild25 :- central_victoria.
central_victoria :- melbourne.
northern_victoria :- yackandandah.
high_fire_danger :- windy, dry, hot35.
yackandandah.
/* comments marked like this */
```

The PROLOG program `weather-001.pl` declares that it will be windy if you are in Melbourne, and also windy if you are Yackandandah. Light rain if in Melbourne while dry if in Yackandandah. Continuing with broader-level weather forecasts, it will be hot with temperature at least 35 degrees celsius if you are in Northern Victoria, while mild with 25 celsius if in Central Victoria (this is a wish!). The next two clauses record the geographical data that you are in central Victoria if you are in Melbourne, and in Northern Victoria if you are in Yackandandah.

The last conditional says there is high fire-danger if it is windy, dry and hot. The final fact expressed in the program is that you, the user, is in fact in Yackandandah.

So, load the program `weather-001.pl` into PROLOG and at the query prompt `?-`, put forth the goal `high_fire_danger` followed by a full-stop. We press the "**return**" key, and what happens?

```
% /Users/jdavoren/.plrc compiled 0.00 sec, 1 clauses
Welcome to SWI-Prolog (Multi-threaded, 64 bits, Version 6.6.1-DIRTY)
Copyright (c) 1990-2013 University of Amsterdam, VU Amsterdam SWI-Prolog
comes with ABSOLUTELY NO WARRANTY. This is free software, and you
are welcome to redistribute it under certain conditions.
Please visit http://www.swi-Prolog.org for details.
For help, use ?- help(Topic). or ?- apropos(Word).
% /Users/jdavoren/Prolog/Prolog-code/weather-001.pl compiled
0.00 sec, 11 clauses
```

```
1 ?- high_fire_danger.
ERROR: windy/0: Undefined procedure: melbourne/0
Exception: (8) melbourne ?
```

Alas, PROLOG complains. It cannot find any program clause with head `melbourne`, so that atom is declared "undefined". It first declares ERROR at line 2, containing the clause: `windy :- melbourne.`, as this occurrence of the atom `windy` as a clause head leads to atom `melbourne` in the body. Then an "exception" is raised at line 8, containing the clause: `central_victoria :- melbourne.` as that is the last occurrence of the atom `melbourne` in the program, and no program clause with head `melbourne` has been found. PROLOG implementations usually insist that every atom occurring in a program must occur at least once as the head of some program clause, with the error messages given differing between implementations. A way round this is to add an extra clause that is a tautology: that is always true. Let the keyword `false` denote the constant proposition that is always valued false. Then for any atom p, the conditional (`false` $\supset$ p) is always true, because it is a conditional with a false antecedent. In particular, the PROLOG program clause:

```
melbourne :- false.
```

is always true, so we can safely add it to our PROLOG program without changing the meaning of the program.

```
/* weather-001.pl revised */
windy :- melbourne.
windy :- yackandandah.
light_rain :- melbourne.
dry :- yackandandah.
hot35 :- northern_victoria.
mild25 :- central_victoria.
central_victoria :- melbourne.
northern_victoria :- yackandandah.
high_fire_danger :- windy, dry, hot35.
yackandandah.
melbourne :- false.

1 ?- high_fire_danger.
true.
2 ?- windy, dry, hot35.
true.
3 ?-
```

With our revised program, PROLOG responds to both the queries:

`?- high_fire_danger.` and `?- windy, dry, hot35.`

with the answers `true`. This is exactly as is expected, given our understanding of the program – and our as yet naive understanding of how PROLOG works. The program `weather-001.pl` declares that the atom `high_fire_danger` is true if `windy`, `dry` and `hot35` are all true. The program also says that the user is in fact in Yackandandah, that it is windy and dry in Yackandandah, and that `hot35` is true in Northern Victoria, which includes Yackandandah. Hence `high_fire_danger` is a logical consequence of the program `weather-001.pl`.

6.2.4 | LOGICAL CONSEQUENCE ANSWERS FROM PROLOG

Let $\mathcal{P}$ be a logic program stored in file `progname.pl` and let $\mathcal{G}$ be a goal:

```
?- r_1, r_2, ..., r_k.
```

where each of the atoms `r_1, r_2, ..., r_k` occur as the head of at least one program clause within $\mathcal{P}$.

To get a PROLOG interpreter to answer the logical consequence question of whether or not $\mathcal{P} \models \mathcal{G}$?,

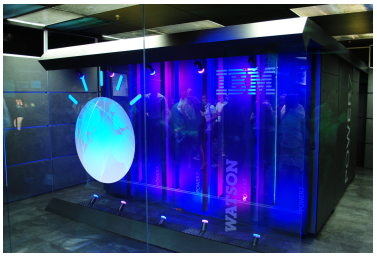
1. load the program file `progname.pl` containing $\mathcal{P}$ into your PROLOG interpreter: can use File→Consult... or `?- [progname].`
2. enter the goal atoms in $\mathcal{G}$ in a list at the query prompt `?-`
3. end list with full-stop `.` and then press "**return**" key
4. the PROLOG interpreter then answers with either:
   - `true` or `yes`, meaning $\mathcal{G}$ is a logical consequence of $\mathcal{P}$, or
   - `false` or `no`, meaning $\mathcal{G}$ is *not* a logical consequence of $\mathcal{P}$, or else

- PROLOG gets stuck in a loop and crashes, or reports some other error.

In section §6.4 below, we will discuss in detail these different possible responses of a PROLOG interpreter, and the inference steps used in arriving at its response.

6.2.5 | PROLOG USED IN CURRENT AI SYSTEMS

Perhaps you are worried that program clauses and goals are too restrictive and too small a fragment of logic to be of much practical use. In fact, PROLOG is widely used in artificial intelligence (AI) applications; in addition to automated reasoning, it is used in natural language processing, expert systems, intelligent database retrieval, machine learning and robot planning.



IBM's AI system "Watson"

Admittedly, it is the full Predicate logic version of PROLOG that is needed for substantial AI applications, while in this introduction, we are only dealing with the *propositional* fragment of PROLOG. [You may want to stay around for the successor course, *Logic: Language & Information 2*, where there is an applications topic on PROLOG for predicate logic.]

In a recent public success, IBM's artificial intelligence computer "Watson" uses PROLOG for logical intelligence – with access to a knowledge base of 200 million pages of structured and unstructured content. The system was first developed to compete on U.S. television quiz show *Jeopardy*. In 2011, Watson competed on *Jeopardy* against two former champions, and beat them both to win $1 million in prize money. In 2013, IBM announced that Watson's first commercial application would be for utilization management decisions in cancer treatment. [See `http://en.wikipedia.org/wiki/Watson_(computer)` (accessed 9 March 2014)]

6.3 | PROLOG PROJECT: SUDOKU PUZZLES

Our project is to solve 4×4 "toy-sized" Sudoku puzzles, and it is offered as an exercise in applying PROLOG to a concrete but easy-enough problem domain. Most people will have seen the usual 9×9 Sudoku puzzles at some point in newspapers or magazines: they have been quite popular for ten years or so. Sudoku puzzles are solved by logic alone, so that makes them a good choice for automated reasoning. We are going to work with the "toy-sized" 4×4 puzzles because they are extremely easy to solve, so the logic is quite transparent, yet they contain much of the same structure as the regular sized puzzles. As we will see, even little 4×4 Sudokus end up generating quite enough work, and require a large number of propositional program clauses.

The purpose of the project is to provide extended practice with PROLOG. In addition, it will serve to illustrate two key themes within the discipline of computer science.

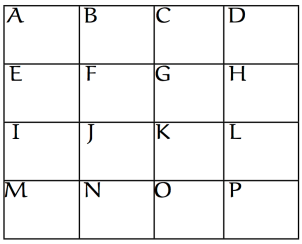
The first is *problem representation*. In getting a computer to solve a problem, the first crucial task is to analyze and describe the problem and its features. One then has to work out how to best *represent* the problem and its features in a way that fits with the software tools available: here, logic programming in propositional PROLOG.

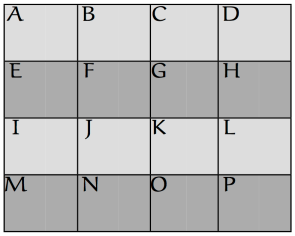
The second theme we will illustrate with this small project is *algorithmic thinking*. An *algorithm* is a set of instructions for solving a computational problem, like a recipe is a set of instructions for cooking a dish. But unlike some recipes, there should be no vagueness and imprecision in the instructions. With this example, we will see an instance of a precise but *non-deterministic* algorithm (with more than one output for each input) and we will also briefly discuss the *correctness* and efficiency of our algorithm.

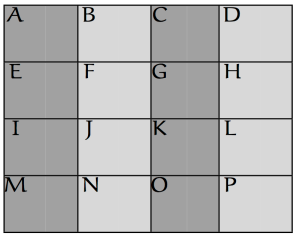
If you are eager to see the details of how and why a PROLOG interpreter gives the responses it does to queries, you may want to skip ahead to section §6.4 and come back to this project later.

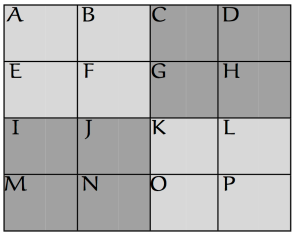
6.3.1 | SUDOKU PUZZLES: PROBLEM REPRESENTATION

Label the 16 cells of the 4×4 grid with capital letters A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P, as shown below. The 16 cells are naturally arranged in 4 rows, in 4 columns, and in 4 blocks, as shown:









The only rule governing Sudoku puzzles is the following:

Sudoku rule: A correctly completed 4×4 Sudoku must contain a permutation of $\{1, 2, 3, 4\}$ in the cells of each row, each column, and each block.

We assume each 4×4 Sudoku puzzle has an initial assignment of values $\{1, 2, 3, 4\}$ to four *pairwise-independent* cells in the grid – and we'll make precise what we mean by "pairwise-independent" a little later.

For example, suppose we have a 4×4 Sudoku puzzle with initial assignment as shown below. (The set of cells $\{B, G, L, M\}$ counts as pairwise-independent.)

A	B 1	C	D
E	F	G 2	H
I	J	K	L 4
M 3	N	O	P

Sudoku example 001

At this first stage, we can readily see how to start completing the puzzle: first identify the empty cells that are immediately constrained by the initial values. Cell D must have value 3 because there is a 1 in the same row (cell B), a 2 in the same block (cell G) and a 4 in the same column (cell L). Likewise, cell E must have value 4 because there is a 1 in the same block (cell B), a 2 in the same row (cell G), and a 3 in the same column (cell M). Proceeding likewise, cell J must have value 2 and cell O must have value 1.

A	B 1	C	D 3
E 4	F	G 2	H
I	J 2	K	L 4
M 3	N	O 1	P

A 2	B 1	C 4	D 3
E 4	F 3	G 2	H 1
I 1	J 2	K 3	L 4
M 3	N 4	O 1	P 2

At the second stage of solving this puzzle, we use the new facts established at stage one together with the initial values to identify the constraints on the remaining empty cells. Thus cell A must have value 2 because there is a 1 in the same block (cell B), a 4 in the same column (cell E) and a 3 in the same row (cell D).

Sudoku project: problem representation
Our goals are:

- represent propositions like "cell B has value 1" as atoms;
- represent as much as we can of the structure of the problem in program clauses in a propositional PROLOG program;
- solve these puzzles using PROLOG queries.

To begin with, we need to clearly identify the objects of interest. The *values* are numbers in the set $\{1, 2, 3, 4\}$. Let Σ be the set of *cell labels*:

$$\Sigma := \{A, B, C, D, E, F, G, H, I, J, K, L, M, N, O, P\}.$$

The most basic information is the value in each cell, and this varies with each different puzzle instance. For each of the 16 cells X in Σ, we will have the following 4 atoms, to give a total of 64:

v_X_1 := Cell X has value 1.
v_X_2 := Cell X has value 2.
v_X_3 := Cell X has value 3.
v_X_4 := Cell X has value 4.

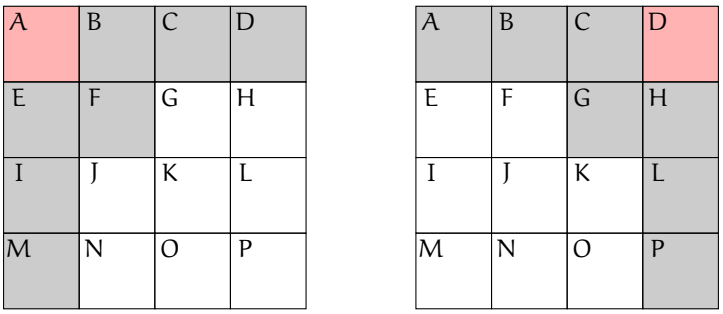
For our first example, we have the following four initial facts:

```
/* Sudoku example 001 */
v_B_1.
v_G_2.
v_L_4.
v_M_3.
```

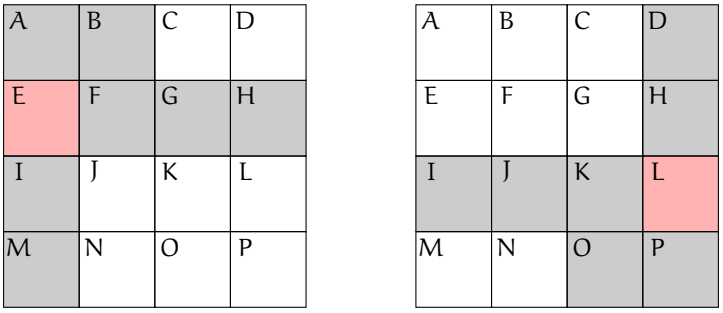
For each of the cells X in Σ, and for any cell Y in Σ, we will say Y is *Sudoku-related* to X if and only if Y is different from X and either Y is in the same *row* as X, or Y is in the same *column* as X, or Y in the same *block* as X.
Hence the Sudoku relationship is symmetric: if Y is Sudoku-related to X then X is Sudoku-related to Y.
Each of the 16 cells has 7 other cells that are Sudoku-related to it.

We will say two cells X, Y in Σ with X ≠ Y are *independent* if and only if X is not Sudoku-related to Y.

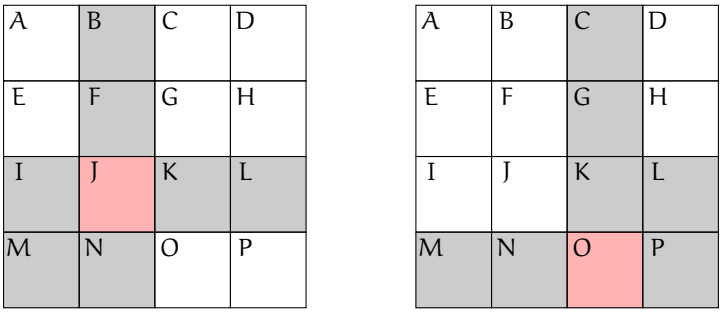
A set of cells is *pairwise-independent* if and only if any two different cells in the set are independent of each other. (In our first example, the set of cells {B, G, L, M} is pairwise-independent.)



Cell A is Sudoku-related to cells B, C, D, E, F, I, M, and cell D is Sudoku-related to cells A, B, C, G, H, L, P.



Cell E is Sudoku-related to cells A, B, F, G, H, I, M, and cell L is Sudoku-related to cells D, H, I, J, K, O, P.



Cell J is Sudoku-related to cells B, F, I, K, L, M, N, and cell O is Sudoku-related to cells C, G, K, L, M, N, P.

Visually, it is easy for us to *see* all the cells Sudoku-related to a given cell; we only scan these when solving the value of a given cell. For access by a computer, one way we could store the Sudoku relation data is as a list of lists, as follows.

cell X in Σ	cells Y Sudoku-related to X
A	B, C, D, E, F, I, M
B	A, C, D, E, F, J, N
C	A, B, D, G, H, K, O
D	A, B, C, G, H, L, P
E	A, B, F, G, H, I, M
F	A, B, E, G, H, J, N
G	C, D, E, F, H, K, O
H	C, D, E, F, G, L, P
I	A, E, J, K, L, M, N
J	B, F, I, K, L, M, N
K	C, G, I, J, L, O, P
L	D, H, I, J, K, O, P
M	A, E, I, J, N, O, P
N	B, F, I, J, M, O, P
O	C, G, K, L, M, N, P
P	D, H, K, L, M, N, O

Alternatively, we can set out the Sudoku relation as a table, with entries indexed by rows and columns labelled with cell names, as shown below. At the intersection of row X and column Y, there is an entry 1 when cells X and Y are Sudoku-related, and an entry 0 when they are not. The symmetry of the Sudoku relation is visible in the table, which is symmetrically mirrored across the diagonal of all 0s; the diagonal is all 0s because the Sudoku relationship only holds between cells that are different, so no cell is Sudoku-related to itself.

Sudoku	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
A	0	1	1	1	1	1	0	0	1	0	0	0	1	0	0	0
B	1	0	1	1	1	1	0	0	0	1	0	0	0	1	0	0
C	1	1	0	1	0	0	1	1	0	0	1	0	0	0	1	0
D	1	1	1	0	0	0	1	1	0	0	0	1	0	0	0	1
E	1	1	0	0	0	1	1	1	1	0	0	0	1	0	0	0
F	1	1	0	0	1	0	1	1	1	0	0	0	0	1	0	0
G	0	0	1	1	1	1	0	1	0	0	1	0	0	0	1	0
H	0	0	1	1	1	1	1	0	0	0	0	1	0	0	0	1
I	1	0	0	0	1	0	0	0	0	1	1	1	1	1	0	0
J	0	1	0	0	0	1	0	0	1	0	1	1	1	1	0	0
K	0	0	1	0	0	0	1	0	1	1	0	1	0	0	1	1
L	0	0	0	1	0	0	0	1	1	1	1	0	0	0	1	1
M	1	0	0	0	1	0	0	0	1	1	0	0	0	1	1	1
N	0	1	0	0	0	1	0	0	1	1	0	0	1	0	1	1
O	0	0	1	0	0	0	1	0	0	0	1	1	1	1	0	1
P	0	0	0	1	0	0	0	1	0	0	1	1	1	1	1	0

Working as we are in propositional logic, one strategy would be to go ahead and add a further 112 (which is 16×7) more atomic propositions, say of the form `s_X_Y`, expressing that cell X is Sudoku-related to cell Y. We would

then use program clauses such a:

```
v_D_3  :-  s_B_D, v_B_1, s_D_G, v_G_2, s_D_L, v_L_4.
s_B_D.
s_D_G.
s_D_L.
```

which express the conditional rule that cell D has value 3 if cell B is Sudoku-related to D, cell B has value 1, cell G is Sudoku-related to D, cell G has value 2, cell L is Sudoku-related to D, and cell L has value 4, together with the facts that B, G and L are each Sudoku-related to D.

At this point, it is important to differentiate between the two kinds of information within this problem domain:

- cell relationship data that is the same for all Sudoku puzzles; and
- cell value data that varies with each different Sudoku puzzle.

Given our goal of solving any particular Sudoku puzzle using propositional Prolog, and in the interests of keeping task size manageable, we make a design decision: we will just stick with the 64 cell value atoms of the form `v_X_i`, and *not* expand the language to include an additional 112 atoms of the form `s_X_Y`. As a result, we will be representing the cell relationship data *implicitly*, via our selection of program clauses about cell values, rather than *explicitly*, via direct coding into program clauses that include atoms expressing relationship facts. For example, the program clause:

```
v_D_3  :-  v_B_1, v_G_2, v_L_4.
```

would be selected precisely because cell D is Sudoku-related to each of cells B, G and L, and the values in those cells, 1, 2 and 4, respectively, are all different from each other, and all different from 3, the value for cell D in the head atom.

In the successor course, *Logic: Language & Information 2* (LLI2), we extend beyond propositional to *predicate logic*, within which it is much easier to express this sort of relationship data as atomic formulas of the language. In predicate logic, tables like the one above (describing the Sudoku relation between cells) are key ingredients in giving the semantics of the language. In the Logic programming application section of LLI2, we revisit this Sudoku project and see how much easier it becomes in a richer language and logic.

6.3.2 | SUDOKU PROJECT REFINED

Having pragmatically chosen to directly encode only the cell value data in atoms `v_X_i`, and to not directly express cell relationship data in atom propositions, we now refine the project goal.

Sudoku project: revised goal
Design an algorithm which:

- takes as *input* a 4×4 Sudoku puzzle, with initial state four pairwise-independent cells assigned a unique value in $\{1, 2, 3, 4\}$, and
- produces as *output* a propositional Prolog program which will solve

just that puzzle, using program clauses containing only the 64 cell value atoms of the form `v_X_i`.

By assuming we start with a Sudoku puzzle with initial state being four pairwise-independent cells assigned a unique value in $\{1,2,3,4\}$, we have a guarantee that the puzzle will have a *unique solution*.

In describing our algorithm, we illustrate with our familiar example:

A	B 1	C	D
E	F	G 2	H
I	J	K	L 4
M 3	N	O	P

Sudoku example 001

```
v_B_1.  v_G_2.  v_M_3.  v_L_4.
```

STAGE 0: Let $\Theta_1 = \{X_1, X_2, X_3, X_4\}$ be the pairwise-independent set of cells from Σ that are initially valued in the given puzzle, and suppose cell X_1 has value 1, cell X_2 has value 2, cell X_3 has value 3 and X_4 has value 4. Then write the four PROLOG facts:

```
v_X1_1.  v_X2_2.  v_X3_3.  v_X4_4.
```

STAGE 1: Let $\Sigma_1 := \Sigma - \Theta_1$ be the set of cells yet to be valued; that is, cells that are in Σ but not in Θ_1. For each cell X in Σ_1, we search for three cells Y_1, Y_2 and Y_3 in Θ_1 with three distinct values j_1, j_2, j_3 (respectively), with all three of the cells Y_1, Y_2 and Y_3 Sudoku-related to X; if we can find such, then we write the PROLOG program clause:

```
v_X_i :- v_Y1_j1, v_Y2_j2, v_Y3_j3.
```

where $i \in \{1,2,3,4\}$ is the value not included among $\{j_1, j_2, j_3\}$.

Because Θ_1 is a set of 4 pairwise-independent cells, there will always be exactly 4 cells X in Σ_1 with the property that there exist 3 cells in Θ_1 with 3 distinct values, with the 3 cells all Sudoku-related to X.

For Sudoku example 001, the initially valued cells $\Theta_1 = \{B, G, L, M\}$ and the yet-to-be-valued cells $\Sigma_1 = \{A, C, D, E, F, H, I, J, K, N, O, P\}$. At STAGE 1, four program clauses are generated by the algorithm (for cells D, E, J and O) and these are unique up to re-ordering of atoms within clause bodies.

```
/* sudoku-example-001.pl */
   v_B_1.
   v_G_2.
   v_L_4.
   v_M_3.
   v_D_3   :-  v_B_1, v_G_2, v_L_4.
   v_E_4   :-  v_B_1, v_G_2, v_M_3.
   v_J_2   :-  v_B_1, v_L_4, v_M_3.
   v_O_1   :-  v_G_2, v_L_4, v_M_3.
```

A	B 1	C	D 3
E 4	F	G 2	H
I	J 2	K	L 4
M 3	N	O 1	P

STAGE 2: Let Θ_2 be the set of cells whose value is determinable after STAGE 1 (cell is either in Θ_1 or in the head of a STAGE 1 program clause), and let $\Sigma_2 := \Sigma - \Theta_2$ be the set of cells whose value is not yet determinable. For each cell X in Σ_2, we search for three cells Y_1, Y_2 and Y_3 in Θ_2 with three distinct values j_1, j_2, j_3 (respectively), with all three of the cells Y_1, Y_2 and Y_3 Sudoku-related to X; then we write the PROLOG program clause:

```
v_X_i :- v_Y1_j1, v_Y2_j2, v_Y3_j3.
```

where $i \in \{1,2,3,4\}$ is the value not included among $\{j_1, j_2, j_3\}$.

For Sudoku example 001, the set of already-valued cells after STAGE 1 is the set $\Theta_2 = \{B, D, E, G, J, L, M, O\}$ while $\Sigma_2 = \{A, C, F, H, I, K, N, P\}$. At STAGE 2, we generate the following eight program clauses:

```
/* sudoku-example-001.pl continued */
   v_A_2   :-  v_B_1, v_D_3, v_E_4.
   v_C_4   :-  v_B_1, v_D_3, v_G_2.
   v_F_3   :-  v_B_1, v_E_4, v_G_2.
   v_H_1   :-  v_D_3, v_E_4, v_G_2.
   v_I_1   :-  v_E_4, v_J_2, v_M_3.
   v_K_3   :-  v_J_2, v_L_4, v_O_1.
   v_N_4   :-  v_J_2, v_M_3, v_O_1.
   v_P_2   :-  v_L_4, v_M_3, v_O_1.
```

The algorithm asks, for each un-valued cell X in Σ_2, to search for *one* triple of valued cells Y_1, Y_2 and Y_3 in Θ_2 with three distinct values j_1, j_2, j_3 (respectively), with all three of the cells Y_1, Y_2 and Y_3 Sudoku-related to X. In STAGE 2, there will always be more than one such triple; in fact, there will always be

two such triples drawn from the 8 cells in Θ_2. For example, for cells A and C, we could have alternatively had the program clauses:

```
v_A_2  :-  v_B_1, v_E_4, v_M_3.
v_C_4  :-  v_D_3, v_G_2, v_O_1.
```

CLEANUP STAGE: so that the final PROLOG program can be queried for *all possible* values in each cell, and we avoid the `Undefined procedure` error message, we add tautology program clauses:

```
v_X_j :- false.
```

for each of the 16 cells X in Σ and each of the three values of $j \in \{1, 2, 3, 4\}$ that have not yet occurred either as an initial fact or in the head of a program clause generated at STAGE 1 or STAGE 2.

```
/* sudoku-example-001.pl continued */
  v_A_1 :- false.  v_A_3 :- false.  v_A_4 :- false.
  v_B_2 :- false.  v_B_3 :- false.  v_B_4 :- false.
  v_C_1 :- false.  v_C_2 :- false.  v_C_3 :- false.
  v_D_1 :- false.  v_D_2 :- false.  v_D_4 :- false.
  v_E_1 :- false.  v_E_2 :- false.  v_E_3 :- false.
  v_F_1 :- false.  v_F_2 :- false.  v_F_4 :- false.
  v_G_1 :- false.  v_G_3 :- false.  v_G_4 :- false.
  v_H_2 :- false.  v_H_3 :- false.  v_H_4 :- false.
  v_I_2 :- false.  v_I_3 :- false.  v_I_4 :- false.
  v_J_1 :- false.  v_J_3 :- false.  v_J_4 :- false.
  v_K_1 :- false.  v_K_2 :- false.  v_K_4 :- false.
  v_L_1 :- false.  v_L_2 :- false.  v_L_3 :- false.
  v_M_1 :- false.  v_M_2 :- false.  v_M_4 :- false.
  v_N_1 :- false.  v_N_2 :- false.  v_N_3 :- false.
  v_O_2 :- false.  v_O_3 :- false.  v_O_4 :- false.
  v_P_1 :- false.  v_P_3 :- false.  v_P_4 :- false.
/* end of file: sudoku-example-001.pl */
```

The resulting output PROLOG program can be queried for all possible values in each cell, and (we claim – see more below) the answers given will always be *correct*. For example, here are some PROLOG queries after loading the file `sudoku-example-001.pl` into SWI PROLOG:

```
% /Users/jdavoren/Prolog/Prolog-code/sudoku-example-001.pl
compiled 0.00 sec, 65 clauses
1 ?- v_D_3, v_E_2.
false.
2 ?- v_D_3, v_F_3, v_K_3, v_O_1, v_H_1, v_I_1.
true.
3 ?-
```

[SWI PROLOG gives a count of 65 clauses when there is in fact 64: there are 4 initial facts; 4 new clauses in STAGE 1, 8 clauses STAGE 2, and a final 48 tautology clauses. SWI PROLOG seems to add 1 for the clause from /Users/.plrc.]

At the end of Chapter 6, we give a complete listing of the PROLOG code `sudoku-example-001.pl` together with two more sample Sudoku puzzles and PROLOG code for each of them generated by our algorithm, with code files called `sudoku-example-002.pl` and `sudoku-example-003.pl`. You are encouraged to experiment with them in PROLOG, and try out various queries. Having experiment with our examples, you might then feel inspired to write your own PROLOG programs and queries.

Discussion of the algorithm:

Notice first that the algorithm can produce two or more different PROLOG programs that solve the same Sudoku puzzle given as input – even though the puzzle itself has a unique (exactly one) solution. Technically, an algorithm with this property is known as *non-deterministic*, in contrast with a *deterministic* algorithm which always produce as output a unique result for each input. The non-determinism is located in STAGE 2, where there will always be two different program clauses with the same head that can be generated from a triple of valued cells drawn from the 8 in Θ_2, and this applies to each of the eight program clauses added this stage. This non-determinism reflects the fact that there are multiple different inference paths that can be taken in solving any given Sudoku puzzle.

To verify that the algorithm is *correct*, we need to prove the following:

> Given as input a 4×4 Sudoku puzzle, with initial state four pairwise-independent cells assigned a unique value in $\{1,2,3,4\}$,
>
> if $\mathcal{P}$ is a propositional PROLOG program generated by our algorithm from that input, and $\mathcal{G}$ is any goal containing any of the 64 cell value atoms of the form `v_X_i` for $\mathtt{X}$ in Σ and $\mathtt{i}$ in $\{1,2,3,4\}$,
>
> then PROLOG's answer to the query $\mathcal{P} \models \mathcal{G}?$ will be correct for the given Sudoku puzzle.

We won't give a proof of correctness here as it would require an excursion into *combinatorics*, an area within discrete mathematics, and is beyond the scope of this short introduction. The first key combinatorial result requiring proof is that our restriction to 4×4 Sudoku puzzles with initial state being four pairwise-independent cells assigned a unique value in $\{1,2,3,4\}$, does indeed guarantee that the puzzle will have a *unique solution*. Each input puzzle having a unique solution in turn guarantees that every PROLOG query about cell values for a given puzzle does indeed have a *correct* answer. The other key result requiring proof is that, if $\mathcal{P}$ is any PROLOG program generated by our algorithm – no matter how choices are resolved in STAGE 2 – then the clauses in $\mathcal{P}$ exactly match the unique completed Sudoku, in the sense that for each cell $\mathtt{X}$ with value $\mathtt{j}$ in the completed Sudoku, there is a clause in $\mathcal{P}$ which either is a fact `v_X_i.` or else a has the atom `v_X_i` as its head and all the atoms in the body of that clause occur earlier in the program $\mathcal{P}$ as either a fact or a clause head. In proving that the clauses in $\mathcal{P}$ exactly match the unique completed Sudoku, for the STAGE 1 clauses one would need to appeal to a combinatorial fact about 4×4 grids that if you take any three pairwise independent cells, then there is exactly one other cell $\mathtt{X}$ such that $\mathtt{X}$ is Sudoku-related to all three.

Our algorithm could be *implemented* using any of the common programming languages (*Python* is a good candidate). Any implementation would have to resolve the non-determinism in STAGE 2 by making a choice: for example,

for each of the cells X in Σ_2, you could choose the *alphabetically first* triple of cells Y_1, Y_2 and Y_3 in Θ_2 having three distinct values j_1, j_2, j_3 (respectively), with all three of the cells Y_1, Y_2 and Y_3 Sudoku-related to X. This is the choice principle used in generating "by hand" the PROLOG programs for the examples in this text.

One can reasonably ask whether our approach of generating PROLOG code one puzzle at a time is the best option. Why don't we just write one giant PROLOG program that contains *all possible* program clauses of the form:

```
v_X_i :- v_Y1_j1, v_Y2_j2, v_Y3_j3.
```

where X, Y_1, Y_2, Y_3 are cells in Σ with each of Y_1, Y_2 and Y_3 Sudoku-related to X, and

The answer is that such a PROLOG program would be a giant *monster*, containing around half a million program clauses! (a total of 483 840 by my calculations)

Using PROLOG for Predicate logic (as we do in *Logic: Language & Information 2*) we will have a *much* easier time as we can directly code up the relationship "cell X is Sudoku-related to cell Y" and "$i \neq j$".

6.3.3 | A NEW EXAMPLE SUDOKU PUZZLE AND PROGRAM

We finish off this project section by running the algorithm again on a new example, starting with the initial facts:

```
v_A_1.  v_G_2.  v_J_3.  v_P_4.
```

A 1	B	C	D
E	F	G 2	H
I	J 3	K	L
M	N	O	P 4

Sudoku example 002

The file `sudoku-example-002.pl` (listed at the end of Chapter 6) is a PROLOG program generated according to the algorithm, with input the puzzle example 002. In STAGE 2, the choice principle is to take the alphabetically first triple of cells Y_1, Y_2 and Y_3 in Θ_2 that "work" for a given cell X in Σ_2.

You are encouraged to write your own PROLOG program for this example to test your understanding of the algorithm, and then load it into a PROLOG interpreter.

The following is a transcript of a session on SWI PROLOG.

```
% /Users/jdavoren/Prolog/Prolog-code/sudoku-example-002.pl
compiled 0.00 sec, 65 clauses
1 ?- v_D_3, v_B_4.
```

```
false.
2 ?- v_D_3.
true.
3 ?- v_B_2, v_C_4, v_D_3, v_E_4, v_F_3, v_H_1.
false.
4 ?- v_B_2, v_C_4, v_D_3, v_E_3, v_F_4, v_H_1.
true.
5 ?- v_I_4, v_K_1, v_L_2, v_M_1, v_N_2, v_O_3.
false.
6 ?- v_I_4, v_K_1, v_L_2, v_M_2, v_N_1, v_O_3.
true.
7 ?-
```

A 1	B 2	C 4	D 3
E 3	F 4	G 2	H 1
I 4	J 3	K 1	L 2
M 2	N 1	O 3	P 4

Sudoku example 002 completed

For **Sudoku example 002**, which of the following pairs of clauses can occur in a PROLOG program generated by our algorithm:

(a)
```
v_D_3 :- v_A_1, v_G_2, v_J_3.
v_H_2 :- v_D_3, v_G_2, v_P_4.
```

(b)
```
v_K_1 :- v_G_2, v_J_3, v_P_4.
v_L_2 :- v_J_3, v_K_1, v_P_4.
```

(c)
```
v_F_4 :- v_A_1, v_G_2, v_J_3.
v_N_1 :- v_F_4, v_G_2, v_J_3.
```

For **Sudoku example 002**, the following pairs of clauses can occur in a PROLOG program generated by our algorithm:

(a) `v_D_3 :- v_A_1, v_G_2, v_J_3.` wrong clause
`v_H_2 :- v_D_3, v_G_2, v_P_4.` wrong clause

(b) CORRECT
`v_K_1 :- v_G_2, v_J_3, v_P_4.` STAGE 1
`v_L_2 :- v_J_3, v_K_1, v_P_4.` STAGE 2

(c) `v_F_4 :- v_A_1, v_G_2, v_J_3.` STAGE 1
`v_N_1 :- v_F_4, v_G_2, v_J_3.` wrong clause

6.4 | HOW PROLOG ANSWERS QUERIES

In this section, we will discover how a PROLOG implementation goes about answering queries, and find out why it gives the answers it does.

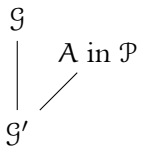
6.4.1 | RESOLUTION PROOF METHOD USED BY PROLOG

We noted early in the chapter that logic programming keeps things "simple for dumb machines" by using just *one* inference rule; this is in contrast with proof trees which have multiple rules, with two for each connective. The proof method is known as *resolution*, and it was first developed for the Horn clause fragment of propositional (and predicate) logic.

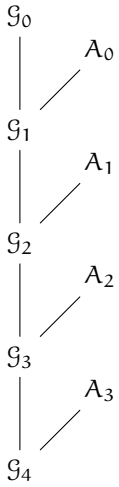
The rule used in PROLOG implementations is known as SLD resolution, standing for Selective Linear Definite clause resolution.

- The S for "selective" is because the rule requires a principle of selection between different clauses in the logic program and between different atoms in the clauses.
- The L for "linear" is because using the rule generates a linear sequence of goals, with the new goal derived from a previous goal by consulting clauses in the logic program.
- The D is for "definite clause", recalling that this term is synonymous with "program clause".

Let $\mathcal{P}$ be a logic program. The proof rule will allow us to derive a new goal $\mathcal{G}'$ from a previous goal $\mathcal{G}$ by consulting program clauses A in $\mathcal{P}$.

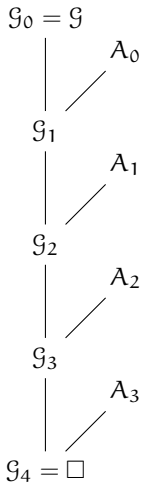


The proof method constructs a chain or sequence of goals with repeated applications of the SLD resolution rule. This can be depicted in a linear-tree diagram, with a sequence of rule applications producing a sequence of goals, and with clauses A from logic program $\mathcal{P}$ appearing as side inputs to the linear sequence of goals.



We include among the derived goals $\mathcal{G}_i$ the *empty goal*, written "□", thought of as the empty list of atoms. As an empty list, □ is *always true* because all its (zero many) atoms are true. The negation of □ is **contradiction**: *always false*.

The proof method implemented in PROLOG is a *refutation* procedure, as is also used in proof trees. To determine whether or not $\mathcal{P} \models \mathcal{G}$, suppose the program $\mathcal{P}$ is true but the initial goal $\mathcal{G}_0 = \mathcal{G}$ is false (that is, one of its atoms is false), and then PROLOG repeatedly applies the SLD resolution rule in an attempt to derive the empty goal □.

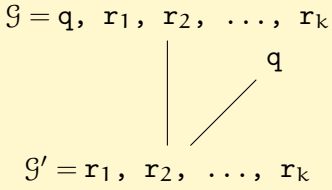


SLD resolution rule:
Given a goal of the form: $\mathcal{G}$ = `q, r1, r2, ..., rk`, with $k \geqslant 0$, PROLOG will start work on the *first* atom `q` in $\mathcal{G}$, and will look for the *first* program clause A in $\mathcal{P}$ whose head is `q`.

Resolution case 1: A is the *fact*:

```
q.
```

Then the new goal $\mathcal{G}'$ is the result of deleting `q` from goal $\mathcal{G}$, and $\mathcal{G}'$ is the *resolvent* of $\mathcal{G}$ with A = `q`. If $\mathcal{G}$ is just `q` then $\mathcal{G}'$ is empty, □.



Here, the goal atom `q` in $\mathcal{G}$ effectively *cancels with* the fact `q.` in $\mathcal{P}$, so that the new goal – the resolvent $\mathcal{G}'$ – is the result of simply deleting `q` from $\mathcal{G}$.

Under the refutation interpretation, we are supposing that $\mathcal{G}$ is false and all the clauses in $\mathcal{P}$ are true. If it were atom `q` in $\mathcal{G}$ that was false, then this directly contradicts the assertion of fact `q.` in $\mathcal{P}$. Hence we can eliminate `q` from consideration, and move on to the next atom in the goal, if there is one.

In the case that $\mathcal{G}$ is the single atom q alone, then the resolvent $\mathcal{G}'$ is the empty goal $\square$, and the sequence of resolution steps can terminate.

Otherwise, the program clause A in $\mathcal{P}$ with head q is a conditional rule.

Resolution case 2: A is a *conditional rule*, with $m \geqslant 1$:

```
q :- p1, p2, ..., pm.
```

Then the new goal $\mathcal{G}'$ replaces the q in $\mathcal{G}$ with body atoms $p_1, p_2, \ldots, p_m$, and $\mathcal{G}'$ is the *resolvent* of $\mathcal{G}$ with program clause A.

$$\mathcal{G} = q, r_1, r_2, \ldots, r_k$$

$$\mathcal{G}' = p_1, p_2, \ldots, p_m, r_1, r_2, \ldots, r_k$$

Here, the goal atom q in $\mathcal{G}$ also *cancels with* the clause head q. within A, but because it is a conditional, the new goal – the resolvent $\mathcal{G}'$ – is the result of replacing the lead atom q in $\mathcal{G}$ with the new sub-goals $p_1, p_2, \ldots, p_m$.

Under the refutation interpretation, we are supposing that $\mathcal{G}$ is false and all the clauses in $\mathcal{P}$ are true. If it is atom q in $\mathcal{G}$ that is false, then since the conditional is assumed to be true, we can apply the principle of *modus tollens* or "denying the consequent", and infer that the antecedent must be false, and thus one of p_1 or p_2 or any of them up to p_m must be false. Thus we get the resolvent $\mathcal{G}'$ as described. Note that in the resolvent $\mathcal{G}'$, the first atom is now p_1, so it will become the target within the next resolution step.

In the search for resolution sequences using the SLD resolution, the following search strategy is used by PROLOG implementations.

Choice points, backtracking and depth-first search:
In developing a sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$ via resolution with program clauses $A_0, A_1, \ldots, A_{n-1}$ from $\mathcal{P}$, respectively, do the following:

- Mark a goal $\mathcal{G}_i$ as a *choice point* if and only if there is a program clause A'_i in $\mathcal{P}$ different from A_i but with the same head atom, the first atom in $\mathcal{G}_i$;
- If the first atom q of the last goal $\mathcal{G}_n$ is such that all the clauses in $\mathcal{P}$ with head atom q have already been explored in earlier resolution steps $\mathcal{G}_i$ for $i < n$, then this goal $\mathcal{G}_n$ is declared a dead-end. In this case, *backtrack* through the sequence to locate the nearest goal $\mathcal{G}_i$ marked as a choice point (with i the largest index less than or equal to n whose goal is a choice point) and restart the search from that goal $\mathcal{G}_i$.
- Continue to extend the sequence from $\mathcal{G}_n$ to a resolvent $\mathcal{G}_{n+1}$, in a *depth-first* manner, whenever $\mathcal{G}_n$ is not $\square$, $\mathcal{G}_n$ is not a dead-end, and there exists a program clause A_n in $\mathcal{P}$ with head atom the first

atom in $\mathcal{G}_n$ that has not yet be explored.

Two cases of dead-end goals $\mathcal{G}_n$ are:

- when the first atom in $\mathcal{G}_n$ is `false`; and
- when the first atom in $\mathcal{G}_n$ is such that there are *no* program clauses in $\mathcal{P}$ with that atom as head.

The first case of dead-end goals is when a goal contains the constant `false`. Under the refutation interpretation, we are supposing that goals are false (the negation of their conjunction is true) while all the clauses in the program $\mathcal{P}$ are true, and searching for a contradiction in the form of the empty goal $\square$. The negation of `false` is `true`, the constant proposition that is always valued true. A goal containing `false` is a dead-end because from it, resolution can never lead to $\square$; indeed, from it, resolution cannot lead anywhere. Recall that $\square$ as a goal has the meaning `true`, so the negation of $\square$ is the contradiction `false`. In brief, a goal containing `false` is a dead-end because it is the negation of the desired goal $\square$.

The second case of dead-end goals is when there is no match for the first atom of the goal among the program clauses of $\mathcal{P}$. When a program $\mathcal{P}$ is compiled, a Prolog interpreter will usually complain if there are any atoms occurring in $\mathcal{P}$ for which there are no program clauses in $\mathcal{P}$ with that atom as head. If the problem atom is `q`, then the Prolog interpreter will treat `q` as "undefined"; expect an error message such as the following in this case:

We saw this in our first Prolog program and query in §6.2.3 above. The remedy is to include a tautology clause `q :- false.` for any atom `q` occurring in $\mathcal{P}$ that is not the head of any other program clause.

```
ERROR: Undefined procedure: q/0
```

Let $\mathcal{P}$ be a logic program loaded into a Prolog interpreter, and suppose goal $\mathcal{G}$ is $r_1, r_2, \ldots, r_k$, where each of these atoms r_j occur as the head of at least one program clause within $\mathcal{P}$. In response to the query:

```
?- r1,r2,...,rk
```

that is, the question whether or not $\mathcal{P} \models \mathcal{G}$, the Prolog interpreter will have one of three responses:

- it answers `true` or `yes`, or
- it answers `false` or `no`, or else
- it gets stuck in a loop and crashes, or reports some other error.

We will consider each of these possibilities in turn.

Prolog will answer `true` to the query `?- r1, r2, ..., rk.`

that is, the question whether or not $\mathcal{P} \models \mathcal{G}$,

if and only if the Prolog interpreter can find a resolution refutation sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \ldots, \mathcal{G}_n$ such that:

- $\mathcal{G}_0 = \mathcal{G}$;
- for each $i < n$, there is a program clause A_i in $\mathcal{P}$ such that goal $\mathcal{G}_{i+1}$ is the resolvent of $\mathcal{G}_i$ with A_i;

- $\mathcal{G}_n = \square$.

Recall our weather example PROLOG program from section §6.2.3, and the query `?- high_fire_danger.`

```
/* weather-001.pl */
windy :- melbourne.
windy :- yackandandah.
light_rain :- melbourne.
dry :- yackandandah.
hot35 :- northern_victoria.
mild25 :- central_victoria.
central_victoria :- melbourne.
northern_victoria :- yackandandah.
high_fire_danger :- windy, dry, hot35.
yackandandah.
melbourne :- false.

1 ?- high_fire_danger.
true.
2 ?-
```

The following resolution sequence shows how a PROLOG interpreter's search response to the query `?- high_fire_danger.` (It is too big to draw is as a linear tree diagram, so we write the goals and program clauses in a list instead.)

$\mathcal{G}_0 =$ `h_f_d`	A_0:	`h_f_d :- windy, dry, hot35.`
$\mathcal{G}_1 =$ `windy, dry, hot35`	A_1:	`windy :- melb.` **Choice point**
$\mathcal{G}_2 =$ `melb, dry, hot35`	A_2:	`melb :- false.`
$\mathcal{G}_3 =$ `false, dry, hot35`		**dead-end**: backtrack to $\mathcal{G}_1$
$\mathcal{G}_1 =$ `windy, dry, hot35`	A_1:	`windy :- yacka.` **2nd choice**
$\mathcal{G}_2 =$ `yacka, dry, hot35`	A_2:	`yacka.`
$\mathcal{G}_3 =$ `dry, hot35`	A_3:	`dry :- yacka.`
$\mathcal{G}_4 =$ `yacka, hot35`	A_4:	`yacka.`
$\mathcal{G}_5 =$ `hot35`	A_5:	`hot35 :- n_victoria.`
$\mathcal{G}_6 =$ `n_victoria`	A_6:	`n_victoria :- yacka.`
$\mathcal{G}_7 =$ `yacka`	A_7:	`yacka.`
$\mathcal{G}_8 = \square$		

Starting from the initial goal $\mathcal{G}_0$ = `high_fire_danger`, PROLOG looks for the first clause in the program with head `high_fire_danger`. This is the conditional `high_fire_danger :- windy, dry, hot35.`, labelled A_0. The resolvent $\mathcal{G}_1$ is then `windy, dry, hot35`. PROLOG looks for the first clause in the program with head `windy`. This is the conditional `windy :- melbourne.`, labelled A_1, but PROLOG makes a note that this is a choice point because there is another clause in the program with head `windy`, namely the conditional `windy :- yackandandah.` The resolvent $\mathcal{G}_2$ of $\mathcal{G}_1$ with A_1 is now `melbourne, dry, hot35`. PROLOG looks for the first clause in the program with head `melbourne`. This is the tautology conditional `melbourne :- false.`, labelled A_2. The resolvent $\mathcal{G}_3$ of $\mathcal{G}_2$ with A_2 is now the dead-end goal `false, dry, hot35`.

At this point, we must backtrack to the last choice point, which is goal $\mathcal{G}_1$: `windy, dry, hot35`. Prolog then looks for the second clause in the program with head `windy`. This is the conditional `windy :- yackandandah.`, and we re-use the label A_1. The resolvent $\mathcal{G}_2$ of $\mathcal{G}_1$ with A_1 is now the goal `yackandandah, dry, hot35`. Prolog looks for the first clause in the program with head `yackandandah`. This is the fact `yackandandah.`, labelled A_2. The resolvent $\mathcal{G}_3$ of $\mathcal{G}_2$ with A_2 is now `dry, hot35`, and the search continues.

Prolog looks for the first clause in the program with head `dry`. This is the conditional `dry :- yackandandah.` The resolvent is the goal `yackandandah, hot35` which in turns becomes the goal `hot35` after resolution with the fact `yackandandah.` Another chain through the two conditional clauses `hot35 :- northern_victoria.` and `northern_victoria :- yackandandah.` leads to the goal $\mathcal{G}_7$ of just `yackandandah`. A further final resolution with the fact `yackandandah.` and the result is the empty goal $\square$.

After excising the dead-end and backtracking, we get a resolution refutation sequence witnessing the answer `true` to query `?- high_fire_danger.`

$\mathcal{G}_0 =$ `h_f_d`	A_0: `h_f_d :- windy, dry, hot35.`
$\mathcal{G}_1 =$ `windy, dry, hot35`	A_1: `windy :- yacka.`
$\mathcal{G}_2 =$ `yacka, dry, hot35`	A_2: `yacka.`
$\mathcal{G}_3 =$ `dry, hot35`	A_3: `dry :- yacka.`
$\mathcal{G}_4 =$ `yacka, hot35`	A_4: `yacka.`
$\mathcal{G}_5 =$ `hot35`	A_5: `hot35 :- n_victoria.`
$\mathcal{G}_6 =$ `n_victoria`	A_6: `n_victoria :- yacka.`
$\mathcal{G}_7 =$ `yacka`	A_7: `yacka.`
$\mathcal{G}_8 = \square$	

The SLD resolution proof method implemented in Prolog is *sound*: if Prolog answers `true` to the query of $\mathcal{G}$ after loading program $\mathcal{P}$, then it is in fact the case that $\mathcal{P} \models \mathcal{G}$.

The soundness of SLD resolution requires verifying that if there exists a resolution refutation sequence from $\mathcal{G}$ to $\square$ then it impossible for $\mathcal{G}$ to be false at the same time that all the clauses in $\mathcal{P}$ are true. We come back to this a little later, when we re-consider program clauses, negated goals and resolution re-written in clausal form.

In response to a query, the second possibility is that Prolog answers `false`.

> Prolog will answer `false` to the query `?- r1, r2, ..., rk.`
>
> that is, the question whether or not $\mathcal{P} \models \mathcal{G}$,
>
> *if and only if* a systematic *depth-first* search with *backtracking* does not produce a resolution refutation sequence ending with $\square$, where the search is through the tree of all possible resolution sequences starting with $\mathcal{G}$ and with input from clauses as ordered in $\mathcal{P}$.

For example, let $\mathcal{P}$ be the program `weather-001.pl` and let $\mathcal{G}$ be the goal query: `?- mild25.` A systematic depth-first search does not produce a resolution refutation sequence ending with $\square$; the search results are as follows:

$\mathcal{G}_0 =$ `mild25`	A_0:	`mild25 :- c_victoria.`
$\mathcal{G}_1 =$ `c_victoria`	A_1:	`c_victoria :- melb.`
$\mathcal{G}_2 =$ `melb`	A_2:	`melb :- false.`
$\mathcal{G}_3 =$ `false`		**dead-end**: but nowhere to backtrack

In this case, the search tree of possible sequences collapses to a single linear branch, as there is no opportunity for backtracking. This is because there is only one program clause in $\mathcal{P}$ with head `mild25`, only one with head `central_victoria`, and only one with head `melbourne`.

For another example with some branching in the search tree, consider the following PROLOG program `example-001.pl` and with a query answered `false.`

```
/* example-001.pl */
p :- q.
q :- w, z.
q :- s.
r :- false.
s :- r.
s :- w.
w :- false.
z.

?- p.
false.
```

In response to the query `?- p.`, the search sequence followed by a PROLOG interpreter in its failed search for a resolution refutation sequence is as follows.

$\mathcal{G}_0$:	`p.`	A_0:	`p :- q.`	
$\mathcal{G}_1$:	`q.`	A_1:	`q :- w, z.`	1st choice q
$\mathcal{G}_2$:	`w, z.`	A_2:	`w :- false.`	
$\mathcal{G}_3$:	`false, z.`		backtrack to $\mathcal{G}_1$	
$\mathcal{G}_1$:	`q.`	A_1:	`q :- s.`	2nd choice q
$\mathcal{G}_2$:	`s.`	A_2:	`s :- r.`	1st choice s
$\mathcal{G}_3$:	`r.`	A_3:	`r :- false.`	
$\mathcal{G}_4$:	`false.`		backtrack to $\mathcal{G}_2$	
$\mathcal{G}_2$:	`s.`	A_2:	`s :- w.`	2nd choice s
$\mathcal{G}_3$:	`w.`	A_3:	`w :- false.`	
$\mathcal{G}_4$:	`false.`		search failed	

Notice that the PROLOG interpreter follows strict program order, taking as first choice the first listed program clause with that head, and moving on to the second listed program clause with the same head, and so on. Within a goal, resolution steps are applied to atoms as listed in left-to-right order, and the order is as the atoms appear in the body of the program clause they came from. For example, when the goal became $\mathcal{G}_2$: `w, z.` from program clause A_1 : `q :- w,z.`, resolution is first applied to atom `w` and then to atom `z`, because that is there listed order.

In response to a query, the last remaining possibility is that PROLOG crashes and does not produce any answer at all. For example, this will be the case when the program $\mathcal{P}$ contains an *implication loop*, such as:

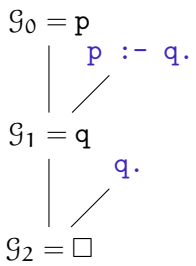
```
p :- q.   q :- s.   s :- p.
```

and the goal $\mathcal{G}$ contains an atom $\mathtt{r}_i$ from which the loop is accessible via resolutions.

```
/* loop-001.pl */
p :- q.
q :- p.
q.
```

```
% /Users/jdavoren/Prolog/Prolog-code/loop-001
 compiled 0.00 sec, 4 clauses
1 ?- p.
ERROR: Out of local stack
 Exception: (3,941,683) q ?  creep
 Exception: (3,941,681) q ?
```

This seemingly trivial example presents a real **problem**: it means that the SLD resolution proof method as implemented in PROLOG is in fact *incomplete*. Let $\mathcal{P}$ be the program listed in `loop-001.pl` and let $\mathcal{G}$ be the goal consisting of just the atom `p`. Then $\mathcal{P} \models \mathcal{G}$: the atom `p` really *is* a logical consequence of the program $\mathcal{P}$. Indeed, there is a resolution refutation sequence starting from goal $\mathcal{G}$ and leading to the empty goal $\square$, as shown:



The problem is that PROLOG gets stuck in the loop flipping between `p :- q.` and the converse conditional `q :- p.`, and it never gets to find the fact `q.`. So the PROLOG interpreter is not able to return any answer at all to the query `?- p.` So we have a simple case of logical consequence that cannot be correctly answered by PROLOG.

One might think that a smarter PROLOG interpreter would try to implement a *loop-checking* strategy, to avoid this incompleteness. However, *loop-checking* is computationally quite expensive and inefficient. There are PROLOG *meta-interpreters* which take program data as input and use it as a basis for additional computations – such as selective loop-checking – in order to modify the control mechanisms used by PROLOG as it searches for a resolution refutation sequence. This topic is beyond the scope of this introduction.

Here's an exercise for you to try. One of the skills to be developed in this chapter is that of determining how PROLOG will respond to a sequence of queries after loading a given PROLOG program.

```
/* exercise.pl */
p1 :- q1
p2 :- q2.
p3 :- false.
q1 :- r1.
q2 :- p2.
q2.
r1.
r2 :- p3.
```

Complete the table to show how PROLOG will answer the goal queries shown. For each goal query, mark one of the options `true`, `false`, or crash. If you choose "crash", further answer YES or NO to the question whether or not that goal is a logical consequence of the program `exercise.pl`.

Query:	`true`	`false`	crash
`?- p1.`			
`?- p2.`			
`?- p3.`			
`?- q1.`			
`?- q2.`			
`?- r1.`			
`?- r2.`			

The answers are given below (inverted, as usual).

Completed table showing how PROLOG will answer the goal queries shown:

Query:	`true`	`false`	crash
`?- p1.`	×		
`?- p2.`			YES
`?- p3.`		×	
`?- q1.`	×		
`?- q2.`			YES
`?- r1.`	×		
`?- r2.`		×	

6.4.2 | WHY RESOLUTION WORKS: CLAUSAL FORM

Recall from section §6.2.1 that a logic formula is called a clause if and only if it is a disjunction of literals, where a literal is either an atomic formula or the negation of an atomic formula, and we allow disjunctions of size 1 (consisting of a single formula). A logic formula is called a Horn clause if and only if it

is logically equivalent to a clause containing at most one positive literal. The resolution proof method was enveloped for the Horn clause fragment of logic.

In a setting where all formulas of interest are clauses, a disjunction of literals:

$$(l_1 \vee l_2 \vee \cdots \vee l_n)$$

may be re-written in clausal notation as simply a *list* or set of literals:

$$\{ l_1, l_2, \ldots l_n \}.$$

Each program clause `q :- p1, p2, ..., pm.` (for $m \geqslant 0$) is logically equivalent to the Horn clause:

$$\{ q, \sim p_1, \sim p_2, \ldots, \sim p_m \}$$

with exactly one positive literal and zero or more negative literals (zero if the program clause is a fact).

For each goal `r1, r2, ..., rk.` (for $k \geqslant 1$), let B be the *goal formula*: the conjunction of the atoms r_i. Then the negation ~B is logically equivalent to the Horn clause:

$$\{ \sim r_1, \sim r_2, \ldots, \sim r_k \}$$

with zero positive literals, and so all literals negative.

If goal $\mathcal{G}$ in negated clausal form is $\{ \sim q, \sim r_1, \sim r_2, \ldots, \sim r_k \}$ and program clause A is the fact $\{ q. \}$, then the *resolvent* $\mathcal{G}'$ of $\mathcal{G}$ with A gives the new negated goal clause:

$$\{ \sim r_1, \sim r_2, \ldots, \sim r_k \}$$

which is formed by taking the union or merge of the negated clause for $\mathcal{G}$ with the program clause, and cancelling out the ~q with the q (resolving on the literal q). In the case $k = 0$ and $\mathcal{G}'$ is the empty goal $\square$, the corresponding negated goal clause is the *always false* constant `false`.

If goal $\mathcal{G}$ in negated clausal form is $\{ \sim q, \sim r_1, \sim r_2, \ldots, \sim r_k \}$ and program clause A from $\mathcal{P}$ is $\{ q, \sim p_1, \sim p_2, \ldots, \sim p_m \}$, then the *resolvent* $\mathcal{G}'$ of $\mathcal{G}$ with A gives the new negated goal clause:

$$\{ \sim p_1, \sim p_2, \ldots, \sim p_m, \sim r_1, \sim r_2, \ldots, \sim r_k \}$$

which is formed by taking the union or merge of the negated clause for $\mathcal{G}$ with the clausal form for A, and cancelling out the ~q with the q (resolving on the literal q).

Return to our example Prolog program `example-001.pl` and the failed search for a resolution refutation sequence in response to the query `?- p.` Re-written with negated goals and program clauses in clausal notation, the search sequence followed by a Prolog interpreter in its failed search is as follows.

$\mathcal{G}_0$:	{~p }	A_0:	{p,~q}	
$\mathcal{G}_1$:	{~q}	A_1:	{q,~w, ~z}	1st choice q
$\mathcal{G}_2$:	{~w,~z}	A_2:	{w, ~false}	
$\mathcal{G}_3$:	{~false, ~z}		backtrack to $\mathcal{G}_1$	
$\mathcal{G}_1$:	{~q}	A_1:	{q,~s}	2nd choice q
$\mathcal{G}_2$:	{~s}	A_2:	{s,~r}	1st choice s
$\mathcal{G}_3$:	{~r}	A_3:	{r, ~false}	
$\mathcal{G}_4$:	{~false}		backtrack to $\mathcal{G}_2$	
$\mathcal{G}_2$:	{~s}	A_2:	{s,~w}	2nd choice s
$\mathcal{G}_3$:	{~w}	A_3:	{w, ~false}	
$\mathcal{G}_4$:	{~false}		search failed	

This failed search for a contradiction in response to the query `?- p.` shows that, under the assumption that all the clauses in $\mathcal{P}$ = `example-001.pl` are true, ~p implies ~`false`; equivalently, ~p implies `true`. What this means is that ~p together with $\mathcal{P}$ cannot yield a contradiction. We can then conclude that ~p is *consistent* with $\mathcal{P}$: ~p together with all the clauses in $\mathcal{P}$ can be true at the same time under some valuation v.

Observe that using the constant `true`, we end up with $(A \supset$ `true`$)$ is a tautology, for any formula A, and all tautologies are logically equivalent to `true`.

In applying resolution steps, we are always assuming that all the clauses in the program $\mathcal{P}$ are true. In establishing the soundness of the resolution rule deriving $\mathcal{G}'$ from $\mathcal{G}$ together with a program clause from $\mathcal{P}$, the core result is the following:

> if the negated clause for $\mathcal{G}$ is true (one of the atoms in $\mathcal{G}$ is false),
> then the negated clause for $\mathcal{G}'$ is true (one of the atoms in $\mathcal{G}'$ is false).

Suppose PROLOG answers `true` because there is a resolution refutation sequence starting with $\mathcal{G}$ ending with the empty clause $\mathcal{G}_n = \square$. Then the inference steps have lead to a contradiction because the corresponding negated clause for $\square$ is never true, as it is the constant `false`. We can then trace back up the resolution sequence to the original goal $\mathcal{G}_0 = \mathcal{G}$, and conclude that the negated clause for $\mathcal{G}$ is false, and thus the goal $\mathcal{G}$ is true. Hence we can conclude that $\mathcal{P} \models \mathcal{G}$: $\mathcal{G}$ is a logical consequence of $\mathcal{P}$.

Suppose instead PROLOG answers `false` because a systematic search for a resolution refutation sequence leads in all cases to a dead-end goal that is not $\square$. Then the negated goal clause for the original goal $\mathcal{G}$ is consistent with $\mathcal{P}$ being true. This means that $\mathcal{P} \cup \{\sim B\}$ is satisfiable, where B is the conjunction of atoms in $\mathcal{G}$. Hence we can conclude that $\mathcal{P} \not\models \mathcal{G}$: $\mathcal{G}$ is not a logical consequence of $\mathcal{P}$.

With the refutation approach as used in the proof tree method, a failed proof attempt in a tree with root $A_1, A_2, \ldots, A_n$ and ~B is manifest in an open branch from which we can directly extract a counter-example to logical consequence: a valuation of atoms that makes all of $A_1, A_2, \ldots, A_n$ true and B false. So we can constructively extract semantic information from the failed proof attempt.

In contrast, the refutation approach as used in resolution for PROLOG is not so constructive. A failed proof attempt to prove goal $\mathcal{G}$ from program $\mathcal{P}$ is manifest in a search tree of refutation sequences, all of whose branches lead to a dead-end and not to $\square$. Unfortunately, there is no easy way to extract out semantic information to build a counter-example valuation that makes all of $\mathcal{P}$ true and $\mathcal{G}$ false.

Considered as a formal proof system, the SLD resolution refutation method is both complete and sound for propositional logic. Completeness says that for a logic program $\mathcal{P}$ and a goal $\mathcal{G}$, if $\mathcal{P} \models \mathcal{G}$ then there exists a resolution refutation sequence starting with $\mathcal{G}$ ending with the empty clause $\mathcal{G}_n = \square$. But PROLOG implementations are *incomplete* because they can get stuck in a loop and fail to find a resolution refutation sequence when it in fact exists.

Summary of Propositional PROLOG soundness/correctness:
Given a propositional logic program $\mathcal{P}$ loaded into PROLOG together with a goal $\mathcal{G}$, all of whose atoms occur as heads of clauses in $\mathcal{P}$,

- If PROLOG answers `true` to the query `?- G` then $\mathcal{P} \models \mathcal{G}$.
- If PROLOG answers `false` to the query `?- G` then $\mathcal{P} \not\models \mathcal{G}$.
- But ... **incompleteness:** PROLOG can crash and not give an answer when it really is the case that $\mathcal{P} \models \mathcal{G}$.

6.5 | NEGATION IN PROLOG

6.5.1 | NEGATION AS FAILURE IN PROLOG

From what we have seen so far, PROLOG has no direct way of handling negative information. Program clauses and goals as they are written only contain positive atoms, with negation only appearing indirectly, when we move to the clausal forms of program clauses and negated goals, in order to clarify our understanding of the resolution rule.

In practice, knowledge bases and databases tend to give a primary role to positive information. A database for a train timetable will include the positive fact that there is a train service from A to B at time 3:25pm, but it will not include any negative facts like there is *not* a scheduled train service from A to B at other times like 3:19pm or 3:32pm. However, when we read the timetable, we are not actually wrong to infer that there is no scheduled train service at any times other than those listed.

In locating negative information, what we can do is load up a logic program $\mathcal{P}$ into PROLOG, and pose a query positively. Then from the answer:

```
?- q.
false.
```

we might be tempted to conclude ~q "*sort-of follows*" from program $\mathcal{P}$.

Semantically, what we know is that the negated goal ~q is *consistent with* the information in the program $\mathcal{P}$: there is at least once valuation v of atoms such that ~q is true under v at the same time all of $\mathcal{P}$ is true under v. This is what it means for q to *not* be a logical consequence of $\mathcal{P}$. However, *be warned*: ~q being consistent with $\mathcal{P}$ is a weaker relationship; we *cannot* conclude that the formula ~q is a *logical consequence* of program $\mathcal{P}$. It may be possible for all of $\mathcal{P}$ to be true and at the same time, q also be true. There may be some crucial facts or conditional rules missing from $\mathcal{P}$ that, were they present, we then would be able to derive the goal.

More succinctly: $\mathcal{P} \not\models \mathcal{G}$ does *not* imply $\mathcal{P} \models \sim\mathcal{G}$.

From PROLOG's answer `false.` in respond to the query `?- q.`, we can validly draw the conclusion that ~q is true if we make an additional assumption called the *closed world assumption* (CWA). CWA is the assumption that the "world" or model defined by the facts and rules in $\mathcal{P}$ – what is known by the PROLOG interpreter – is *all there is to know*.

For example, suppose there is a total of three conditional clauses in $\mathcal{P}$ with the head q, say `q :- p_1`, `q :- p_2`, and `q :- p_3`. In this case, CWA says that these three clauses together amount to a *definition* of the atom q. This can be expressed by assuming that the *bi-conditional* formula:

$$\mathtt{q} \equiv (\mathtt{p\_1} \vee \mathtt{p\_2} \vee \mathtt{p\_3})$$

is true. Such a biconditional is sometimes called the *completion* of the three clauses `q :- p_1`, `q :- p_2`, and `q :- p_3` within $\mathcal{P}$, according to CWA.

In respond to the query `?- q.`, when a systematic depth-first search for a resolution refutation sequence leads to a dead-end for each of `p_1`, `p_2` and `p_3`, the PROLOG interpreter answers `false.` because it has exhausted all these three and failed. According to CWA, these three are all and only the ways in which `q` can be true. Thus from what it knows of the world, according to program $\mathcal{P}$, the atom `q` is false, and thus ~`q` must be true.

Another way of formulating the closed world assumption is using an operator which expands a program. For any set of formulas $\mathcal{P}$, (which does not have to be a logic program), define the expansion CWA($\mathcal{P}$) as follows:

$$\text{CWA}(\mathcal{P}) \;:=\; \mathcal{P} \cup \{\sim\mathtt{q} \mid \mathtt{q} \text{ is an atom and } \mathcal{P} \not\models \mathtt{q}\}.$$

So the set CWA($\mathcal{P}$) contains all of $\mathcal{P}$ together with the negation of any atoms that are not logical consequences of $\mathcal{P}$. In these terms, the closed world assumption is that, instead of a program $\mathcal{P}$ being true, we assume the expanded set CWA($\mathcal{P}$) is true.

The relation of logical consequence is *monotonic* in the sense that if $\mathcal{P}$ and $\mathcal{P}'$ are two sets of formulas, and $\mathcal{P}$ is included in (is a subset of) $\mathcal{P}'$, then:

$$\text{if } \mathcal{P} \models \mathtt{B} \quad \text{then } \mathcal{P}' \models \mathtt{B}$$

for all logical formulas B. This says that if we expand our information set from $\mathcal{P}$ to the larger set $\mathcal{P}'$, then we have at least as many logical consequences; all of the logical consequences of $\mathcal{P}$ are carried along.

The closed world assumption operator CWA is *non-monotonic* is the sense that if $\mathcal{P}$ and $\mathcal{P}'$ are two sets of formulas, and $\mathcal{P}$ is included in $\mathcal{P}'$, then it is *not* the case that:

$$\text{if B is in CWA}(\mathcal{P}) \quad \text{then B is in CWA}(\mathcal{P}')$$

for all logical formulas B. If we expand our information set from $\mathcal{P}$ to the larger set $\mathcal{P}'$, it may be that the extra formulas in $\mathcal{P}'$ may make it the case that $\mathcal{P}' \models \mathtt{q}$ for some atom `q` even though $\mathcal{P} \not\models \mathtt{q}$. Then ~`q` is in CWA($\mathcal{P}$) but ~`q` is not in CWA($\mathcal{P}'$). Non-monotonicity means that some consequences can be invalidated by adding more knowledge.

6.5.2 | USING NEGATION IN QUERIES

Here we introduce a new symbol, "`\+`" that is read as *not provable*; the `\` stands for *not* and the `+` stands for *provable*.

> **Syntax:** We can write in a query `?- \+ q`
>
> to ask whether the positive query for `q` will return `false.`

We will illustrate using negation in queries with a small PROLOG program.

```
/* neg-001.pl */
p1 :- q1
p2 :- false.
q1.
q2 :- p2.
```

Consider the following sequence of queries and PROLOG answers.

```
% /Users/jdavoren/Prolog/Prolog-code/neg-001
 compiled 0.00 sec, 5 clauses
1 ?- p1.
true
2 ?- p2.
false
3 ?- q1.
true
4 ?- q2.
false
5 ?- \+ p2.
true
6 ?- \+ q2.
true
7 ?- \+ p1.
false
8 ?-  p1, \+ p2, q1, \+ q2.
true
```

Let $\mathcal{P}$ be the program listed in `neg-001.pl`. For a goal $\mathcal{G}$, we have the logical consequence relationship $\mathcal{P} \models \mathcal{G}$ exactly when, for all truth valuations ν to the atoms, if ν makes all the clauses in $\mathcal{P}$ true, then ν also makes all the atoms in $\mathcal{G}$ true.

Now if ν makes all the clauses in $\mathcal{P}$ true, then it follows that $\nu(\texttt{p1}) = 1$ and $\nu(\texttt{q1}) = 1$, because ν makes true both the clauses `p1 :- q1.` and `q1.`

This directly confirms that `p1` and `q1` really are logical consequences of the program $\mathcal{P}$ – which we know already from **true** answers to queries for `p1` and `q1`, plus soundness of PROLOG.

The closed world assumption is that we only consider valuations ν that makes true all of the clauses in the expanded set CWA($\mathcal{P}$). In particular, the negations ~p2 and ~q2 are both in CWA($\mathcal{P}$).

To see the discrepancy between $\mathcal{P}$ and CWA($\mathcal{P}$), consider all the valuations ν that makes all the clauses in $\mathcal{P}$ true. For the two $\mathcal{P}$ clauses:

$$A_1\texttt{: p2 :- false.} \text{ and } A_2\texttt{: q2 :- p2.}$$

their both being true under a valuation ν does *not* imply that $\nu(\texttt{p2}) = 0$ and $\nu(\texttt{q2}) = 0$, as is required for ν to make all the formulas in CWA($\mathcal{P}$) true. Rather, their being both false is just one among several possibilities for the atoms p2 and q2. A different valuation ν' that gives $\nu'(\texttt{p2}) = 1$ and $\nu'(\texttt{q2}) = 1$ will also make the program clauses A_1 and A_2 both true. Yet another different valuation $\nu^\dagger$ that gives $\nu^\dagger(\texttt{p2}) = 0$ and $\nu^\dagger(\texttt{q2}) = 1$ will also make the program clauses A_1 and A_2 both true. So ~p2 and ~q2 are *not* logical consequences of program $\mathcal{P}$, and neither is the formula:

$$(\texttt{p1 \& \sim p2 \& q1 \& \sim q2})$$

the goal for which is queried in line 8 above and answered `true`.

We have only allowed the use of the not-provable symbol `\+` preceding atoms within a goal query. One might reasonably ask what would happen if we were allowed use it in the body of a program clause.

```
/* neg-trouble.pl */
p :- \+ p

% /Users/jdavoren/Prolog/Prolog-code/neg-trouble
 compiled 0.00 sec, 2 clauses

1 ?- p.
ERROR: Out of local stack
2 ?-
```

If the clause `p :- \+ p` is translated as the logic formula $(\sim p \supset p)$, then it is logically equivalent to $(p \vee p)$, which is in turn equivalent to p. So the atom `p` should be a logical consequence of the program `neg-trouble.pl` and PROLOG should answer `true`. The problem is that the PROLOG interpreter gets stuck in a loop with a stack overflow, so no answer is returned.

The moral of this example is that were we to allow use of the not-provable symbol in the body of a program clause, we would only make worse the incompleteness of PROLOG by creating more situations in which an interpreter would get stuck in a loop and crash.

In summary: the negation-like not-provable symbol `\+` should be approached with caution!

- `\+` should be used only within a goal query, preceding an atom, to ask whether the positive query for that atom will fail.
- A `true` answer to a negated query does not allow us to conclude the negated atom is a logical consequence of the logic program.

FURTHER LEARNING

The aim of this chapter is to give a basic introduction to automated reasoning using propositional logic programming. In the sequel Logic: Language and Information 2, there is a further applications section on logic programming in the enriched language of predicate logic.

For those with a deeper interest in logic programming, there is a wealth of textbooks and online resources available. These include:

- Bartak, Roman. *Online Guide to* PROLOG *Programming*. http://kti.mff.cuni.cz/~bartak/prolog/ (Accessed 9 March 2014).
- Bratko, Ivan. *Prolog Programming for Artificial Intelligence* (4th Edition), (Pearson Education Canada, 2011).
- Lloyd, John. *Foundations of Logic Programming* (2nd edition), (Springer-Verlag, Berlin, 1987).
- Wilson, Bill. *The* PROLOG *Dictionary*. http://www.cse.unsw.edu.au/~billw/prologdict.html (Accessed 9 March 2014).

Sudoku example 001

A	B 1	C	D
E	F	G 2	H
I	J	K	L 4
M 3	N	O	P

```
/* sudoku-example-001.pl */
   v_B_1.
   v_G_2.
   v_L_4.
   v_M_3.
   v_D_3   :-  v_B_1, v_G_2, v_L_4.
   v_E_4   :-  v_B_1, v_G_2, v_M_3.
   v_J_2   :-  v_B_1, v_L_4, v_M_3.
   v_O_1   :-  v_G_2, v_L_4, v_M_3.
   v_A_2   :-  v_B_1, v_D_3, v_E_4.
   v_C_4   :-  v_B_1, v_D_3, v_G_2.
   v_F_3   :-  v_B_1, v_E_4, v_G_2.
   v_H_1   :-  v_D_3, v_E_4, v_G_2.
   v_I_1   :-  v_E_4, v_J_2, v_M_3.
   v_K_3   :-  v_J_2, v_L_4, v_O_1.
   v_N_4   :-  v_J_2, v_M_3, v_O_1.
   v_P_2   :-  v_L_4, v_M_3, v_O_1.
   v_A_1 :- false.   v_A_3 :- false.   v_A_4 :- false.
   v_B_2 :- false.   v_B_3 :- false.   v_B_4 :- false.
   v_C_1 :- false.   v_C_2 :- false.   v_C_3 :- false.
   v_D_1 :- false.   v_D_2 :- false.   v_D_4 :- false.
   v_E_1 :- false.   v_E_2 :- false.   v_E_3 :- false.
   v_F_1 :- false.   v_F_2 :- false.   v_F_4 :- false.
   v_G_1 :- false.   v_G_3 :- false.   v_G_4 :- false.
   v_H_2 :- false.   v_H_3 :- false.   v_H_4 :- false.
   v_I_2 :- false.   v_I_3 :- false.   v_I_4 :- false.
   v_J_1 :- false.   v_J_3 :- false.   v_J_4 :- false.
   v_K_1 :- false.   v_K_2 :- false.   v_K_4 :- false.
   v_L_1 :- false.   v_L_2 :- false.   v_L_3 :- false.
   v_M_1 :- false.   v_M_2 :- false.   v_M_4 :- false.
   v_N_1 :- false.   v_N_2 :- false.   v_N_3 :- false.
   v_O_2 :- false.   v_O_3 :- false.   v_O_4 :- false.
   v_P_1 :- false.   v_P_3 :- false.   v_P_4 :- false.
  /* end of file:  sudoku-example-001.pl */
```

Sudoku example 002

A 1	B	C	D
E	F	G 2	H
I	J 3	K	L
M	N	O	P 4

```
/* sudoku-example-002.pl */
   v_A_1.
   v_G_2.
   v_J_3.
   v_P_4.
   v_D_3   :-  v_A_1, v_G_2, v_P_4.
   v_F_4   :-  v_A_1, v_G_2, v_J_3.
   v_K_1   :-  v_G_2, v_J_3, v_P_4.
   v_M_2   :-  v_A_1, v_J_3, v_P_4.
   v_B_2   :-  v_A_1, v_D_3, v_F_4.
   v_C_4   :-  v_A_1, v_D_3, v_G_2.
   v_E_3   :-  v_A_1, v_F_4, v_M_2.
   v_H_1   :-  v_D_3, v_G_2, v_P_4.
   v_I_4   :-  v_A_1, v_J_3, v_M_2.
   v_L_2   :-  v_J_3, v_K_1, v_P_4.
   v_N_1   :-  v_J_3, v_M_2, v_P_4.
   v_O_3   :-  v_K_1, v_M_2, v_P_4.
   v_A_2 :- false.   v_A_3 :- false.   v_A_4 :- false.
   v_B_1 :- false.   v_B_3 :- false.   v_B_4 :- false.
   v_C_1 :- false.   v_C_2 :- false.   v_C_3 :- false.
   v_D_1 :- false.   v_D_2 :- false.   v_D_4 :- false.
   v_E_1 :- false.   v_E_2 :- false.   v_E_4 :- false.
   v_F_1 :- false.   v_F_2 :- false.   v_F_3 :- false.
   v_G_1 :- false.   v_G_3 :- false.   v_G_4 :- false.
   v_H_2 :- false.   v_H_3 :- false.   v_H_4 :- false.
   v_I_1 :- false.   v_I_2 :- false.   v_I_3 :- false.
   v_J_1 :- false.   v_J_2 :- false.   v_J_4 :- false.
   v_K_2 :- false.   v_K_3 :- false.   v_K_4 :- false.
   v_L_1 :- false.   v_L_3 :- false.   v_L_4 :- false.
   v_M_1 :- false.   v_M_3 :- false.   v_M_4 :- false.
   v_N_2 :- false.   v_N_3 :- false.   v_N_4 :- false.
   v_O_1 :- false.   v_O_2 :- false.   v_O_4 :- false.
   v_P_1 :- false.   v_P_2 :- false.   v_P_3 :- false.
  /* end of file:  sudoku-example-002.pl */
```

Sudoku example 003

A	B 4	C	D
E	F	G	H 1
I 2	J	K	L
M	N	O 3	P

```
/* sudoku-example-003.pl */
   v_B_4.
   v_H_1.
   v_I_2.
   v_O_3.
   v_C_3   :-  v_B_4, v_H_1, v_O_3.
   v_E_3   :-  v_B_4, v_H_1, v_I_2.
   v_L_4   :-  v_H_1, v_I_2, v_O_3.
   v_N_1   :-  v_B_4, v_I_2, v_O_3.
   v_A_1   :-  v_B_4, v_E_3, v_I_2.
   v_D_3   :-  v_B_4, v_C_2, v_H_1.
   v_F_2   :-  v_B_4, v_E_3, v_H_1.
   v_G_4   :-  v_C_2, v_H_1, v_O_3.
   v_J_3   :-  v_B_4, v_I_2, v_N_1.
   v_K_1   :-  v_I_2, v_L_4, v_O_3.
   v_M_4   :-  v_I_2, v_N_1, v_O_3.
   v_P_2   :-  v_H_1, v_L_4, v_O_3.
   v_A_2 :- false.   v_A_3 :- false.   v_A_4 :- false.
   v_B_1 :- false.   v_B_2 :- false.   v_B_3 :- false.
   v_C_1 :- false.   v_C_3 :- false.   v_C_4 :- false.
   v_D_1 :- false.   v_D_2 :- false.   v_D_4 :- false.
   v_E_1 :- false.   v_E_2 :- false.   v_E_4 :- false.
   v_F_1 :- false.   v_F_3 :- false.   v_F_4 :- false.
   v_G_1 :- false.   v_G_2 :- false.   v_G_3 :- false.
   v_H_2 :- false.   v_H_3 :- false.   v_H_4 :- false.
   v_I_1 :- false.   v_I_3 :- false.   v_I_4 :- false.
   v_J_1 :- false.   v_J_2 :- false.   v_J_4 :- false.
   v_K_2 :- false.   v_K_3 :- false.   v_K_4 :- false.
   v_L_1 :- false.   v_L_2 :- false.   v_L_3 :- false.
   v_M_1 :- false.   v_M_2 :- false.   v_M_3 :- false.
   v_N_2 :- false.   v_N_3 :- false.   v_N_4 :- false.
   v_O_1 :- false.   v_O_2 :- false.   v_O_4 :- false.
   v_P_1 :- false.   v_P_3 :- false.   v_P_4 :- false.
  /* end of file: sudoku-example-003.pl */
```

REFERENCES

[1] ALAN R. ANDERSON AND NUEL D. BELNAP. *Entailment: The Logic of Relevance and Necessity*, volume 1. Princeton University Press, Princeton, 1975. [Cited on page 50]

[2] ALAN ROSS ANDERSON, NUEL D. BELNAP, AND J. MICHAEL DUNN. *Entailment: The Logic of Relevance and Necessity*, volume 2. Princeton University Press, Princeton, 1992. [Cited on page 50]

[3] IRVING H. ANELLIS. "From semantic tableaux to Smullyan trees: a history of the development of the falsifiability tree method". *Modern Logic*, 1(1):36–69, 1990. [Cited on page 66]

[4] JC BEALL AND GREG RESTALL. *Logical Pluralism*. Oxford University Press, Oxford, 2006. [Cited on page 139]

[5] NUEL BELNAP. "Declaratives are not enough". *Philosophical Studies*, 59(1):1–30, 1990. [Cited on page 11]

[6] B. A. DAVEY AND H. A. PRIESTLEY. *Introduction to Lattices and Order*. Cambridge University Press, Cambridge, 1990. [Cited on page 125]

[7] COLIN HOWSON. *Logic with Trees: An introduction to symbolic logic*. Routledge, 1996. [Cited on pages 4, 66]

[8] JAN ŁUKASIEWICZ. "On Determinism". In L. BORKOWSKI, editor, *Selected Works*. North Holland, Amsterdam, 1970. [Cited on page 129]

[9] EDWIN D. MARES. *Relevant Logic: A Philosophical Interpretation*. Cambridge University Press, 2004. [Cited on page 50]

[10] RICHARD MONTAGUE. "Universal grammar". *Theoria*, 36(3):373–398, 1970. [Cited on page 142]

[11] GREG RESTALL. "Negation in Relevant Logics: How I Stopped Worrying and Learned to Love the Routley Star". In DOV GABBAY AND HEINRICH WANSING, editors, *What is Negation?*, volume 13 of *Applied Logic Series*, pages 53–76. Kluwer Academic Publishers, 1999. [Cited on page 50]

[12] GREG RESTALL. *Logic*. Fundamentals of Philosophy. Routledge, 2006. [Cited on page 4]

[13] R. M. SMULLYAN. *First-Order Logic*. Springer-Verlag, Berlin, 1968. Reprinted by Dover Press, 1995. [Cited on page 66]

[14] TIMOTHY WILLIAMSON. *Vagueness*. Routledge, London; New York, 1994. [Cited on page 138]

[15] TIMOTHY WILLIAMSON. *Knowledge and Its Limits*. Oxford University Press, Oxford, 2002. [Cited on page 138]

IMAGE ACKNOWLEDGEMENTS

George Boole in Colour, *artist unknown* (c. 1860)
[PUBLIC DOMAIN], via *Wikimedia Commons*.
http://commons.wikimedia.org/wiki/File:George_Boole_color.jpg

Charles Sanders Peirce by Author Unknown (c. 1900)
[PUBLIC DOMAIN], via *Wikimedia Commons*.
http://en.wikipedia.org/wiki/Charles_Sanders_Peirce.

Claude Shannon with his electromechanical mouse "Theseus".
Reprinted with permission of Alcatel-Lucent USA Inc.
http://www.landley.net/history/mirror/pre/shannon.html.

Jan Łukasiewicz (1935), *photographer unknown* [PUBLIC DOMAIN].
http://www.calculemus.org/MathUniversalis/6/lukas.jpg

Noam Chomsky by *Andrew Rusk* [CC-BY-2.0]
http://www.flickr.com/photos/andrewrusk/5598986979/

Barbara Partee [PUBLIC DOMAIN], via *Wikimedia Commons*
http://commons.wikimedia.org/wiki/File:Barbara_partee.jpg

Addo Elephant National Park by *Brian Snelson* [CC-BY-2.0]
http://www.flickr.com/photos/32659528@N00/429421469

Toy elephant snip by *sammydavis dog* [CC-BY-2.0]
http://www.flickr.com/photos/25559122@N06/6057895435/

Our cat?? by *irrational cat* [CC-BY-SA-2.0]
http://www.flickr.com/photos/irrational_cat/35776501

Zac and Greg in Arizona (January 2009) by *Christine Parker*, reproduced with permission.

Wilson Tennis Racquet by *Keven Payravi* [CC-BY-SA-3.0]
http://commons.wikimedia.org/wiki/File:Wilson_Tennis_Racquet.jpg

Laptop by mystica [PUBLIC DOMAIN]
http://openclipart.org/detail/15413/laptop-by-mystica

Thomas Hobbes (1588-1679), English philosopher, artist unknown, [PUBLIC DOMAIN] via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Thomas_Hobbes.jpeg

Portrait of Gottfried Leibniz (1646-1716), German philosopher, artist Christoph Bernhard Francke, circa 1700. Herzog-Anton-Ulrich-Museum, Braunschweig, Germany. [PUBLIC DOMAIN] via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Gottfried_Wilhelm_von_Leibniz.jpg

Portrait of David Hilbert (1862-1943), German mathematician, artist unknown [CC BY SA 2.0]; Mathematisches Forschungsinstitut Oberwolfach, Germany: archives of P. Roquette, Heidelberg.
http://owpdb.mfo.de/detail?photo_id=9240

Photograph of The Stacked Slate Sculpture of Alan Turing by Stephen Kettle, Photograph by Jon Callas from San Jose, USA, [CC BY 2.0], http://commons.wikimedia.org/wiki/File:Alan_Turing.jpg via Wikimedia Commons