

LOGIC

Language & Information

VOLUME 2

Jen Davoren and Greg Restall

The University of Melbourne

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

VERSION OF APRIL 13, 2015

© JEN DAVOREN AND GREG RESTALL

CONTENTS

Part I Core Techniques in Predicate Logic

1 *Language of Predicate Logic* | 11

- 1.1 Limitations of Propositional Logic · 11
- 1.2 Splitting the Atom · 14
- 1.3 Quantifiers and Variables · 19
- 1.4 Defining the Language · 24
- 1.5 Translating into Predicate Logic · 28
- 1.6 Instances of Quantified Formulas · 31

2 *Models for Predicate Logic* | 33

- 2.1 Interpreting a Language · 34
- 2.2 Defining Models · 37
- 2.3 Truth in a Model · 42
- 2.4 Finite and Infinite Models · 49
- 2.5 Classifying Formulas · 53
- 2.6 Relationships Between Formulas · 57
- 2.7 Validity of Arguments · 61

3 *Tree Proofs for Predicate Logic* | 67

- 3.1 Why We Need Proof Trees · 67
- 3.2 Tree Rules for Predicate Logic · 72
- 3.3 Development of Trees for Predicate Logic · 74
- 3.4 Simple Examples · 79
- 3.5 Complex Examples · 83
- 3.6 Soundness and Completeness · 89

4 *Identity and Functions* | 95

- 4.1 Identity: Expressive Power · 95
- 4.2 Identity: Syntax and Models · 97
- 4.3 Identity: Tree Proofs · 101
- 4.4 Counting · 104
- 4.5 Functions and Other Extensions · 109

Part II Applications of Predicate Logic

5 *Quantifiers in Mathematics* | 117

- 5.1 Mathematics and Logic · 118
- 5.2 The Real Number Line · 123

5.3	Distance Predicates on the Real Numbers	· 134
5.4	Sequences of Real Numbers	· 141
5.5	Convergence and Divergence of Sequences of Reals	· 144
6	<i>Predicate Logic Programming</i>	· 157
6.1	Predicate Logic, Computers, & Automated Reasoning	· 158
6.2	Fragment of Predicate Logic in PROLOG	· 166
6.3	Logic Programming in PROLOG	· 176
6.4	How PROLOG Answers Queries	· 188
7	<i>Definite Descriptions and Free Logic</i>	· 209
7.1	What <i>does</i> exist, and what <i>doesn't</i> ?	· 210
7.2	Russell's Analysis of Definite Descriptions	· 219
7.3	Free Logic	· 227
8	<i>Quantifiers in Linguistics</i>	· 235
8.1	Quantifier Meaning	· 236
8.2	Quantifiers and Pronouns	· 242
8.3	Quantifier Ambiguities	· 247
9	<i>Sequential Digital Systems</i>	· 253
9.1	Digital Signals and Systems	· 254
9.2	Time, Memory and Flip-Flops	· 267
9.3	Feedback and State Signals	· 276
9.4	Linear Temporal Logic	· 286
9.5	Formal Verification of Digital Systems	· 298

References · 305

INTRODUCTION

These are the notes for use in *Logic: Language and Computation 2*, an introduction to predicate logic and its applications, available on [Coursera](#) from April 2015. Like its precursor, *Logic: Language and Computation 1*, 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 great deal 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.

WHAT THIS COURSE IS ABOUT: Information is everywhere: it is in our words and in 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

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.

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 *computer science*, *electronic engineering*, *linguistics*, *mathematics* and *philosophy*.

The class is taught over a period of eight weeks. In the first four weeks, we learn *core predicate logic*.

- WEEK 1—*The Language of Predicate Logic.*
- WEEK 2—*Models for Predicate Logic.*
- WEEK 3—*Proof Trees for Predicate Logic.*
- WEEK 4—*Identity and Functions.*

Then in the second part of the subject, we cover different *applications of predicate logic*. In this subject, we *expect* students to take at least three. 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 areas in whatever order you like.

- COMPUTER SCIENCE—*databases, resolution and predicate Prolog.*
- ELECTRONIC ENGINEERING—*sequential digital systems.*
- LINGUISTICS—*quantifier expressions and meaning.*
- MATHEMATICS—*quantifiers and mathematical reasoning.*
- PHILOSOPHY—*definite descriptions and free logic.*

This subject presumes as background some familiarity with propositional logic. The *best* background you can have is competence in truth tables and tree proofs for propositional logic—in particular, the material we cover in the core section of *Logic: Language and Information 1*. In addition, for the sections in *computer science* and *electronic engineering*, we will presume a little background knowledge with respect to propositional Prolog and combinational systems respectively. We will explain the necessary background knowledge for these sections in their chapters.

Other than this, all you need is 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.

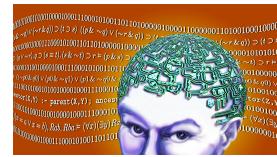
HOW TO USE THESE NOTES: These notes provide you with a written complement to the course videos and discussion boards available on Coursera. Most students will find it easiest to watch the videos, and then refer to the notes to follow up things in greater depth as necessary. But if you learn best by reading, 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* [8] 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* [5] 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 course's logo. We thank our colleagues 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 gave us advice and feedback on course design, 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,
2015

PART 1

Core Techniques in Predicate Logic

LANGUAGE OF PREDICATE 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.

- **BACKGROUND:** the propositional connectives—conjunction ($\&$), disjunction (\vee), conditional (\supset), biconditional (\equiv), negation (\sim).
- *Names, predicates, quantifiers, variables.*
- The *arity* (number of places) of a predicate. The *universal quantifier* ($\forall x$) and the *existential quantifier* ($\exists x$).
- *Well-formed formulas* in the language of predicate logic; and the way they are recursively defined from atomic formulas (using predicates and names).
- The *scope* of a quantifier in a formula.
- The *instances* of a quantified formula.

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 practise your skills and get familiar with each of these concepts.

SKILLS

These are the skills you will learn to demonstrate.

- Recognising whether an expression is a well-formed formula in the language of predicate logic.
- Translating *from* English (or another natural language) into the language of predicate logic.
- Learning how to read formulas in the language of predicate logic, and how to understand their meaning in an intuitive sense—at least in the case where the formula is relatively simple.
- Recognising the scope of a quantifier in a formula and determining the variables bound by a quantifier.
- Recognising whether a formula is an *instance* of a quantified formula.

1.1 | LIMITATIONS OF PROPOSITIONAL LOGIC

This is a course in first order predicate logic and its applications. Predicate logic goes beyond *propositional* logic, by adding predicates, names and quantifiers to the propositional connectives. We'll start our investigation in this chapter by looking at the language of predicate logic.

Our first step will be to revisit *propositional* logic, and to see why propositional logic is not the end of the story of logical structure and relationships between propositions.

1.1.1 | THE VOCABULARY OF PROPOSITIONAL LOGIC

Declarative sentences are the kinds of sentences we take to be true or false, we can agree with or disagree with, take as the premises or conclusions in reasoning. Not all sentences are declarative—we can do other things with our sentences, such as ask questions, make requests and issue commands.

These are not the only options for the connectives. You may also see \wedge for conjunction, \neg for negation, and \rightarrow for the conditional. More exotic forms of notation include Polish notation for formulas, which uses a *prefix* capital letter for the operators, so the infix notation $(p \ \& \ q) \supset (r \vee s)$ becomes $CKpqArs$. There are no parentheses needed in Polish notation, but many people find it not so easy to read.

Propositional logic is the study of *propositions*, the information carried by declarative sentences and operators or connectives that modify or combine propositions to form new propositions from old. The connectives in classical propositional logic are conjunction ('and'), disjunction ('or'), the conditional ('if ... then ...'), the biconditional ('if and only if') and negation ('not'). Here are the conventions we use for notation for the propositional connectives:

$$\begin{array}{ccccccc} \& \vee & \supset & \equiv & \sim \\ \text{and} & \text{or} & \text{if} & \text{iff} & \text{not} \end{array}$$

A formal propositional logic uses a *grammar* to precisely define a formal language. Here is the grammar we use.

DEFINITION: We define the set of **FORMULAS** of the *language of propositional logic* (given a choice of a collection of *atoms*) like this:

- (1) *Atoms* are *formulas*.
- (2) If A is a *formula*,
then so is $\neg A$.
- (3) If A and B are *formulas*,
then so are $(A \ \& \ B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$.
- (*) *Nothing else is a formula.*

The *atoms* in this definition are our starting points for formulas. Typically, we use letters like p , q , r and so on as atoms. These formulas allow us to study the structures of many different kinds of arguments, and apply logic to many different domains, as we have seen in *Logic: Language and Information 1*. However, propositional logic is not the end of the story.

1.1.2 | SOME VALID ARGUMENTS AREN'T VALID ACCORDING TO PROPOSITIONAL LOGIC



In the margin, you can find a picture of a *wombat*, a burrowing marsupial herbivore, native to south-eastern Australia and Tasmania. Suppose we tell you we know a particular wombat we call "Will." If you're curious as to how many legs Will the wombat has, you could ask us. However, if we weren't around to tell you the answer, you could reason like this:

All wombats have four legs.

Will is a wombat.

Therefore, Will has four legs.

You're using the information that you've independently heard—that all wombats have four legs. This seems like a *valid* piece of reasoning—it's not that the conclusion is certain, but *given* the premises, the conclusion follows. If it turns out that Will is a three legged wombat (having lost a leg in an injury) then it turns out that not all wombats have four legs, and the first premise is false. If it turns out, on the other hand, that Will is a wasp and not a wombat, then will could have six legs (the conclusion is also false) but now it is not the first premise that fails, it is the second—Will is not a wombat. However, if the premises are both true, the conclusion must follow: it is impossible for the premises to hold and for the conclusion to fail. It seems to satisfy our intuitive concept of *validity*.

Furthermore, the validity of the argument seems to be a matter of its *structure*. Other arguments with the same structure seem to be just as valid

All frogs are green.

Kermit is a frog.

Therefore, Kermit is green.

All cities are crowded.

Paris is a city.

Therefore, Paris is crowded.

and each of these arguments seem to be valid because of the form they share.

However, propositional logic is blind to that structure. The only thing we can say about the argument, using the language of propositional logic, is that it has the form $p, q, \text{ therefore } r$.

All wombats have four legs. p

Will is a wombat. q

Therefore, Will has four legs. $\text{Therefore}, r$

The second premise of the argument “Will is a wombat” is an atomic proposition. It is not a conjunction, disjunction, negation, conditional or biconditional. In this form, we formalise it with the atom q . The conclusion, “Will has four legs” is also an atomic proposition—it also is not a conjunction, a disjunction, a negation, a conditional or a biconditional. It’s also not the same as the second premise, so we formalise the conclusion with another atom, r . The premise, “all wombats have four legs” is certainly not a conjunction, a disjunction, a negation, or a biconditional. It’s not a conditional, either. Regardless, even if it *were* a conditional, it is not the conditional $q \supset r$, because it doesn’t mention Will—it doesn’t *just* say that if Will is a wombat, Will has four legs. The sentence “all wombats have four legs” certainly has logical structure, but as far as propositional logic can see, there is no structure at all. Its validity cannot be explained in terms of propositions and their combinations.

The same goes for arguments we can see to be *invalid*. This argument:

You might think it means “if something is a wombat, it has four legs.” This is a good suggestion, but it cannot be right, at least in propositional logic. If it were a conditional, the consequent couldn’t be “it has four legs” (this has no meaning out of that context). What could the antecedent be? “Something has four legs”? The conditional “if something is a wombat, something has four legs” has a different meaning.

Some wombats are hairy-nosed.
Will is a wombat.
Therefore, Will is hairy-nosed.

is invalid, and we can explain its invalidity straightforwardly. (Maybe Will isn't one of the hairy-nosed wombats. All the first premise says is that *some* of the wombats was hairy-nosed.) Again, the invalidity is indicated by its structure. Arguments involving other concepts, but with the same form, seem equally invalid.

Some frogs are green.	Some cities are crowded.
Kermit is a frog.	Paris is a city.
<i>Therefore</i> , Kermit is green.	<i>Therefore</i> , Paris is crowded.

Our job with *predicate* logic is to explain the validity and invalidity of arguments like these, isolating what is important in the logical *structure* of these propositions, and the role they play in arguments involving these propositions.

1.2 | SPLITTING THE ATOM

To isolate that structure, we need to look inside atomic propositions, to isolate structural features there. The validity of the argument

All wombats have four legs.
Will is a wombat.
Therefore, Will has four legs.

has something to do with the shared structure between the premises and the conclusion. The name “Will” occurs in the second premise and in the conclusion. If the name had changed, and the conclusion were “Wendy has four legs”, the argument would be invalid. Similarly, the categorisation “wombat” is shared between the two premises. If the second premise had said “Will is a wasp” again, the argument would have been invalid. This categorisation or description is shared between the two premises, and this, too, is important for the validity of the argument. Finally, the word “all” in the first premise plays a vital role. If it had been replaced by “some” or even “most” the argument would not have been valid. All these words or concepts play an important role in the argument. We need some way to make that structure explicit, and work with it. Because this structure is present even *inside* atomic propositions, we'll think of this as “splitting the atom.” The atomic proposition is not featureless. It has internal structure of its own, which plays a role in logical analysis. The first feature inside atomic propositions we will consider is the *name*.

1.2.1 | NAMES

The name “Will” featured in our argument. Its role was to name a particular animal. Names are an important grammatical feature in

many languages—they pick out objects, whether people (such as *Aristotle* or *Ada Lovelace*), animals (such as *Will the Wombat*), or other items, whether cities (such as *Melbourne*), abstract objects like numbers (such as *Five*) or even mythical creatures (like *Pegasus*).

A name is used to refer to an item, to bring it to mind so we can talk about it, describe it, inform each other about it or argue about it. Names are one way our propositions can reach out and connect with particular parts of the world around us.

It will be important, later, to distinguish names from other words or phrases we use to pick out an object. A phrase like “The Prime Minister of Australia” is also used to pick out an individual person, and to select it for categorisation (such as when we say “the Prime Minister of Australia is a member of the Liberal Party”). However, a phrase like this does that job by way of *describing* the item (the Australian PM is currently Tony Abbot, but previously, Kevin Rudd was the PM, and before him, Julia Gillard) rather than *naming* him. (We will see more of the logical behaviour of descriptions in Chapter 7.) A name differs from a description by having no descriptive role at all.

In the formal language of first order predicate logic, we will use the letters ‘a’, ‘b’, ‘c’... from the first part of the alphabet as **names**.



Aristotle



Ada Lovelace



Will (the Wombat)

DEFINITION: A *name* is a word (or string of words) used to pick out an individual item, not by way of *describing* it, but by conventionally designating it. In the language of first order predicate logic, we will use lower case letters from the beginning of the alphabet (a, b, c, etc.) for names.



Melbourne

1.2.2 | PREDICATES

A language which only has names cannot form propositions. If we can only name things, we cannot form judgements. But if we can *classify*, *categorise* or *describe* things, we have the wherewithal to form judgements, to express propositions. When we say

Will is a quadruped.

the words “is a quadruped” function to *describe* or to *classify* Will. We are saying that Will is a four-legged creature. Now we have a proposition, a judgement, the kind of thing that can be either *true* or *false*. We call these forms of words which describe or classify *predicates*. Predicates take a number of forms. Some predicates, like “... is a quadruped”, “... is a wombat”, “... is a multiple of ten”, “... runs” or “... is happy” take a single name to form a proposition. We call these “one-place” or “unary” predicates.

Not all predicates are one-place. Some predicates need two names to form a judgement, like “... likes ...”, “... is a multiple of ...”, “... is a sister of ...” or “... is next to ...” require two names to make a

Notice that some predicates, like “... is a wombat” describe constant features of things, while others are transitory, like “... runs” or “... is happy.” In either case, we have a predicate.

judgement. These are called “two-place” or “binary” predicates. In this case, the predicate *relates* two items. This relation often takes the *order* in which the two items are presented into account. 10 is a multiple of 5, while 5 is not a multiple of 10. Greg’s cat likes birds, but birds do not like Greg’s cat.

Predicates can have even higher arities than one or two. When we say that 5 is between 4 and 6, the “between” predicate relates the *three* items, 5, 4 and 6. This is a *ternary*, or three-place predicate.

In the formal language of first-order predicate logic, we will use upper case letters like ‘F’, ‘G’ as **predicates**.

DEFINITION: A *predicate* is a word or phrase used to classify or describe or relate objects. In the language of first order predicate logic, we will use upper case letters (F, G, H, etc.) for predicates. In each case, we will assume that each predicate comes with a fixed *arity*, the number of names it takes to make a sentence.

This will make our language rather more precise and less flexible than English and other natural languages. We use two place predicates flexibly. It makes sense to say that Wanda loves Will, and also to say that Wanda loves. We cannot say this in quite that way in the language of first order predicate logic—but as we will see, we will have a way of expressing this.

The final condition in this definition is important. If we use the letter F as a one place predicate in our language, then we will not *also* use that letter in the place of a two place predicate. If R serves as a two place predicate (a binary relation) then we will not also use it as a predicate of any other arity.

FOR YOU: Some of these items are *names*, and some are *predicates*, and some are something neither. Which of these items are the *predicates*? (a) hates (b) deliberately (c) is glad (d) Angela Merkel (e) the professor

ANSWER: (a) predicate. It can be used as a one-place predicate (“Will hates”) or a two-place predicate (“Will hates hot weather”). (b) This isn’t a predicate. It’s an adjective—a predicate modifier (comparable “Will chose” to “Will chose deliberately”). (c) This is a one-place predicate (“Will is glad”). (d) This is a name. (e) This is not a name, and not a predicate. You can say “Will is the professor”, but this is not a predication. “Will is a professor” is clearly a predication, but “Will is the professor”, like “the professor is Will” is identifying a name and a description as both picking out the one person.

1.2.3 | ATOMIC PROPOSITIONS

Given these two grammatical categories, we can construct propositions.

A *predicate*, combined with the appropriate number of *names*, makes an *atomic proposition*

- Predicates must be paired with the appropriate number of names:
 - ★ ‘...is a quadruped’ requires one (and only one).
 - ★ ‘...loves...’ requires exactly two—the *lover* and the *beloved*.

In the formal language of first order predicate logic, we write the predicate first, followed by the names, in succession (without brackets or other punctuation). So, if F and G are one place predicates and L is a two place predicate, and a, b and c are names, then these are all different atomic propositions.

Fa Gb Lab Lba Laa Lca

We can construct *very* many atomic propositions in this vocabulary.

How many, exactly? $(2 \times 3) + (1 \times 3 \times 3) = 15$. Can you see how this number was calculated?

DEFINITION: In the language of first order predicate logic, an *atomic formula* is given by combining a predicate of arity n with n names. So if F is an n-place predicate and a_1, \dots, a_n are names (not necessarily all distinct), then $Fa_1 \dots a_n$ is an atomic predicate. Nothing else is an atomic formula.

Given the vocabulary of atomic formulas, constructed by way of names and predicates, we can then form simple propositions, and we can combine them in the usual way, with the standard propositional connectives.

SENTENCE	FORMULA
Clancy is forgetful	Fc
Clancy is forgetful but Madison isn’t	Fc & ~Fm
Clancy looks after Madison	Lcm
Clancy and Madison look after other	Lcm & Lmc
Clancy and Madison look after themselves	Lcc & Lmm
Clancy looks after both Madison and Lee	Lcm & Lcl
Clancy looks after Madison but not Lee	Lcm & ~Lcl
If Clancy looks after Madison she looks after Lee	Lcm \supset Lcl
Either Clancy or Madison looks after Lee	Lcl \vee Lml

What about atomic formulas that don’t include any names at all? Can we have atomic formulas that are unstructured statements like the ps and qs of propositional logic? Yes, we can. We allow a predicate to take *any* whole number as an arity, including zero. A zero place predicate requires no names to make an atomic formula. These do the job of atomic propositions featuring no names.

Notice that the formulas for these sentences sometimes diverge from the structure of the sentences themselves—in the language of first order predicate logic, the propositional connectives like \wedge and \vee connect sentences, not names. When we say that Clancy looks after Madison and Lee, it is *not* understood as a special ‘conjunctive person’ m & l,

What could such a conjunctive object m & l be? What about a disjunctive object m \vee l?

which is looked after by Clancy, rather, this is formalised as the conjunction of the atomic propositions Lcm and Lcl . This seems appropriate here, for to say that Clancy looks after Madison and Lee is to say nothing more and nothing less than that Clancy looks after Madison and Clancy looks after Lee. Since this paraphrase preserves the meaning of the expression, it seems appropriate to formalise the sentences in this way.

FOR YOU: Given the dictionary, $Sxy = x$ is shorter than y ; $c = \text{Clancy}$; $m = \text{Madison}$; $l = \text{Lee}$, which of the following formulas is the best translation for Clancy is shorter than Madison but not Lee?

- (a)** Scm & Sc~l **(b)** cS(m & ~l) **(c)** Scm & ~ScI

Which formula is the best translation for Clancy is shorter than neither Madison nor Lee?

- $$\text{(a)} \ cS \sim (m \vee l) \quad \text{(b)} \ \sim(Scm \vee Scl) \quad \text{(c)} \ \sim Scm \vee \sim Scl$$

nor B is to say $\sim(A \vee B)$.

For the same reasons, Clancy is shorter than neither Madison nor Lee is best translated as (b) $\sim(S \text{cm} \vee S \text{cl})$. (To say that neither A

is shorter than that.

is shorter than Madison but not Lee—we do not say that there is an object such as “Madison and not Lee”, which is such that Clancy

Scal is an appropriate way to say that Clancy is not shorter than Lee, and *CS(m ~l)* is not an appropriate way to say that Clancy

$\sim SCl$). It is the conjunction of SCm ($Clancy$ is shorter than $Madison$) and $\sim SCl$ (It 's not the case that $Clancy$ is shorter than Lee).

ANSWER: Clancy is shorter than Madison but not Lee is (c) (Scm &

That is the structure of atomic formulas in the language of first order predicate logic. It provides insight into the structure of atomic propositions and their propositional combinations. In our simple argument about Will the wombat, we can now see some structure we couldn't see before. If F and W are one place predicates (for *having Four legs* and *being a Wombat* respectively), and if we take w to formalise the name *Will*, then we have at least this much when it comes to the form of our argument:

All wombats have four legs.

222

Will is a wombat.

Ww

Therefore, Will has four legs.

Therefore, Fw

We can see the shared structure between the second premise and the conclusion. We can't yet see the structure of the first premise "All

wombats have four legs,” but at the very least we can see that whatever form it has, it *does* involve the predicates F and W , and it *doesn't* involve the name w . How are we to understand the logical form of “All wombats have four legs”? That is our topic for the next section.

1.3 | QUANTIFIERS AND VARIABLES

We've already seen examples of how ‘all’ and ‘some’ play particular structural roles in reasoning with propositions involving predicates and names. Instead of starting with “all” (as appears in our example valid argument about Will the wombat), we'll start with concepts like “some” or “someone” because this seems closest to the behaviour of names.

1.3.1 | DO ‘SOMEONE’ OR ‘SOMETHING’ FUNCTION LIKE NAMES?

If we want to formalise “Will is a quadruped” and “Clancy looks after Madison” like this

- Will is a quadruped: Qw
- Clancy looks after Madison: Lcm

then it is tempting to attempt to formalise “Something is a quadruped” and “Clancy looks after someone” in a similar way:

- Something is a quadruped: — $Qs?$
- Clancy looks after someone: — $Lcs?$

We might try to take “something” or “someone” to act like a name, so we might try to formalise it with something that goes in the same position as a name—perhaps we could use special *underlined* name, like \underline{s} . However, this won't work, as we'll see when we look at what logical complexity adds to the picture. Consider how we might say “something is not a quadruped” as opposed to saying “*nothing* is a quadruped.”

- Something is not a quadruped: — $\sim Q\underline{s}?$
- Nothing is a quadruped: — $\sim Qs?$

These both have a good claim to have the form $\sim Q\underline{s}$. Something is *not* a quadruped is saying of something that it has the property of *not* being a quadruped. We're saying $\sim Q$ holds of \underline{s} , in just the same way that we've said that Q holds of \underline{s} when we say that something *is* a quadruped. On the other hand, when I say that nothing is a quadruped, I'm denying the claim that *something* is a quadruped. I'm taking the claim $Q\underline{s}$ and negating it. I'm saying $\sim Q\underline{s}$. But “something isn't a quadruped” and “nothing is a quadruped” say very different things. (For a start, the first is true and the second is false.) So, they shouldn't be formalised in the same way. We must have made a mistake somewhere. Our diagnosis is that we should not treat “something” like a name—it doesn't act like one.

Before explaining what we will do instead, let's look at another example of where the treatment of “something” or “someone” as acting like a name gets things wrong, this time involving conjunction and not negation. Suppose I say “Clancy looks after someone and Madison looks after someone.” If “Clancy looks after someone” is L_{Cs} and “Madison looks after someone” is L_{Ms} , then their conjunction is $L_{Cs} \& L_{Ms}$.

If I wanted to say “there's someone looked after by both Clancy and Madison” I'd also say $L_{Cs} \& L_{Ms}$. But this says something more than just saying that Clancy looks after someone and Madison looks after someone. Again, these two sentences should not have the same form. There should be some way to explain the difference in meaning.

That's the job of the *quantifier* in the language of first order predicate logic. Whatever we are doing when we use words like “some” or “all,” they don't just do the work of a name, to pick out a single thing. Quantifiers like these modify or combine predicates to make sentences without using names—and as a result, they something quite subtle and interesting.

“All frogs are sticky” or “some students are smart” combine the predicates “is a frog,” “is sticky,” “is a student” and “is smart” and the quantifiers “all” and “some” and nothing else.

1.3.2 | EXAMPLES OF QUANTIFIERS

Quantifier expressions in English aren't restricted to “someone” and “something”. There are many different examples of quantifier expressions.

- *Some* wombats are hairy-nosed.
- *All* valid arguments with true premises have a true conclusion.
- *Most* logic students are intelligent.
- *Many* Australians live in poverty.
- *At least seven* people have been on the moon.
- *Typical* philosophers are well read, but impractical.

The *quantifiers* in these sentences are ‘*some*’, ‘*all*’, ‘*most*’, ‘*many*’, ‘*at least seven*’ and ‘*typical*.’ These sentences are, in some sense, about ‘*some wombats*’, or ‘*all valid arguments*’, ‘*most logic students*’, ‘*many Australians*’, ‘*at least seven people*’ or ‘*typical philosophers*’. But in every case, the quantifier expressions do not pick out an entity and *name* it. The expression does something else: it selects a *range* of entities satisfying some condition.

For more, Stanley Peters and Dag Westerståhl's *Quantifiers in Language and Logic* is a wonderful resource [7].

There are very interesting things one could say about the logical features of all of these quantifiers (and more) but in this subject we will focus on just two quantifiers—‘*some*’ and ‘*all*’—because the logical behaviour of these quantifiers is *relatively* straightforward, and they are incredibly powerful, in that very many important and interesting logical properties can be explained in terms of the behaviour of these quantifiers.

There have been different theories about how the quantifiers ‘*some*’ and ‘*all*’ behave, throughout the history of logic. The dominant tradi-

tion up until the 20th Century was from the work of the Ancient Greek philosopher Aristotle. For Aristotle, all basic judgements involve the linking of *terms*. So, for example, when I say

- All wombats are quadrupeds.
- Some wombats are hairy-nosed.
- No wombats are logic students.

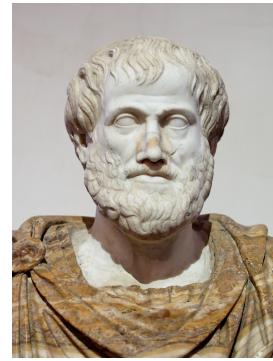
in each case we have linked a subject term S with a predicate term P, and we have combined them either directly a quantifier like ‘All’ or ‘Some’ or ‘No.’ The subject of the judgement is ‘all wombats,’ ‘some wombats’ or ‘no wombats,’ respectively. Logical syllogism link these forms of judgement. So, if I say ‘some wombats are hairy nosed’ then ‘some wombats’ is the subject and ‘are hairy nosed’ is the predicate. If the subject satisfies the predicate, the judgement is true, and if not, the judgement is false. This works, as far as it goes, but it does not handle complex quantifier expressions at all well. Instead of following Aristotle, modern logic follows the analysis of the German philosopher and mathematician, Gottlob Frege.

According to Frege, quantification involves a *choice* or a *selection* of an object or a range of objects, satisfying certain criteria. When we say “some wombats are hairy-nosed” this statement is true if and only if it is possible to make a choice of some wombats, which satisfy the condition of being hairy-nosed. When we say “some wombats aren’t hairy-nosed,” then, *this* statement is true if and only if it is possible to make a choice of some wombats, which don’t satisfy the condition of being hairy-nosed. This is different from saying “no wombats are hairy-nosed” which is the negation of “some wombats are hairy-nosed,” and this is true if and only if it is *not* possible to make a choice of some wombats which satisfy the condition of being hairy-nosed. This gives us two different places in a ‘some’ statement at which a negation can occur. In “some wombats are not hairy-nosed”, the choice occurs first, and the negation is inside the scope of that choice. The statement says that a choice can be made for the negative criterion—a non-hairy-nosed wombat. In “no wombats are hairy-nosed,” the negation occurs first, and the choice occurs inside the scope of that negation—in effect, the negation denies the claim that the choice of a hairy-nosed wombat can be made. The attempted formalisation involving something like a name (in $\sim H_s$, where s stands for some wombats) gave us no point at which to draw this distinction.

So, if we are going to say that Will is a quadruped, and is hairy-nosed, we formalise this as

$$Qw \ \& \ Hw$$

and if I want to make the less specific claim, not that we have one object in mind that is a hairy-nosed quadruped, but merely that *some* quadruped is hairy-nosed, we’ll formalise this by saying we can make *some* choice for an object (let’s use the *variable* ‘x’ for the object, since we don’t have a particular object in view), such that $Qx \ \& \ Hx$ holds.



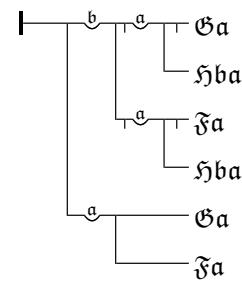
Aristotle (384 BCE – 322 BCE)

But be careful: what is the subject when we say (and say *truly*) that some wombats are hairy-nosed and some wombats aren’t hairy nosed. Is it the same subject in both conjuncts of that conjunction?



Gottlob Frege (1848–1925)

Frege had a special notation for what he called a ‘concept script’ (Begriffschrift). Be glad that we’re not using Frege’s notation.



We use the back-to-front E, the *existential quantifier* to mark that we can make *some* choice for x for which $Qx \& Hx$ is true. This is written

$$(\exists x)(Qx \& Hx)$$

So, to say that *no* quadrupeds are hairy-nosed, we deny this—we say that there is no choice to be made of x to satisfy $Qx \& Hx$. This is written

$$\sim(\exists x)(Qx \& Hx)$$

If we wanted to say, instead, that some quadruped is not hairy-nosed, we'd write the negation inside the scope of the quantifier—in particular, we'd say that there is a choice to make for x such that $Qx \& \sim Hx$ holds—there is a quadruped that is *not* hairy-nosed:

$$(\exists x)(Qx \& \sim Hx)$$

Furthermore, when we say that something *isn't* a hairy-nosed quadruped, we're saying that we can make a choice, not of a quadruped that isn't hairy-nosed, but merely of something that isn't a hairy-nosed quadruped—an x such that $\sim(Qx \& Hx)$.

$$(\exists x)\sim(Qx \& Hx)$$

The existential quantifier works in the presence of two place predicates, too. If we wanted to say that Clancy looks after something, we'd say

$$(\exists x)Lcx$$

since there is some choice we can make for x such that Lcx holds. So, to say that Clancy looks after something and Madison looks after something, we can conjoin two statements of this form:

$$(\exists x)Lcx \& (\exists x)Lmx$$

There is a choice that can be made of an object that Clancy looks after, and a choice for an object looked after by Madison. Two choices, two quantifiers. On the other hand, if we wanted to say that there's something that's looked after by both Clancy and Madison, we'd use one quantifier to make the choice:

$$(\exists x)(Lcx \& Lmx)$$

since there is some object we could choose for x that satisfies $Lcx \& Lmx$ —whatever there is that both Clancy and Madison look after.

That is the behaviour of the existential quantifier, which we use for expressions like ‘some wombat’ or ‘something’. The same kind of notation will be useful for expressions like ‘all wombats’ or ‘everything’. We want, in particular, to understand the structure and behaviour of expressions like “all wombats are quadrupeds.” One upshot of a statement like this is, at the very least, that if Will is a wombat, then Will is a

quadruped. So whatever “all wombats are quadrupeds” says, it should tell us

$$Ww \supset Qw$$

(where W is the predicate “is a wombat” and Q is the predicate “is a quadruped”) for the object w . But “all wombats are quadrupeds” should not just tell us this about *Will*. It should also say this about any other wombat we care to mention. In fact, it tells us this about anything at all, for if all wombats are quadrupeds, then given any thing x at all, *if* it’s a wombat, it’s a quadruped. This is how Frege understands the structure of “al wombats are quadrupeds”. It involves a *universal* quantifier, written with an upside down A:

$$(\forall x)(Wx \supset Qx)$$

which says for *whatever* choice we make for x , the condition $Wx \supset Qx$ holds true. It is to be read as saying for *every* object x , if Wx then Qx . So, we could formalise “all wombats *aren’t* hairy-nosed” as

$$(\forall x)(Wx \supset \sim Hx)$$

while “not all wombats are hairy-nosed” would be

$$\sim(\forall x)(Wx \supset Hx)$$

You might notice that it’s tempting to read “all wombats aren’t hairy-nosed” as “no wombats are hairy-nosed” which would be formalised as

$$\sim(\exists x)(Wx \& Hx)$$

So, it would be a good thing if we were able to show that the two statements $\sim(\exists x)(Wx \& Hx)$ and $(\forall x)(Wx \supset \sim Hx)$ are logically equivalent. Simiarly, you might hope that “not all wombats are hairy-nosed” ($\sim(\forall x)(Wx \supset Hx)$) is logically equivalent to “some wombats aren’t hairy-nosed”, whih should come out as

$$(\exists x)(Wx \& \sim Hx)$$

and indeed, this will turn out to be the case. There are a number of logical connections between the existential and universal quantifiers, and these can be made precise when we understand what it takes to interpret these quantifiers. But before then, we’ll need get a grip on what kinds of sentences can be formalisd in the language of first-order predicate logic. In particular, you can do a lot when you combine the quantifiers. “Will has a friend” is formalised as

$$(\exists x)Fxw$$

where Fab is to be read as “a is a friend of b” so Fxw says that x is a friend of Will. So how would we formalise “all wombats have friends”? The “all wombats” is to be formalised using a universal quantifier, like $(\forall x)(Wx \supset \dots)$, but we can’t use the variable x in this case, because when we say that all wombats have friends, we’re making two

Notice that “some Fs are Gs” is $(\exists x)(Fx \& Gx)$ while “all Fs are Gs” is $(\forall x)(Fx \supset Gx)$. It is worth your while spending time thinking about what $(\exists x)(Fx \supset Gx)$ and $(\forall x)(Fx \& Gx)$ mean, and why we don’t use sentences of these forms to formalise everyday expressions anywhere near as much as we use $(\exists x)(Fx \& Gx)$ and $(\forall x)(Fx \supset Gx)$.

And indeed, when it comes to the next chapter, we’ll be able to show this very thing. But we can only do this when we have a precise account of what it would take for these formulas to be true, since only then can we show that they’ll be true in exactly the same circumstances.

choices—first, the general choice of each wombat, and then for each wombat chosen, we can make a choice of a friend of that wombat. If we use the variable x for the choice of the friend, we'll use a different variable for the choice of the wombat. Let's use y . If $(\exists x)F_{xy}$ says that something is a friend of Will, and we want to generalise this to all wombats, we'll say

$$(\forall y)(Wy \supset (\exists x)F_{xy})$$

since we're saying that any choice we can make for y is such that if it's a wombat, then there is some friend of y . In this case, we make two choices, and the order of the quantifiers in the expression makes this clear: first we choose y . Our condition holds for *any* choice of y : if it's a wombat, then it has a friend. To say that y has a friend, we say that *some* choice can be made for x such that F_{xy} holds. The order of the two quantifiers respects the order in which the choices are made. If we look at a similar formula, in which the quantifiers are in different positions—where the existential quantifier is now *outside* the universal quantifier—

$$(\exists x)(\forall y)(Wy \supset F_{xy})$$

we have a very different expression, which says that there is some thing (the x) that is a friend of all wombats (the ys), for we can choose a candidate for x such that for any choice of y , if y is a wombat, then x is a friend of y .

These examples should give you some sense of the power of the language of first-order predicate logic. In the next section, we'll precisely define the that language, to make precise the languag we have been using.

1.4 | DEFINING THE LANGUAGE

First, let's remind ourselves of what we've already seen, with predicates, names and connectives.

DEFINITION: Quantifier-free formulas in the language of first-order predicate logic are defined as follows:

- If F is a **PREDICATE_n** (of arity n) and a_1, \dots, a_n are **NAMES** then $Fa_1 \dots a_n$ is a **FORMULA**.
- If A is a **FORMULA**, then $\sim A$ is a **FORMULA**.
- If A and B are **FORMULAS**, then $(A \& B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$ are also **FORMULAS**.

To add quantifiers to this picture, we need to be a little careful. In an expression like $(\exists x)(Hx \& Qx)$, the quantifier $(\exists x)$ has been added to the conjunction $Hx \& Qx$, but in this case, the conjunction $Hx \& Qx$ *isn't a formula*. Formulas without quantifiers only contain predicates,

It isn't a formula for good reason: what does $Hx \& Qx$ say? It doesn't say anything by itself. It says something only when we are told what x is. But to give x a particular interpretation is to treat it as a name, not as a variable. We use variables to be governed by quantifiers.

names and connectives. Variables are only introduced when we have quantifiers. Quantifiers will attach, not to formulas, but to *complex predicates*. $Hx \& Qx$ isn't something which says anything true or false, but it says something which is true or false of *something*. (Just like H it can be understood as a predicate, which is true when we choose an object that satisfies both H and Q , and is false for any other choice for x .) Complex predicates are found by taking formulas and replacing a name in that formula by a variable. Here is the definition, and the notation we will use for that kind of replacement:

DEFINITION: A COMPLEX PREDICATE is defined by replacing a name in a formula by a variable.

- If A is a **FORMULA**, a is a **NAME**, and x is **VARIABLE** then $A[a := x]$ is the result of replacing all occurrences of a in A by the variable x .

The complex predicate $A[a := x]$ is not necessarily a formula by itself, but it can be a part of one, when the variable x is governed by a quantifier.

EXAMPLE COMPLEX PREDICATES:

- $(Wa \supset Fab)[a := x]$ is $(Wx \supset Fxb)$.
- $(\forall x)(Wx \supset Fxb)[b := y]$ is $(\forall x)(Wx \supset Fxy)$.

Now we can expand our definition of the class of formulas in the language of first-order predicate logic to include quantified formulas.

Notice that there is *no* restriction here demanding that the name a appear in the formula A . By these lights, $(\exists x)Fb$ is a formula—even though x does not appear in Fb —since Fb is $Fb[a := x]$. Strange, but true. It's easier to leave these formulas, with what we call ‘vacuous quantification’, in the language than keep them out, and they do no harm, even if they are a bit strange. (What difference is there between $(\exists x)Fb$ and $(\exists x)(Fb \& (Gx \vee \neg Gx))$?)

DEFINITION: The formulas in the language of first-order predicate logic are defined as follows:

- If F is a **PREDICATE_n** (of arity n) and a_1, \dots, a_n are **NAMES** then $Fa_1 \dots a_n$ is a **FORMULA**.
- If A is a **FORMULA**, then $\sim A$ is a **FORMULA**.
- If A and B are **FORMULAS**, then $(A \& B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$ are also **FORMULAS**.
- If A is a **FORMULA**, if a is a **NAME**, and if x is **VARIABLE** then $(\forall x)A[a := x]$ and $(\exists x)A[a := x]$ are also **FORMULAS**.

Here are some examples of how the rules generate formulas.

Gnerating formulas from the inside out.

- Fa is a FORMULA if F is a PREDICATE₁ and a is a NAME.
- Gba is a FORMULA if G is a PREDICATE₂ and b is also a NAME.
- $Fa \& Gba$ is also a FORMULA.
- $(Fa \& Gba)[a := x]$ is $Fx \& Gbx$.
- $(\exists x)(Fx \& Gbx)$ is a FORMULA.
- $(\exists x)(Fx \& Gbx)[b := y]$ is $(\exists x)(Fx \& Gyx)$.
- So, $(\forall y)(\exists x)(Fx \& Gyx)$ is a FORMULA.

This shows how the complex formula $(\forall y)(\exists x)(Fx \& Gyx)$ can be generated from atomic formulas by a series of steps. We could achieve the same result by *decomposing* a formula, from the outside in.

$(\forall x)(Fx \vee Gx) \supset ((\forall x)Fx \vee (\exists x)Gx)$ is a FORMULA,

- since $(\forall x)(Fx \vee Gx)$ is a FORMULA,
 - Since $Fa \vee Ga$ is a FORMULA
 - ★ because Fa is a FORMULA
 - ★ and Ga is a FORMULA
 - and $(Fx \vee Gx)$ is $(Fa \vee Ga)[a := x]$.
- and $(\forall x)Fx \vee (\exists x)Gx$ is a FORMULA.
 - Since $(\forall x)Fx$ is a FORMULA,
 - ★ because Fa is a FORMULA
 - ★ and Fx is $Fa[a := x]$
 - and $(\exists x)Gx$ is a FORMULA,
 - ★ because Ga is a FORMULA
 - ★ and Gx is $Ga[a := x]$

Now it's time for you to check your understanding:

FOR YOU: Which of the following are formulas, according to our rules? (Assume that a, b, c are names, x and y are variables, F is a unary predicate, and L and R are binary predicates.)

- (a) $Fa \vee (\exists x)(Fx \& \sim Rab)$
- (b) $(\forall x)x F \supset (\sim Fa \& Rab)$
- (c) $\sim(\forall x)(Rab \vee \sim(Fx \& Rxc))$

- (d) $(\forall y)(Fa \supset Lxa)$
 (e) $(\forall x)(Fx \supset Rxx) \vee Rx a$
 (f) $(\exists x)(Fx \& (\exists x)Lax)$

equivalent to—and clearer than—our original formula. Different variables: $(\exists x)(Fx \& (\exists y)Lay)$ will turn out to be logically the x in Lax . It could have been made less confusing by using different $(\exists x)$ in the formula governs the x in Fx , while the second governs the first $(\exists x)$. As a result, $(\exists x)(Fx \& (\exists x)Lax)$ is well-formed. The first (ungoverned by a quantifier) while the x in Lax is still governed by is a complex predicate: $Fx \& (\exists x)Lax$. Here, the x in Fx is free for example, $Fb \& (\exists x)Lax$ is a formula, so $(Fb \& (\exists x)Lax)[b = x]$ formula, and this quantifier $(\exists x)$ binds the variable x in Lax . Then, (f) is a formula, but it is potentially confusing: $(\exists x)Lax$ is a for-

$(\forall x)$ quantifier binds $(Fx \supset Rxx)$ but does not touch Rxa . ANSWER: (a), (c) and (f) are formulas. The rest are not. (b) is not a formula because $x F$ is not well-formed. (d) and (e) are not formulas, because the variable x appears without being bound by a quantifier, in (d) there is no quantifier binding x , while in (e) the quantifier binds the variable x but does not touch Rxa .

In these last examples, it was important to understand how different quantifiers influence different parts of the formula. The crucial notions are *scope* and *binding*.

DEFINITION: The *scope* of a connective, operator or quantifier in a formula is the part of the formula which was introduced before that connective, operator or quantifier in the recursive definition of the formula.

EXAMPLE: Here are some examples of scopes in formulas.

So, the scope of the $\&$ in $\sim(Fa \& (Gb \supset Lba)) \vee Lab$ is the formulas Fa and $(Gb \supset Lba)$, since the conjunction combined these formulas, while the negation and disjunction occurs outside that conjunction.

The scope of $(\forall x)$ in $(\exists y)(Fy \& (\forall x)(Fx \supset Lxy))$ is $(Fx \supset Lxy)$ while the scope of $(\exists y)$ is $(Fy \& (\forall x)(Fx \supset Lxy))$.

Once we have the definition of scope we can define what it is for a quantifier to *bind* a variable.

DEFINITION: A quantifier $(\exists x)$ or $(\forall x)$ **BINDS** all occurrences of the variable x that occur its scope, provided that occurrence is not already bound by another quantifier with narrower scope.

EXAMPLE: In the formula below

$$(\exists x)(Fx \ \& \ (Gx \ \& \ (\forall x)(Fx \supset Gx)))$$

the red variables are bound by the existential quantifier $(\exists x)$, and the blue variables are bound by the universal quantifier $(\forall x)$. In $(\forall x)(Fx \supset Gx)$, it is clear that the variables x are bound by $(\forall x)$, but this $(\forall x)$ cannot bind anything *else* in the formula. It can only bind variables within its scope. Then since the $(\exists x)$ has the rest of the formula $(Fx \ \& \ (Gx \ \& \ (\forall x)(Fx \supset Gx)))$ within its scope, so it binds the remaining x variables there.



Ada Lovelace (1815–1852)

1.5 | TRANSLATING INTO PREDICATE LOGIC

Ada Lovelace is a Victorian Englishwoman, a mathematician and author, who worked on Charles Babbage's *Analytical Engine*. Some of her work included the first precisely specified algorithm to be carried out by a computer. So, it's correct to say

Some Victorian Englishwoman wrote a computer program.

How do we find the logical structure of this expression? First, we isolate the predicates that are used in the sentence.

$Vx = x$ is Victorian. $Ex = x$ is English. $Wx = x$ is a woman.
 $Px = x$ is a computer program. $Rxy = x$ wrote y .

We can do this piece-by-piece. We start with “Some Victorian Englishwoman...” Here, we'll understand “Victorian Englishwoman” as describing anyone who is Victorian, English and a woman. So, “Some Victorian Englishwoman...” will have this structure.

$$(\exists x)((Vx \ \& \ (Ex \ \& \ Wx)) \ \& \ \dots)$$

To complete our translation, we want to fill in the ‘...’ with whatever is required to say “ x wrote a computer program.” But x wrote a computer program if and only if there is some computer program that x wrote. That is, there is some y such that Rxy and Py , that is: $(\exists y)(Rxy \ \& \ Py)$. So, piecing this together, we get

$$(\exists x)((Vx \ \& \ (Ex \ \& \ Wx)) \ \& \ (\exists y)(Rxy \ \& \ Py))$$

On the other hand, if we want to say that

Every Victorian Englishwoman wrote a computer program.

we cannot merely substitute a universal quantifier for the existential quantifier, to get this formula:

$$(\forall x)((Vx \ \& \ (Ex \ \& \ Wx)) \ \& \ (\exists y)(Rxy \ \& \ Py))$$

This says not that every Victorian Englishwoman wrote a computer program, but that *everything* is both a Victorian Englishwoman and wrote a computer program. This says altogether too much. No, to say that every Victorian Englishwoman had some property we say

$$(\forall x)((Vx \ \& \ (Ex \ \& \ Wx)) \supset \dots)$$

where we place the complex predicate expressing that property in place of the ‘ \dots ’ So, to say that all Victorian Englishwomen wrote a computer program, we write

$$(\forall x)((Vx \ \& \ (Ex \ \& \ Wx)) \supset (\exists y)(Rxy \ \& \ Py))$$

FOR YOU: Translate the sentence

Every kitten loves some fish with green eyes.

Use this dictionary: Kx — x is a kitten. Fx — x is a fish. Gx — x has green eyes. Lxy — x loves y .

- (a) $(\forall x)(Kx \supset (\exists y)((Fy \ \& \ Gy) \ \& \ Lyx))$
- (b) $(\forall x)(Kx \ \& \ (\exists y)((Fy \ \& \ Gy) \ \& \ Lxy))$
- (c) $(\forall x)(Kx \supset (\exists y)((Fy \ \& \ Gy) \ \& \ Lxy))$
- (d) $(\exists y)(\forall x)(Kx \supset ((Fy \ \& \ Gy) \ \& \ Lxy))$
- (e) $(\forall x)(Kx \supset (\exists y)((Fy \ \& \ Gy) \supset Lxy))$

green eyes, and x loves y .

ANSWER: (c) is the correct answer. It says that there is some y which is a fish and has for x , if x is a kitten, then there is some y which is a fish and has

It's important to know the difference involved in the order of quantifiers. $\forall\exists$ differs from $\exists\forall$.

- $(\forall x)(\exists y)Sxy$ — *everything is smaller than something.*

This says that for any x we care to choose, we can find some y where x is smaller than y . If we were talking about numbers, for example, this is true. For any number x we choose, we could find another number (say $x + 1$) larger than x . If we reverse the order of the quantifiers, we get a very different formula:

- $(\exists y)(\forall x)Sxy$ — there's something such that everything is smaller than it.

This is *false* when we are talking about numbers. There is no number y such that *every* number is smaller than y , because this would mean y is *larger* than every number x . This is false for two reasons: (i) a number y cannot be larger than itself, or smaller than itself; and (ii) the numbers (say the natural numbers $\mathbb{N} = \{0, 1, 2, 3, 4, \dots\}$) keep on increasing without a bound, because $y + 1$ is always larger than y (or equivalently, y is smaller than $y + 1$)

EXAMPLE: Translate Some philosopher knows every linguist who owns a dog. Use the dictionary: $Px = x$ is a philosopher; $Lx = x$ is a linguist; $Dx = x$ is a dog; $Kxy = x$ knows y ; $Oxy = x$ owns y .

To start, “some philosopher knows every linguist who …” is

$$(\exists x)(Px \ \& \ (\forall y)((Ly \ \& \ \dots) \supset Kxy))$$

where the ‘…’ fills in the other conditions we require the linguist y to satisfy. In this case, we want to say that y owns a dog. To say that, we need to say that there is some z where z is a dog and y owns z . In other words: $(\exists z)(Dz \ \& \ Oyz)$. So, putting things together, we get

$$(\exists x)(Px \ \& \ (\forall y)((Ly \ \& \ (\exists z)(Dz \ \& \ Oyz)) \supset Kxy))$$

which says some philosopher knows every linguist who owns a dog.

We'll end this section with another example translation for you to try.

FOR YOU: Which formula is the best translation for

Some book on our reading list was read by every student.

Sx : x is a student. Bx : x is a book. l : Our reading list.

Rxy : x has read y . Oxy : x is on y .

- $(\forall x)(Sx \supset (\exists y)((By \ \& \ Oyl) \ \& \ Ryx))$
- $(\exists x)((Bx \ \& \ Oxl) \ \& \ (\forall y)(Sy \supset Ryx))$
- $(\forall x)(Sx \supset (\exists y)((By \ \& \ Oyl) \ \& \ Rxy))$
- $(\exists x)((Bx \ \& \ Oxl) \ \& \ (\forall y)(Sy \supset Rxy))$
- $(\forall x)(Sx \supset (\forall y)((By \ \& \ Oyl) \supset Rxy))$

ANSWER: (b) is correct. It states that there is some x that is a book on our reading list, and for every choice of y , if y is a student, then y has read x . This indeed says that some book on our reading list was read by every student.

1.6 | INSTANCES OF QUANTIFIED FORMULAS

We'll end this chapter with a short discussion of quantifiers and *instances*. A quantified formula $(\forall x)A$ or $(\exists x)A$ bears an important relationship with formulas involving the complex predicate A . The formula $(\forall x)(Wx \supset Qx)$ (say, all wombats are quadrupeds) bears an intimate relationship to the formula $Wa \supset Qa$, whenever a is a name. If all wombats are quadrupeds, then if a is an object, it follows that $Wa \supset Qa$. We will say that $Wa \supset Qa$ is an *instance* of the formula $(\forall x)(Wx \supset Qx)$.

The same sort of thing holds with the existential quantifier. The formula $(\exists x)(Wx \& \sim Qx)$, for example, has a special relationship with formulas like $Wb \& \sim Qb$. In this case, $Wb \& \sim Qb$ doesn't follow from $(\exists x)(Wx \& \sim Qx)$, rather, $Wb \& \sim Qb$ is a *stronger* formula than $(\exists x)(Wx \& \sim Qx)$. In other words, $Wb \& \sim Qb$ is strong enough to entail $(\exists x)(Wx \& \sim Qx)$. We say, too, that $Wb \& \sim Qb$ is an instance of the quantified formula $(\exists x)(Wx \& \sim Qx)$.

DEFINITION: Instances of a quantified formula. Given a collection of names, the **INSTANCES** of a quantified formula $(\forall x)A$ or $(\exists x)A$ are the formulas $A[x := n]$ for each name n in the collection.

EXAMPLE: We'll use the names $\{a, b\}$. The formula $(\exists x)(Fx \supset (\forall y)(Gy \supset Lxy))$ has instances $Fa \supset (\forall y)(Gy \supset Lay)$ and $Fb \supset (\forall y)(Gy \supset Lib)$. (It does *not* have $Fa \supset (Gb \supset Lab)$ as an instance.) The substitution of a name inside the inner quantifier here is only appropriate when $(\forall y)$ is the dominant operator in the formula.) $(\forall y)(Gy \supset Lay)$ has the instances $Ga \supset Laa$ and $Gb \supset Lab$.

Now there is an example for you to try.

FOR YOU: Use the names $\{a, b, c\}$. Which of the following formulas are instances of the formula $(\forall x)(\exists y)(Fx \supset (Gy \& Lxy))$?

- (a) $(\exists y)(Fa \supset (Gy \ \& \ Lay))$
 (b) $(\forall x)(Fx \supset (Ga \ \& \ Lxa))$
 (c) $Fa \supset (Gb \ \& \ Lab)$
 (d) $(\exists y)(Fb \supset (Gy \ \& \ Lby))$
 (e) $(\exists y)(Fb \supset (Gy \ \& \ Lay))$

ANSWER: The correct answers are (a) and (d). The other formulas are not instances of this starting formula. (b) is incorrect, as an instance always results from the processing of the outermost quantifier, in this case the $(\forall x)$ and not the inside $(\exists y)$. (c) is not correct, because an instance involves the processing of one quantifier, not two. (e) is incorrect, because when the outer quantifier $(\forall x)$ is processed, one name should be substituted for the variable x , not the two names a and b .

The notion of an instance of a quantified formula will become important in the next section, when we consider what it takes for a quantified formula to be true. And that is the topic we'll pursue in the next chapter.

MODELS FOR PREDICATE LOGIC

2

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- Models for a predicate language: non-empty domain and interpretation of names and interpretation of predicates.
- Standard names for objects in domain of a model, and how they are used in determining truth of quantified formulas.
- Evaluation of truth in a model of atomic formulas, and of complex formulas: $(\forall x)A$ is true in M iff all instances $A[x := d]$ are true in M , using standard names d ; $(\exists x)A$ is true in M iff one instance $A[x := d]$ is true in M , using standard names d .
- Models with a finite domain: $(\forall x)A$ equivalent to conjunction of instances; $(\exists x)A$ equivalent to disjunction of instances.
- Logical classifications: tautology, contradictory and contingency formulas of predicate logic.
- Relationships between formulas: logical consequence; logical equivalence; validity of arguments.

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 practise your skills and get familiar with each of these concepts.

SKILLS

These are the skills you will learn to demonstrate.

- Given a predicate logic formula and a model, determine whether or not the formula is true in the model.
- Given a predicate logic formula, describe a model (usually finite domain) in which the formula is true.
- Given a predicate logic formula, determine whether it is a tautology, it is contradictory or it is a contingency, using semantic analysis.
- Given two predicate logic formulas, determine whether or not there are logical relationships between them using semantic analysis.
- Given an argument form in predicate logic, determine whether or not it is valid using semantic analysis.

2.1 | INTERPRETING A LANGUAGE

In the previous series of lessons on the language of predicate logic, we have developed the idea of *splitting the atom* of atomic propositions, and revealing within the internal structure of *names*, *predicates* and *quantifiers*, *for all* and *there exists*. Our task in this series of lessons is to see how to *interpret* the language of predicate logic, so let's begin.

2.1.1 | INTERPRETING ATOMIC FORMULAS

- To *interpret* a language is to determine which sentences are *true* (1) and which ones are *false* (0).
- For propositional logic we can simply specify which of the atomic sentences are *true* and which ones are *false*.
- *Truth tables* for the connectives do the rest.
- For predicate logic, we need to do *more*.

For propositional logic, interpreting the language was quite easy. Each valuation specifies the truth or falsity of atomic propositions, which describes one row of a truth table, and from there, we can calculate the truth or falsity of complex formulas using the truth table rules for each of the propositional connectives. For predicate logic, we need to do rather more!

How do we interpret a simple atomic formula like Fa ?

- A *model* has to give *every* formula a truth value.
- How will we interpret Fa ?
 - What is the interpretation of the *name* a ? — An *object*.
 - So, the interpretation of a *1-place predicate*, F will need to *assign a truth value to every object*.
 - (Similarly, to interpret a 2-place predicate, we assign a truth value to every *pair* of objects, etc.)

Instead of valuations and rows of a truth table, predicate logic uses *models*, and each model must contain the ingredients needed to give every formula a truth value.

Start with an atomic formula Fa , where a is a name and F is a 1-place predicate symbol. The interpretation of a name a is an *object* named by that name, while the interpretation of a 1-place predicate symbol F will need to assign a truth value to each object, giving a YES/NO answer to the question *does this object have the property represented by F ?*.

Similarly, the interpretation of a 2-place predicate symbol will need to assign a truth value to each *pair of objects*, giving a YES/NO answer to the question *is the first object in the pair related to the second by the relationship represented by the predicate symbol?*

A *model* is some fragment of the world we want to talk about – how it actually is or how it could be.

Let's start by talking about the current leaders of the four largest economies in the world, as measured by total GDP, as of 2014: there is Barack Obama of the USA; Xi Jinping of China; Shinzo Abe of Japan; and Angela Merkel of Germany.



Let's start with a statement of the obvious:

Angela Merkel is a woman

Wa

Here, we let W be a 1-place predicate symbol for “is a woman”, and let the name a denote Angela Merkel. Then the atomic formula Wa is true. Easy.

Now some less well known facts about this fragment of the world consisting of these four world leaders. It is true that: none among them with a physics PhD is *not* a woman. Obama, Xi and Abe have degrees in politics and/or law – although Xi did study chemical engineering as an undergraduate. Merkel was awarded a doctorate in physics from the University of Leipzig in 1978, in the area of quantum chemistry.

OK, so let P be a 1-place predicate symbol for “has a PhD in physics”. To express this statement in predicate logic, we want to say:

None among them with a physics PhD is not a woman.

$$\sim(\exists x)(Px \ \& \ \sim Wx)$$

The formula says: “There does not exist x that is Px and not Wx ”. So none among them with a physics PhD is not a woman. But this is a negative way of stating it. In equivalent positive terms, we can truthfully assert: “All of them with a physics PhD are women”.

All of them with a physics PhD are women.

$$(\forall x)(Px \supset Wx)$$

This says: “For all x , if Px then Wx ”. This is *logically equivalent* to the formula expressing: “There does not exist x that is Px and not Wx ”. Don't worry: we will make precise the notion of *logical equivalence* later in this chapter.

Let's turn instead to a different fragment of the world, this ridiculously easy version of **Sudoku puzzles**. We will use little 4-by-4 ones instead of the usual 9-by-9. Here, the objects of interest are the 16 cells

in the grid, named a, b, c , up to p (the first 16 letters of the alphabet), as well as the number values 1, 2, 3 and 4 that can be assigned to these cells.

a	b	c	d
e	f	g	h
i	j	k	l
m	n	o	p

$Vg2$

Rfg

$(\forall x)(\forall y)(\forall z)((Vxy \ \& \ Rxz) \supset \sim Vzy)$

We can express a basic fact such as “Cell g has value 2” with an atomic formula $Vg2$. We can express that cell f and cell g are in the same row, or are *row-related* with the atomic formula Rfg . And we can express a more complex fact, that applies to all cells and all number values in the puzzle: “for all x , for all y and for all z : if x has value y and x is row-related to z , then it is not the case that z has value y ”. This formula would be true in a model of 9-by-9 Sudoku puzzles, as well as our little 4-by-4 ones.

Now suppose, instead, that we want to talk about *numbers*, the natural numbers or whole counting numbers:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, ...

Let E be a 1-place predicate meaning “is an even number”. Then the atomic formulas $E2$ and $\sim E3$ are both true in the model.

Let O be a 1-place predicate meaning “is an odd number”. Then the atomic formulas $\sim O2$ and $O3$ are both true in this model.

Now consider the quantified formulas:

$$(\forall x)(Ex \vee Ox) \quad \text{and} \quad (\forall x)\sim(Ex \ \& \ Ox)$$

This model of the natural numbers makes true the first of these, $(\forall x)(Ex \vee Ox)$, since all natural numbers are either even or odd. It also makes true the formula $(\forall x)\sim(Ex \ \& \ Ox)$, since no number can be both even and odd.

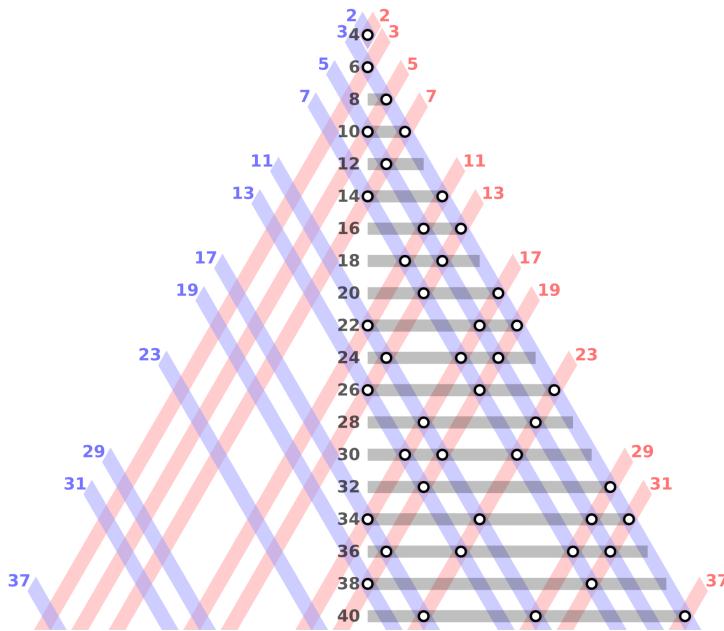
While it is easy to see that the two previous quantified formulas are true in this model of the natural numbers, it is not hard to devise formulas whose truth or falsity is *unknown* or *yet to be determined*. One of the most famous unsolved problems in mathematics is the *Goldbach conjecture* in number theory, which asserts that: **Every even number greater than 2 can be expressed as the sum of two prime numbers.**

$$(\forall x)((Ex \ \& \ Gx2) \supset (\exists y)(\exists z)(Py \ \& \ Pz \ \& \ Sxyz))$$

Here, G is a 2-place predicate with Gxy “ x is greater than y ”, so $Gx2$ says “ x is greater than 2”; P is a 1-place predicate meaning “is a prime

number”; and S is the 3-place predicate with $Sxyz$ meaning “ x is the sum of y and z ”. With these interpretations of predicates, the formula says: “For all numbers x , if x is even and x is greater than 2, then there exists y and z such that y and z are both prime and x is the result of adding y and z ”. For example, 12 is the sum of 7 and 5; 64 is the sum of 47 and 17.

This image illustrates the Goldbach Conjecture for even numbers from 4 to 40. Tracing the intersecting lines, for example, 18 is the sum of 11 and 7, but also 13 and 5.



Goldbach’s conjecture has been empirically verified by computer up to 4 quintillion or 4 times 10^{18} . But in spite of considerable effort from both professional and amateur mathematicians . . . there is still no proof that *every* even integer greater than 2 can be written as the sum of two primes.

2.2 | DEFINING MODELS

2.2.1 | THE DOMAIN OF A MODEL

The natural numbers, the parts of a Sudoku puzzle, 4 world leaders: these collections are all *domains*. Each model of predicate logic has a non-empty collection of objects called its *domain*.

A model has a non-empty collection of objects, D , its *domain*.

- Each *name* is interpreted by an object in the domain.
- Each *predicate* is interpreted by a truth value for each object (or pair, triple, ...) from the domain.
- The *quantifiers* range over the domain.

Each *name* in the language is interpreted by an object in the domain. That is clear: names name objects.

Each 1-place predicate is interpreted by a truth value for each object in the domain. Each 2-place predicate is interpreted by a truth value for each ordered pair of objects in the domain. And so on.

The quantifiers *for all* and *there exists* range over objects in the domain.

Example domains:

- **FOUR WORLD LEADERS IN 2014:** {Obama, Xi, Abe, Merkel}
- **SUDOKU PUZZLES:** the *cells* and *numbers*.
- **NATURAL NUMBERS:** {0, 1, 2, 3, 4, 5, ...}

Summary: A model consists of:

- A *domain* — a non-empty collection of objects.
- An *interpretation for every name* — i.e. an object.
- An *interpretation of every predicate* — i.e. a distribution of truth values over the domain.

A model for predicate logic is analogous to one row of a truth table in propositional logic: in each case, they contain all the information required to determine the truth value of atomic formulas of the language, predicate and propositional, respectively. In the next lesson, we will formally define models and give more examples.

2.2.2 | DEFINING MODELS

Our task here is to formally define models of predicate logic, making precise what we mean by an *interpretation* of names and predicates. Keep in mind that a model for predicate logic is analogous to one row of a truth table in propositional logic: in each case, they contain all the information required to determine the truth value of atomic formulas of the language.

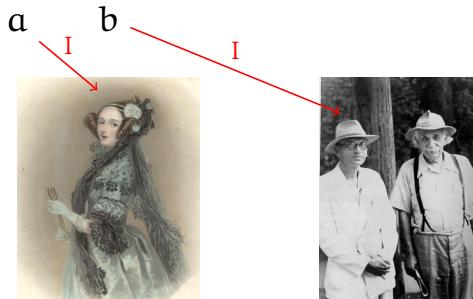
Definition A MODEL for a predicate logic language consists of:

- A non-empty DOMAIN D of objects.
- An INTERPRETATION FUNCTION I(\cdot), which interprets each name and predicate in the language.

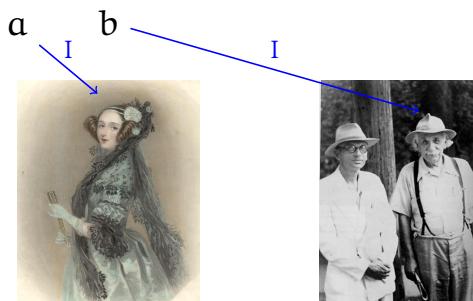
We will write $M = \langle D, I \rangle$ to say that the model M has domain D and interpretation function I. This lesson will focus on the interpretation part, with lots of examples.

2.2.3 | INTERPRETING NAMES

Let's begin with a model containing three objects: namely, three scientists: the nineteenth century English mathematician Ada Lovelace; the Austrian-Czech logician Kurt Gödel; and the German-born physicist Albert Einstein.



In an model of a language, each name a must be interpreted by an object $I(a)$ from the domain. In our domain $D = \{l, g, e\}$, one choice is to interpret the name a as Ada Lovelace, and the name b as Kurt Gödel, so $I(a) = l$ and $I(b) = g$. Letter a for Ada, obviously, and well, b for Brno, the birthplace of Kurt Godel; in 1906 when Gödel was born, Brno was part of the Austro-Hungarian empire, but today it is in the Czech Republic. That is *one* possible interpretation, but there are several others.



Another choice: make $I(a) = l$ and $I(b) = e$, so a names Lovelace and b names Einstein. Another: make $I(a) = l$ and $I(b) = l$, so a names Lovelace and b also names Lovelace. Yet another: make $I(a) = g$ and $I(b) = g$, so a names Gödel and b also names Gödel. And we haven't exhausted all possibilities. The only requirement on an interpretation I is that each name in the language must be mapped to *an* object in the domain. If a is a name, $I(a)$ is an object in the domain.

Suppose the domain $D = \{l, g, e\}$

How many *different ways* are there to assign objects to a and b ?

- (a) 3 (b) 5 (c) 6 (d) 8 (e) 9

In a model with domain $D = \{l, g, e\}$, containing 3 objects, there are 3 different ways to assign objects to each name. So if there are two names, a and b , then there are $3 \times 3 = 9$ different ways to assign objects to the two names. Correct answer is (e) 9.

In general, in a finite domain with n objects, if there are m -many names in the language, then there are n^m different interpretations assigning objects to names.

2.2.4 | INTERPRETING 1-PLACE PREDICATES

Now lets turn to interpreting predicates. Let F be a 1-place predicate symbol. If we interpret F to mean “is a mathematician”, then $I(F)$ will assign 1 or true to Lovelace and Gödel, and 0 or false to the physicist Einstein.



$$\begin{array}{c} \downarrow I(F) \\ 1 \end{array}$$



$$\begin{array}{c} \downarrow I(F) \\ 1 \end{array} \quad \begin{array}{c} \downarrow I(F) \\ 0 \end{array}$$

Consider another interpretation, say the predicate symbol F means “has their birth date in the first half of a month”. Then $I(F)$ would assign 1 to Lovelace and Einstein (birth dates 10 December and 14 March, respectively), while $I(F)$ would assign 0 to Gödel, with a birth date of 28 April.



$$\begin{array}{c} \downarrow I(F) \\ 1 \end{array}$$



$$\begin{array}{c} \downarrow I(F) \\ 0 \end{array} \quad \begin{array}{c} \downarrow I(F) \\ 1 \end{array}$$

Yet another interpretation: let F be a 1-place predicate symbol interpreted as meaning “is a physicist”. So $I(F)$ would assign 1 to Einstein and 0 to both Lovelace and Gödel. So $I(F)$ divides the domain into 2 camps: one with all the objects assigned 1 because they do have the property denoted by F , and the other with all the objects assigned 0 because they do not have the property denoted by F .



Equivalently, for a 1-place predicate, we can use a 1-dimensional table with a *column* of 0-1 truth-values, with the length of the column equal to the size of the domain. So for a size-3 domain, as here, we need a length-3 column of truth-values to represent $I(F)$.

	$I(F)$
l	0
g	0
e	1

This 1-dimensional table describes the *interpretation* $I(F)$ of the predicate symbol F , since it assigns a truth value for each object in the domain.

Suppose $D = \{l, g, e\}$.

How many *different ways* are there to interpret a 1-place predicate symbol F ?

- (a) 3 (b) 5 (c) 6 (d) 8 (e) 9

In a model with domain $D = \{l, g, e\}$, containing 3 objects, a 1-place predicate F is interpreted by a length-3 column of 0s and 1s, and there $8 = 2^3$ of these. Correct answer is (d) 8.

In general, in a finite domain with n objects, there are 2^n many different interpretations of a 1-place predicate symbol.

2.2.5 | INTERPRETING k -PLACE PREDICATES, FOR $k \geq 2$

Now consider a 2-place predicate symbol R , and suppose we interpret Rab to mean “ a admires b ”. (Notice: the order really matters.) This interpretation $I(R)$ can be represented by a 2-dimensional *table*:

$I(R)$	l	g	e
l	1	0	0
g	1	0	1
e	1	1	1

The intended interpretation $I(R)$ has Lovelace admiring Lovelace, but not admiring either Gödel or Einstein, because Lovelace was dead before they were born. $I(R)$ could have Gödel admiring Lovelace and Einstein, but not himself – Gödel was plagued by anxieties. And Einstein admires everyone: Lovelace, Gödel and himself. Here we use a 2-dimensional table, and look at the intersection of the rows and the columns. To check whether Lovelace admires Gödel, we look at the intersection of the Lovelace row with the Gödel column, and see the entry is 0, so no. To check whether Gödel admires Lovelace, we look at the intersection of the Gödel row with the Lovelace column, and see the entry is 1, so yes.

For a 3-place predicate symbol, each interpretation must assign a truth value to a 3-tuple or ordered triple of objects in the domain. In general, each **k-place predicate** is interpreted by an assignment of truth values to all **k-tuples** of objects in the domain.

A predicate behaves like a proposition with holes in it: fill in the appropriate number of holes with names for objects, and the result is a proposition. that is either true or false.

2.3 | TRUTH IN A MODEL

In this section, we will get to the heart of predicate logic semantics, and see how models determine the truth or falsity of predicate logic formulas.

2.3.1 | FROM MODELS TO TRUTH: SIMPLE FORMULAS

We start with simple formulas made from atoms or compound formulas built from atoms using propositional connectives.

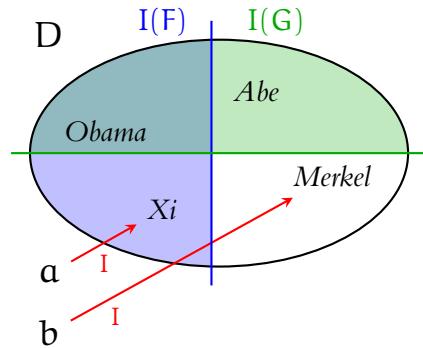
Our running example will be the size-4 model whose domain consists of 2014 world leaders: Barack Obama, Xi Jinping, Shinzo Abe, and Angela Merkel.



$$D = \{\text{Obama}, \text{Xi}, \text{Abe}, \text{Merkel}\}$$

$$M = \langle D, I \rangle$$

where I interprets the names a and b , the 1-place predicate symbols F and G , and the 2-place predicate symbol R , as shown:



$$D = \{\text{Obama}, \text{Xi}, \text{Merkel}, \text{Abe}\}$$

$$I(a) = \text{Xi}, I(b) = \text{Merkel}$$

	I(F)	I(G)
Obama	1	1
Xi	1	0
Merkel	0	0
Abe	0	1

I(R)	Obama	Xi	Merkel	Abe
Obama	0	0	0	0
Xi	1	0	1	1
Merkel	1	0	0	1
Abe	1	0	0	0

This model $M = \langle D, I \rangle$ interprets name a as Xi Jinping and the name b as Angela Merkel. Suppose we interpret F to mean “has a professional doctorate in Law”; that is, has a JD or LLD degree. Then $I(F)$ will assign 1 for true to both Obama and Xi, and 0 for false to Merkel and Abe. We can visually represent $I(F)$ as dividing the domain D in two, as shown.

Now we have enough information to determine the truth value of the atomic formula Fa , because Fa now has a meaning: “Xi Jinping has a professional doctorate in Law”, which is true.

How about the atomic formula Ga ? Well, we need an interpretation for predicate G . Suppose G means “has an undergraduate degree in political science”. Then $I(G)$ will assign 1 to Obama and Abe, and 0 to Xi and Merkel. We also represent $I(G)$ as dividing the domain D in two. So the atomic formula Ga has the meaning: “Xi Jinping has an undergraduate degree in political science”, which is false: his undergraduate degree is in chemical engineering.

If we consider the conjunction $(Fa \ \& \ \neg Ga)$, the result must be true, because both conjuncts are true; since Ga is false, $\neg Ga$ must be true.

Next, consider the conjunction $(Fb \ \& \ \neg Gb)$. In the model M , the interpretation $I(b)$ of name b is Angela Merkel. So the conjunction $(Fb \ \& \ \neg Gb)$, because Fb is false.

So for simple formulas involving 1-place predicates, that all looks good. Now try 2-place predicates. When R is a 2-place predicate, there

are two holes with names for objects to be filled. To determine the truth of an atomic formula Rab , we need to examine the interpretation $I(R)$ of the predicate R .

Suppose Rab means “ a is older than b ”. So $I(R)$ is given by the 2-dimensional table as shown: Obama is the youngest of the four, so he is not older than anyone, and $I(R)$ has a row of 0s for Obama. Xi Jinping is the oldest of the four, so the row for Xi has a 1 for all the others and a 0 for Xi himself, since Xi is not older than Xi. Merkel is older than Obama and Abe, but not older than Xi or herself. And Abe is older than Obama, but not older than Xi, Merkel or himself. Since under interpretation I , the name a denotes Xi and the name b is Merkel, and there is a 1 at the intersection of the Xi row with the Merkel column in $I(R)$, so we can see that Rab is true in this model.

To summarise: in this model $M = \langle D, I \rangle$, we can determine truth values for these (and similar) simple formulas:

Fa	TRUE
Ga	FALSE
$Fa \ \& \ \sim Ga$	TRUE
$Fb \ \& \ \sim Gb$	FALSE
Rab	TRUE

We can now spell out the rules for evaluating simple formulas of predicate logic. Remember that a k -place predicate symbol is like a proposition with k holes in it. If we fill in those holes with the right number of names, the result is an atomic proposition.

Truth for atomic formulas is determined completely by the interpretation I in the model. For the propositional connectives, truth is propagated to complex formulas in the same way as it is in propositional logic.

Rules for Evaluating Simple Formulas:

Truth in a model $M = \langle D, I \rangle$:

- $Fa_1 \dots a_k$ is true in M iff $I(F)$ assigns 1 to the k -tuple of objects $\langle I(a_1), \dots, I(a_k) \rangle$ from domain D of M .
- $\sim A$ is true in M iff A is not true in M .
- $(A \ \& \ B)$ is true in M iff A is true in M and B is true in M .
- $(A \vee B)$ is true in M iff A is true in M or B is true in M .
- $(A \supset B)$ is true in M iff A is not true in M or B is true in M .
- $(A \equiv B)$ is true in M iff both A and B are true in M , or both A and B are not true in M .

Now it is your turn to evaluate truth in a model M . We go back to the model from last lesson with domain consisting of Lovelace, Gödel and Einstein, where $I(a)$ is Lovelace, $I(b)$ is Einstein, $I(F)$ means “has their birth date in the first half of a month”, as shown, and $I(R)$ means the first “admires” the second, with table as shown. Which of these formulas are true in this model M ?

Exercise (i): Which of these formulas are true in *this* model M ?

$$M = \langle D, I \rangle, \quad D = \{l, g, e\}, \quad I(a) = l, \quad I(b) = e;$$

	$I(F)$		$I(R)$	l	g	e
l	1		l	1	0	0
g	0		g	1	0	1
e	1		e	1	1	1

- (a) Fa
- (b) $\sim Fb$
- (c) $(Fa \supset \sim Fb)$
- (d) Rab
- (e) $(Rab \vee Rba)$

intersection of the Einstein row and the Lovelace column.

(e) $(Rab \vee Rba)$ **TRUE** because Rba is true: there is a 1 at the intersection of the Lovelace row ($I(a)$ is Lovelace) and the Einstein column ($I(b)$ is Einstein).

(d) Rab **FALSE** because there is a 0 at the intersection of the Lovelace row ($I(a)$ is Lovelace) and the Einstein column, the conditional is false.

(c) $(Fa \supset \sim Fb)$ **FALSE** because Fa is true and $\sim Fb$ is false, so Einstein.

(b) $\sim Fb$ **FALSE** because $I(b)$ is Einstein and $I(F)$ assigns 1 to Lovelace.

(a) Fa **TRUE** because $I(a)$ is Lovelace and $I(F)$ assigns 1 to Lovelace.

2.3.2 | FROM MODELS TO TRUTH: QUANTIFIERS

Having sorted out the semantics of simple formulas, we next move on to the evaluation of truth in a model for quantified formulas of predicate logic. We come back to our running example will be the size-4 model whose domain consists of 2014 world leaders: Barack Obama, Xi Jinping, Angela Merkel, and Shinzo Abe. In this model, the 1-place predicate F means “has a professional doctorate in Law” (a JD or LLD

degree), while the 1-place G means “has an undergraduate degree in political science”. The 2-place predicate R is such that Rab means “ a is older than b ”: the first is older than the second.

$$D = \{ \text{Obama}, \text{Xi}, \text{Abe}, \text{Merkel} \}$$

$$M = \langle D, I \rangle$$

where I interprets the names a and b , the 1-place predicate symbols F and G , and the 2-place predicate symbol R , as shown:

$$I(a) = \text{Xi}, \quad I(b) = \text{Merkel};$$

	$I(F)$	$I(G)$
Obama	1	1
Xi	1	0
Merkel	0	0
Abe	0	1

$I(R)$	Obama	Xi	Merkel	Abe
Obama	0	0	0	0
Xi	1	0	1	1
Merkel	1	0	0	1
Abe	1	0	0	0

Consider the following predicate logic formulas and their semantic evaluation in the model M .

- $(\exists x)Rax$ is **TRUE** in M because Rab is true in M .
Xi is older than Merkel, hence Xi is older than someone.
- $(\exists x)Rbx$ is **TRUE** in M
because this means Merkel is older than someone, which is true: Merkel is older than Obama and Merkel is older than Abe. The problem is that there are only two names: a for Xi and b for Merkel. $(\exists x)Rbx$ is true in M because Rbo is **TRUE** in M , where o is a name for Obama (and Rbe is also **TRUE** in M , where e is a name for Abe).
- $(\forall x)(Rax \supset \neg Fx)$ is **FALSE** in M
since if we instantiate x with name o , we get
 $(Rao \supset \neg Fo)$ which is **FALSE**
because antecedent Rao is **TRUE** and the consequent $\neg Fo$ is **FALSE** (because Obama does have a professional doctorate in Law, so Fo is **TRUE** in M).

In our listing of the domain, let's abbreviate each of them with a one-character symbol: o for Obama, i for Xi (because x is already used for variables), m for Merkel, and e for Abe (since a and b are already in use as names, we needed to go to the third letter in Abe's name). We add additional names o, i, m and e to give a means to refer to any of the four objects in the domain.

2.3.3 | STANDARD NAMES

In order to evaluate quantified formulas in our 2014 world leaders models, we introduced a *standard name* for each of the objects in our domain, so we could refer by name to each of them, and use those names in atomic formulas to take instances of quantified formulas. Note that these standard names are additional, over and above any names already occurring in the predicate language being interpreted. In the language we are currently working with, there is already **a** and **b**.

- The **STANDARD NAME** for a domain element is the name used to describe the element in your listing of the domain.
- We write the domain elements normally as {**o**, **i**, **m**, **e**}, but when we want or need to distinguish the *names* from the objects themselves, we'll use *bold* for their standard names: **o**, **i**, **m**, **e**.
- (For writing on paper you can use an underline to describe the domain, like this: $D = \{\underline{o}, \underline{i}, \underline{m}, \underline{e}\}$.)
- If **d** is a standard name for object d , then $I(\mathbf{d}) = d$.

When **d** is a standard name for object d , then we will *always* interpret **d** as the object d . To repeat, standard names are a technical device that lets us refer by name to *each object in the domain*, and lets us use those names in atomic formulas to in order to take all relevant instances of quantified formulas.

- The **INSTANCES** of a quantified formula $(\forall x)A$ or $(\exists x)A$ are the results $A[x := \mathbf{n}]$ of substituting in *each* standard name **n** for the bound variable x , once the quantifier is removed.
- The instances of $(\forall x)Rxa$ in the domain {**o**, **i**, **m**, **e**} are

$$\mathbf{Ro}a \quad \mathbf{Ria} \quad \mathbf{Rma} \quad \mathbf{Re}a$$

- The instances of $(\exists x)(Rxa \ \& \ (\exists y)(Fy \ \& \ Rxy))$ are

$$\begin{array}{ll} \mathbf{Ro}a \ \& \ (\exists y)(Fy \ \& \ \mathbf{Roy}) & \mathbf{Ria} \ \& \ (\exists y)(Fy \ \& \ \mathbf{Riy}) \\ \mathbf{Rma} \ \& \ (\exists y)(Fy \ \& \ \mathbf{Rmy}) & \mathbf{Re}a \ \& \ (\exists y)(Fy \ \& \ \mathbf{Rey}) \end{array}$$

2.3.4 | EVALUATING QUANTIFIED FORMULAS

Adopting the convention of adding standard names for all objects in the domain of a model, we are now ready to give the rules for evaluating quantified formulas of predicate logic.

- A *universally quantified formula* $(\forall x)A$ is true in M if and only if *every* instance $A[x := \mathbf{n}]$ is true in M .
- An existentially quantified formula $(\exists x)A$ is true in M if and only if *some* instance $A[x := \mathbf{n}]$ is true in M .

Let's work through the evaluation of a more complex formula in our 2014 world leaders model.

$$M = \langle D, I \rangle, \quad D = \{o, i, m, e\}, \quad I(a) = i, \quad I(b) = m;$$

	I(F)	I(G)		I(R)	o	i	m	e
o	1	1	o	0	0	0	0	0
i	1	0	i	1	0	1	1	
m	0	0	m	1	0	0	1	
e	0	1	e	1	0	0	0	

Example (1): The formula:

$$(\forall x)(Gx \supset (\exists y)(Fy \& Ryx))$$

means: “For everyone who has an undergraduate degree in political science, there is someone older than them who has a professional doctorate in Law.” This formula is **TRUE** in M because all four instances of the \forall -quantified x are true.

The detailed analysis of the four instances of the formula is as follows:

- $G\mathbf{o} \supset (\exists y)(Fy \& Ry\mathbf{o})$ is **TRUE** in M, because
 - $G\mathbf{o}$ is true in M.
 - $(F\mathbf{i} \& R\mathbf{io})$ is true in M.
- $G\mathbf{i} \supset (\exists y)(Fy \& Ry\mathbf{i})$ is **TRUE** in M, because
 - $G\mathbf{i}$ is false in M.
- $G\mathbf{m} \supset (\exists y)(Fy \& Ry\mathbf{m})$ is **TRUE** in M, because
 - $G\mathbf{m}$ is false in M.
- $G\mathbf{e} \supset (\exists y)(Fy \& Ry\mathbf{e})$ is **TRUE** in M, because
 - $G\mathbf{e}$ is true in M.
 - $(F\mathbf{i} \& R\mathbf{ie})$ is true in M.

Example (2): The formula:

$$(\forall x)(Fx \supset (\exists y)(Gy \& Rx\mathbf{y}))$$

means: “For everyone who has a professional doctorate in Law, there is someone *younger* than them who has a undergraduate degree in political science” This formula is **FALSE** in M because the instance $x := \mathbf{o}$ of the \forall -quantified x is false.

The analysis of the false instance $x := \mathbf{o}$ of the formula is as follows:

- $\text{Fo} \supset (\exists y)(Gy \ \& \ Ro y)$ is TRUE in M
 - Fo is true in M .
 - $(Go \ \& \ Ro o)$ is false.
 - $(Gi \ \& \ Ro i)$ is false.
 - $(Gm \ \& \ Ro m)$ is false.
 - $(Ge \ \& \ Ro e)$ is false.

Exercise (ii): Which of these formulas are true in *this* model M ?

$$M = \langle D, I \rangle, \quad D = \{l, g, e\}, \quad I(a) = l, \quad I(b) = e;$$

	I(F)	I(R)		
l	1	1	0	0
g	0	g	1	0
e	1	e	1	1

- (a) $(\exists x)\sim Fx$
- (b) $(\exists x)Rx x$
- (c) $(\forall x)Rx x$
- (d) $(\forall x)(\forall y)(Rx y \supset Ry x)$
- (e) $(\forall x)(\exists y)(Fy \ \& \ Rx y)$

- (H $\&$ RII), (Fe $\&$ Rge), and (Fe $\&$ Ree) are true in M .
(e) $(\forall x)(\exists y)(Fy \ \& \ Rx y)$ is TRUE in M because these 3 formulas
true in M .
(d) $(\forall x)(\forall y)(Rx y \subset Ry x)$ is FALSE in M because $\sim(Rel \subset Rel)$ is
true in M .
(c) $(\forall x)Rx x$ is FALSE in M because $\sim Rgg$ is true in M .
(b) $(\exists x)Rx x$ is TRUE in M because RII is true in M .
(a) $(\exists x)\sim Fx$ is TRUE in M because $\sim Fg$ is true in M .

2.4 | FINITE AND INFINITE MODELS

2.4.1 | FINITE MODELS

So far in this series of lessons, we have mostly looked at finite models, such as our Four world leaders 2014 model.

$$D = \{ \text{Obama}, \text{Xi}, \text{Merkel}, \text{Abe} \}$$

A model is called *finite* whenever its domain is a finite set of objects. In contrast, some models have *infinite* domains. For example, the Natural

Numbers \mathbb{N} :

$$\{0, 1, 2, 3, 4, 5, \dots\}$$

Technically, it is known as a *countably infinite* set; there are even larger infinities that we will be relevant later in Chapter 5 in the Mathematics application section of the course.

The definition of truth in a model is the same for all models, finite or infinite. But a model with a *finite* domain, we can simplify the clauses for truth in a model for quantified formulas.

- In a finite model M , formula $(\forall x)A$ is true in M if and only if the **conjunction of all its instances** is true in the model M .
- In a finite model M , formula $(\exists x)A$ is true in M if and only if the **disjunction of all its instances** is true in the model M .

For example, if $D = \{o, i, m, e\}$ is the domain of a model M , then:

$$(\forall x)Rax$$

is true in the model M if and only if

$$Rao \text{ & } Rai \text{ & } Ram \text{ & } Rae$$

is true in M .

What this means is that if we focus on a particular finite model, the quantifiers are *useful*, and let us be more *concise*, but they do not give us any more *expressive power* than propositional logic (particularly ‘&’ and ‘ \vee ’) coupled with predicate symbols and names which together generate atomic formulas.

This situation changes radically when we go *infinite*.

2.4.2 | INFINITE DOMAINS

We need our original definition of truth in a model because some models are infinite.

- Consider formalising arithmetic properties in predicate logic, with domain $D = \mathbb{N} = \{0, 1, 2, 3, \dots\}$, the natural numbers.
- Look at the 2-place predicate S , interpreted as “is strictly smaller than”: Sab is true if and only if a is strictly smaller than b .

$I(S)$	0	1	2	3	4	\dots
0	0	1	1	1	1	\dots
1	0	0	1	1	1	\dots
2	0	0	0	1	1	\dots
3	0	0	0	0	1	\dots
4	0	0	0	0	0	\dots
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

We can describe the interpretation $I(S)$ in an infinite table. The row for number 0 shows that 0 is strictly less than 1, 2, 3 and so, but 0 is not strictly less than 0. The row for 1 shows that 1 is strictly less than 2, 3, 4 and so on, but 1 is not strictly less than 0 or 1. Even though we can only write out a finite fragment of the table, we know how to fill in as much as we need, and how the pattern continues indefinitely. The table pattern here is *triangular*: there is an upper right-side triangle of 1s starting above the diagonal, and a lower left-side triangle of 0s including the diagonal. The diagonal of a table for the interpretation of a 2-place predicate determines the truth value of all atomic formulas Saa .

We use the **bold** symbols

0 1 2 3 4 \dots

as standard names for the numbers in the domain \mathbb{N} .

Example (1): The formula:

$$(\forall x)S\mathbf{0}x$$

says: “For every number x , **0** is strictly smaller than x ”, or more concisely, “**0** is strictly smaller than every number”.

In the model $M = \langle D, I \rangle$, this formula is **FALSE** in M because $\sim S\mathbf{0}\mathbf{0}$ is true in M – even though $S\mathbf{0}\mathbf{1}$, $S\mathbf{0}\mathbf{2}$, ... are all true in M ; one false instance is enough.

We try another example with nested quantifiers.

Example (2): The formula:

$$(\forall x)(\exists y)Sxy$$

says: “For every number x , there is a number y such that x is strictly smaller than y ”, or more succinctly, “For every x there is a y strictly larger than x ”.

In the model $M = \langle D, I \rangle$, this formula is **TRUE** in M because all infinitely many instances of the \forall -quantified formula are true:

$(\forall x)(\exists y)Sxy$ is true in M if and only if ...

- $(\exists y)S0y, (\exists y)S1y, (\exists y)S2y, \dots$ are all true.
 - $(\exists y)S0y$ is true because $S01$ is true;
 - $(\exists y)S1y$ is true because $S12$ is true;
 - $(\exists y)S2y$ is true because $S23$ is true;
 - and so on ...
 - $(\exists y)Sny$ is true because $Sn(n+1)$ is true;
 - and so on ...

Hence $(\forall x)(\exists y)Sxy$ is true in M .

Now it is your turn. The model has the same domain the infinite set of Natural Numbers \mathbb{N} ; the 2-place predicate S is interpreted as “is strictly less than”, as before, and the 2-place predicate R is interpreted as “differing by 1”, so Rab is true exactly when $b = a + 1$ or $a = b + 1$. In particular, $R01$ and $R10$ are true; $R12$ and $R21$ are true; and $R23$ and $R32$ are true; and so on.

$I(S)$	0	1	2	3	4	...	$I(R)$	0	1	2	3	4	...
0	0	1	1	1	1	...	0	0	1	0	0	0	...
1	0	0	1	1	1	...	1	1	0	1	0	0	...
2	0	0	0	1	1	...	2	0	1	0	1	0	...
3	0	0	0	0	1	...	3	0	0	1	0	1	...
4	0	0	0	0	0	...	4	0	0	0	1	0	...
:	:	:	:	:	:	..	:	:	:	:	:	:	..

Exercise: Which of the following sentences are true in this model?

- (a) $(\exists x)Rxx$
- (b) $(\forall x)(\forall y)(Rxy \supset (Sxy \vee Syx))$
- (c) $(\forall x)(\forall y)(Rxy \supset Ryx)$
- (d) $(\forall x)(\exists y)(Sxy \& Rxy)$

is smaller than the second, or the second is smaller than the first”.
muila says „If two numbers differ by 1, then either the first

(b) $(\forall x)(\forall y)(Rxy \subset (Sxy \vee Syx))$ is TRUE in M . The for-

the diagonal of the table for $I(R)$.

(a) $(\exists x)Rxx$ is FALSE in M , as $(\forall x)\sim Rxx$ is true in M , visible from

(p) $(\forall x)(\exists y)(Sxy \wedge Rxy)$ is TRUE in M. The formula asserts that for all x , there exists y such that Sxy and Rxy . This is also true. For each choice of $x = n$, we can choose y to be $n+1$, the next number, and we have both $Sn(n+1)$ and $Rn(n+1)$ true in this model.

(c) $(\forall x)(\forall y)(Ryx \subseteq Rxy)$ is TRUE in M. The formula asserts that for all x , Ryx implies Rxy . R is interpreted by a symmetric relation, and this is visible from the table for $I(R)$ because there are mirror images across the diagonal of the table.

2.5 | CLASSIFYING FORMULAS

Models allow us to see how propositions work, and what the world needs to be like for them to be true. Models also allow us to *classify* propositions according to their logical status, and allow us to *compare* propositions with other propositions.

2.5.1 | TAUTOLOGIES

The first classification is that of tautologies. A formula A is a tautology if and only if the formula is true in *every* model M. Recall that in propositional logic, tautologies are true under every truth valuation or every row of the truth table. Each model of predicate logic is analogous to one row of the truth table in propositional logic, in that it contains all the information needed to determine the truth of atomic formulas, and from there, the truth of all formulas.

A formula A is a **TAUTOLOGY** if and only if it is true in every model M.

So, being a tautology means true in *all* models. But *which* models? Well, we need to check all models that interpret the names and predicates occurring in the formula. Compare propositional logic, where we needed to check all truth valuations to the atomic propositions within the formula.

Example (1): The formula:

$$(\exists x)Fx \vee (\forall x)\sim Fx$$

is a tautology. The formula says: “Either something is an F, or everything is not an F”.

We can see this by *direct reasoning*.

Consider some model $M = \langle D, I \rangle$. If the table for $I(F)$ contains a 1, then $(\exists x)Fx$ is true in M – due to that instance. Otherwise, $(\forall x)\sim Fx$ is true, since every instance of $\sim Fx$ is true in M . Either way, $(\exists x)Fx \vee (\forall x)\sim Fx$ is true in M .

We assumed nothing about M other than it contains an interpretation of F . Hence we can conclude that $(\exists x)Fx \vee (\forall x)\sim Fx$ is true in *every* model. It's a *tautology*.

Example (2): The formula:

$$(\forall x)\sim Fx \equiv \sim(\exists x)Fx$$

is a tautology. The formula says: “Everything is not F if and only if there are no F things”.

We can see this by *indirect reasoning*.

Suppose we have a model $M = \langle D, I \rangle$ where $(\forall x)\sim Fx$ is true and $\sim(\exists x)Fx$ is false. If $(\forall x)\sim Fx$ is true, then every instance of $\sim Fx$ is true, so *no* instance of Fx is true. And if $\sim(\exists x)Fx$ is false, $(\exists x)Fx$ is true, which means that some instance of Fx is true. Which contradicts what we've already seen, so that can't happen.

Alternatively, suppose that in $M = \langle D, I \rangle$, $\sim(\exists x)Fx$ is true and $(\forall x)\sim Fx$ is false. If $\sim(\exists x)Fx$ is true, then $(\exists x)Fx$ is false, so each instance of Fx is false. If $(\forall x)\sim Fx$ is not true, then not every instance of $\sim Fx$ is true. But that means some instance of Fx is true. This contradicts what we've already seen, so that can't happen either.

Either way, we cannot have a model M where one of the formulas $\sim(\exists x)Fx$ or $(\forall x)\sim Fx$ is true and the other is false. Hence the biconditional

$$((\forall x)\sim Fx \equiv \sim(\exists x)Fx)$$
 is true in *every* model. It is a *tautology*.

The formula $((\forall x)\sim Fx \equiv \sim(\exists x)Fx)$ is a quantifier version of a De Morgan?‐s-like law, expressing the negation‐duality of \forall and \exists , and it is used often in predicate logic reasoning. We include a list of useful tautologies of predicate logic at the end of this Chapter.

2.5.2 | CONTRADICTIONS

At the other end of the spectrum to tautologies, the opposite classification is that of a *contradiction*. A formula A is a contradiction if and only if it comes out *false* in every model.

A formula A is a **CONTRADICTION** if and only if it is false in every model M .

It follows immediately from the definitions that A is a contradiction if and only if $\sim A$ is a tautology.

Example (1): The formula:

$$(\exists x)(\forall y)Rxy \ \& \ \sim(\forall y)(\exists x)Rxy$$

is a contradiction.

We can see this by *indirect reasoning*.

Suppose $(\exists x)(\forall y)Rxy \ \& \ \sim(\forall y)(\exists x)Rxy$ is true in some model $M = \langle D, I \rangle$. All we assume about model M is that it includes an interpretation $I(R)$ for the 2-place predicate symbol R .

Take the first conjunct: since $(\exists x)(\forall y)Rxy$ is true in M , there is a d in D such that $(\forall y)Rdy$ is true. So object d is R -related to everything. This means that in the table for $I(R)$, the d -row must be all 1s.

Next, the second conjunct: since $\sim(\forall y)(\exists x)Rxy$ is true in M , there is an e in D such that $\sim(\exists x)Rxe$ is true. This means there does not exist any x that is R -related to e . In the table for $I(R)$, this means that the e -column is all 0s. Since $(\forall y)Rdy$ is true, we also know Rde is true. But $\sim(\exists x)Rxe$ is true, so Rde is false. This is a contradiction: at the intersection of the d -row and the e -column, there must be both a 1 and a 0 for Rde . So no such model M can exist, and hence $(\exists x)(\forall y)Rxy \ \& \ \sim(\forall y)(\exists x)Rxy$ is a *contradiction*.

Now it is your turn to assess tautology and contradiction status.

Exercise: These formulas are either tautologies or contradictions.

Which of them are the contradictions?

- (a) $(\forall x)(Fx \ \& \ Gx) \ \& \ (\exists x)\sim Fx$
- (b) $(\exists x)(Fx \supset Gx) \equiv \sim(\forall x)(Fx \ \& \ \sim Gx)$
- (c) $(\exists x)(Fx \supset Gx) \equiv (\forall x)(Fx \ \& \ \sim Gx)$
- (d) $(\exists x)(\forall y)Rxy \supset (\forall y)(\exists x)Rxy$
- (e) $(\forall x)(\exists y)Rxy \ \& \ (\exists x)(\forall y)\sim Rxy$

the previous example, except with $\sim Rx$ instead of Rxy . The other teasing out the meaning to show this is impossible. (e) is similar to positing that there exists a model in which the formula is true, and (a) (c) and (e) are contradictions, and each can be shown by sup-

two, (b) and (d) are tautologies.

2.5.3 | CONTINGENCIES

Tautologies and contradictions are the extreme ends of the spectrum. Intuitively, it would seem that *most* formulas are somewhere in the middle, sometimes true and sometimes false, depending on the model.

A formula A is **CONTINGENT** if and only if it is true in at least one model, and false in another model.

In other words, A is contingent if and only if it is not a contradiction, and it is also not a tautology.

Examples: The following formulas are contingent:

$$\begin{aligned} & \exists x \\ & (\exists x)(Fx \ \& \ Gx) \\ & (\forall x)Fx \vee (\forall x)\sim Fx \end{aligned}$$

In each case, we can find two models M_1 and M_2 such that the formula is true in the first model M_1 and the negation of the formula is true in the second model M_2 .

For the last one, the disjunctive universal formula:

$$(\forall x)Fx \vee (\forall x)\sim Fx$$

asserts that either the table for $I(F)$ has a column of all 1s or a column of all 0s. This could be the case, but simply by changing one bit entry in the table for $I(F)$, we could make the formula false. This is a basic approach to thinking about contingency. Find one model in which the formula is true, and then think about how to tweak the model to make the formula false.

Exercise: Which of these formulas are contingencies?

- (a) $(\forall x)(\exists y)Rxy \ \& \ (\exists x)(\forall y)\sim Rxy$
- (b) $(\forall x)(\forall y)(Rxy \supset Ryx)$
- (c) $(\exists x)(Fx \supset (\forall y)Fy)$
- (d) $(\forall x)(\exists y)Rxy \supset (\exists y)(\forall x)Rxy$
- (e) $(\exists y)(\forall x)Rxy \supset (\forall x)(\exists y)Rxy$

Formula (a) is a **CONTRADICTION** as we saw in the previous exercise. Formula (b) is a **TAUTOLOGY**. The middle three, (b), (c) and (d) are all contingencies. For example, (b) asserts that the relation R is symmetric in x and y , which is sometimes true, sometimes false. (c) is a tautology, and (d) is a contingency, while (e) is a tautology.

2.6 | RELATIONSHIPS BETWEEN FORMULAS

In the previous lesson, we looked at the classification of single formulas: tautologies, contradictions and contingencies. In many applications of logic, we want to know about *relationships between* propositions.

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

This is clear in assessing an argument from a list of premises to a conclusion: *validity* is the core logical relationship of interest. In linguistic analysis as well as in the analysis of digital electronic circuits and in logic programming, we are interested in when two formulas are *logically equivalent*, in particular, when one of them is much simpler than the other, or one is in a desired form or shape, and the other isn't. In digital systems, simpler formulas correspond to smaller, more compact circuits. You might want to review the corresponding material from propositional logic in **LLI1 §2-1**.

2.6.1 | LOGICAL CONSEQUENCE

We start with relationships between single propositions.

Formula B is a **LOGICAL CONSEQUENCE** of formula A , or A **LOGICALLY IMPLIES** B , exactly when for every model in which A is true, B is true too. We write this as $A \models B$.

The logical consequence relationship requires a *truth-transferring* directed connection from A to B: if A is true in M, then B is also true, but nothing is promised if A is false in M. To paraphrase: from A, B follows by logic alone.

Notice: $A \models B$ if and only if $(A \supset B)$ is a tautology.

This is because both of these relationships, $A \models B$ and $(A \supset B)$ is a tautology, are truth-transferring from A to B.

Our first example has A as the formula $(\forall x)(Fx \vee Gx)$, which says “Everything is either an F or a G”. Take B be the formula $(\forall x)Fx \vee (\exists x)Gx$, which says: “Either everything is an F, or something is a G”. To show that B is a logical consequence of A, we start with an *arbitrary* model M such that A is true, and then tease out the semantics in order to arrive at the conclusion that B is also true in M. As long as we assume nothing about M other than it contains interpretations for all the names and predicates occurring in formulas A and B, then we will show that A logically implies B.

$$(\forall x)(Fx \vee Gx) \models (\forall x)Fx \vee (\exists x)Gx$$

Suppose $M = \langle D, I \rangle$ is a model in which $(\forall x)(Fx \vee Gx)$ is true. Model M includes table with columns for $I(F)$ and $I(G)$, with rows indexed by objects in domain D. Since $(\forall x)(Fx \vee Gx)$ is true in M, each row must contain at least one 1, either for $I(F)$ or for $I(G)$. One possibility: column for $I(F)$ is all 1s. Then $(\forall x)Fx$ is true. Hence $(\forall x)Fx \vee (\exists x)Gx$ is true.

Otherwise, column for $I(F)$ is not all 1s, so must be at least one 1 in column for $I(G)$. Then $(\exists x)Gx$ is true. Hence $(\forall x)Fx \vee (\exists x)Gx$ is true.

Since the model M was an arbitrary model containing $I(F)$ and $I(G)$, we can conclude that $(\forall x)Fx \vee (\exists x)Gx$ is a logical consequence of $(\forall x)(Fx \vee Gx)$.

2.6.2 | LOGICAL EQUIVALENCE

The bi-directional version of logical consequence is the relationship of *logical equivalence*.

Formulas A and B are **LOGICALLY EQUIVALENT** if and only if in every model, A and B have the *same* truth value. This is written $A \equiv B$.

Notice: $A \equiv B$ if and only if $(A \equiv B)$ is a tautology.

This is because both these conditions, $A \dashv\vdash B$ and $(A \equiv B)$ is a tautology, mean there is a two-way bi-directional truth-transferring relationship between A and B.

$$\sim(\forall x)Fx \dashv\vdash (\exists x)\sim Fx$$

Proceed indirectly, to show $\sim(\forall x)Fx \equiv (\exists x)\sim Fx$ is a tautology.

Suppose $M = \langle D, I \rangle$ is a model in which $\sim(\forall x)Fx$ is true and $(\exists x)\sim Fx$ is false. This means $I(F)$ column has at least one 0..., and at the same time, it has all 1s: *impossible!*

Next suppose $M = \langle D, I \rangle$ is a model in which $\sim(\forall x)Fx$ is false and $(\exists x)\sim Fx$ is true. This means $I(F)$ column has all 1s..., and at the same time, it has at least one 0: *impossible!*

Hence $\sim(\forall x)Fx \dashv\vdash (\exists x)\sim Fx$.

2.6.3 | CONTRADICTIONES

The relationship opposite to logical equivalence is that between *contradictiones*.

Formulas A and B are **CONTRADICTIONES** if and only if for every model, A and B have *opposite* truth values.

Notice: A and B are contradictiones if and only if the bi-conditional $(A \equiv \sim B)$ is a tautology.

The general approach to showing A and B are contradictiones is to first suppose M is an arbitrary model in which A is true, and set out to show that B is false in M; then second, suppose that M' is an arbitrary model in which A is false, and set out to show that B must be true in M'. Provided all we assume about the models M and M' is that they contain interpretations of all the names and predicates that occur in formulas A and B, and that A is true in M and false in M', and we can show B is false in M and true in M', then we will demonstrated that A and B are contradictiones.

$(\forall x)(Fx \supset Gx)$ and $(\exists x)(Fx \& \sim Gx)$ are contradictiones.

Suppose model M is such that $(\forall x)(Fx \supset Gx)$ is true. Then in the table with columns for $I(F)$ and $I(G)$, and rows indexed by objects in the domain, in each row in which $I(F)$ is 1, we also have $I(G)$ is

1. This means $(\exists x)(Fx \ \& \ \sim Gx)$ is false.

Suppose instead that model M' is such that $(\forall x)(Fx \supset Gx)$ is false. Then in some row of the table with columns for $I(F)$ and $I(G)$, there is a 1 for $I(F)$ and a 0 for $I(G)$. Hence the formula $(\exists x)(Fx \ \& \ \sim Gx)$ is true in this model M' .

Hence $(\forall x)(Fx \supset Gx)$ and $(\exists x)(Fx \ \& \ \sim Gx)$ are contradictions.

2.6.4 | TRADITIONAL LOGICAL RELATIONS

Traditional logical relationships studied by medieval logicians include those of being *contraries* and *sub-contraries*. As we shall see, these relationships are weaker than being contradictions.

A and B are **CONTRARIES** if and only if there is no model where they are both true.

A and B are **SUB-CONTRARIES** if and only if there is no model where they are both false.

Contraries and sub-contraries are related to contradictions, tautologies and contradictions as follows:

A and B are contraries
if and only if $(A \ \& \ B)$ is a contradiction.

A and B are sub-contraries
if and only if $(A \vee B)$ is a tautology.

A and B are contradictions
if and only if A and B are contraries, and A and B are sub-contraries.

$((\forall x)(Fx \supset Gx) \ \& \ (\exists x)Fx)$ and $(\forall x)(Fx \supset \sim Gx)$ are contraries.

$((\forall x)(Fx \supset Gx) \ \& \ (\exists x)Fx)$ means “All F’s are G’s, and there is at least one F”, while $(\forall x)(Fx \supset \sim Gx)$ means “All F’s are not G’s”.

This contraries relationship can be established by indirect reasoning. Suppose $M = \langle D, I \rangle$ is a model in which both formulas are true; we will show this is impossible. Since $((\forall x)(Fx \supset Gx) \ \& \ (\exists x)Fx)$ is

true in M , we know that at least one row of the table has $I(F)$ value 1, and in each row in which $I(F)$ is 1, we also have $I(G)$ is 1. But since $(\forall x)(Fx \supset \neg Gx)$ is also true in M , we know that in each row in which $I(F)$ is 1, we have that $I(G)$ is 0. This is an impossibility, so we are done: $(\forall x)(Fx \supset Gx) \& (\exists x)Fx$ and $(\forall x)(Fx \supset \neg Gx)$ are contraries but they are *not* contradictories, as they can both be *false*.

2.7 | VALIDITY OF ARGUMENTS

The relationship of *validity* is a truth-transferring directed connection from a set of premises X to a conclusion A .

Let X be a set of formulas, and let A be a single formula.

An argument from premises X to conclusion A is *valid* if and only if in every model M in which each formula in X is true, A is also true in M . We write this as $X \models A$.

In other words, an argument is valid if and only if it is *impossible* for the premises to be true while at the same time the conclusion is false.

We use the same symbol as for logical consequence, and write $X \models A$ to mean the argument from X to A is valid. When X is a finite set of formulas, then the argument from X to A is valid exactly when A is a logical consequence of the *conjunction* of all the finitely-many formulas in X . The notion of validity here is general enough to allow the premises set X to be finite or *infinite*, but since we do not allow infinite conjunctions, we do not define validity directly in terms of logical consequence.

Example (1):

$$(\forall x)(Fx \supset Gx), Fa \models Ga$$

Let $M = \langle D, I \rangle$ be any model for a predicate language including 1-place predicates F and G , and the name a .

Now suppose both the premises $(\forall x)(Fx \supset Gx)$ and Fa are true in the model M . Then $(Fa \supset Ga)$ is true in M , since all instances of $(\forall x)(Fx \supset Gx)$ must be true in M . Since Fa is also true in M , we can conclude by *modus ponens* that Ga must be true in M .

Since M was an arbitrary model, we can conclude that $(\forall x)(Fx \supset Gx), Fa \models Ga$.

Example (2):

$$(\forall x)(Fx \vee Gx), \sim(\forall x)Fx \not\models (\forall x)Gx$$

To show *non-validity* of this argument form, we need to describe a model $M = \langle D, I \rangle$ in which the premises $(\forall x)(Fx \vee Gx)$ and $\sim(\forall x)Fx$ are both true, and the conclusion $(\forall x)Gx$ is false.

Let $D = \{a, b\}$ and consider the following interpretations $I(F), I(G)$:

	$I(F)$	$I(G)$
a	1	0
b	0	1

Then $(\forall x)(Fx \vee Gx)$ is true in M , and $\sim(\forall x)Fx$ is true in M , but $(\forall x)Gx$ is false in M , as required to show non-validity.

Now it is your turn. Answer **YES** or **NO** to each of the following validity or logical consequence questions. To answer **YES**, you need to be able to explain the truth-transferring relationship from the premises to conclusion, while to answer **NO**, you need to describe one model in which the premises are true and the conclusion is false.

Exercise: Answer **YES** or **NO** to each of the following validity or logical consequence questions:

- (a) $(\exists x)(Fx \vee Gx) \models (\exists x)Fx \vee (\exists x)Gx$?
- (b) $(\exists x)Fx, (\exists x)Gx \models (\exists x)(Fx \wedge Gx)$?
- (c) $(\forall x)(Fx \supset Gx), (\exists x)Fx \models (\forall x)Gx$?
- (d) $(\forall x)(Fx \supset Gx), (\exists x)Fx \models (\exists x)Gx$?

(a) answer **YES**. Let M be any model that makes true $(\exists x)(Fx \vee Gx)$. Then there exists an object named d such that either Fd is true or Gd is true. In the first case, if Fd is true, we can conclude that $(\exists x)Fx$ is true in M , and hence the disjunction $(\exists x)Fx \vee (\exists x)Gx$ is true in M . In the second case, if Gd is true, we can conclude that $(\exists x)Gx$ is true in M , and hence the disjunction $(\exists x)Fx \vee (\exists x)Gx$ is true in M , and hence the disjunction $(\exists x)Fx \vee (\exists x)Gx$ is true in M .

(b) answer **NO**. Consider a model M with two objects, a and b , where Fa and $\sim Ga$ are true, while Gb and $\sim Gb$ are true. Then the premises of (b) are both true in M , because $(\exists x)Fx$ and $(\exists x)Gx$ are both true, but the conclusion is false in M as $(\exists x)(Fx \wedge Gx)$ is false.

the conclusion $(\exists x)Gx$ must be true in M , as required. true, we can conclude by *modus ponens* that Gd is true in M . Hence named d such that Fd is true; since the instance $(Fd \supset Gd)$ is also premises $(\forall x)(Fx \supset Gx)$ and $(\exists x)Fx$. Then there exists an object

(d) answer **YES**. Let M be any model that makes true both the

$(\forall x)Gx$ is false.

true, but the conclusion is false in M , as $\sim Gb$ being true shows that of (c) are both true in M , since $(\forall x)(Fx \supset Gx)$ and $(\exists x)Fx$ are both that Fa and $\sim Fb$ and Ga and $\sim Gb$ are true in M . Then the premises

(c) answer **NO**. Consider a model M with two objects a and b such

This brings us to the end of this chapter on the semantics of predicate logic, and it provides a natural segue to the next chapter, on proof trees for predicate logic.

In classifying formulas and relationships between them, we need to reason about *all possible models* of the predicate language at hand, and in general, there are infinitely many different models to consider. This situation for predicate logic stands in contrast with that of propositional logic, where there is a large but still finite number of models – in the form of rows of a truth table. So in predicate logic, there is an even more urgent need for a method for classifying formulas and relationships between them that gives an alternative to direct semantic analysis – provided the proof method can be shown to be *sound* and *complete* for the semantics. Proof trees provide just such a method. So stay tuned for the next chapter.

LIST OF SOME TAUTOLOGIES

As an appendix to the chapter, here is a list of some useful tautologies, many of them bi-conditionals expressing logical equivalence. The 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 patterns in it.

Let A , B and C be any formulas of a predicate logic language. The following are tautologies (true in all models).

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 &:	$\sim(\sim A \vee \sim B) \equiv (A \& B)$
Conditionals for &:	$(A \& B) \supset A$ and $(A \& B) \supset B$
Conditionals for \vee :	$A \supset (A \vee B)$ and $B \supset (A \vee B)$
Contraposition:	$(A \supset B) \equiv (\sim B \supset \sim A)$
Material conditional:	$(A \supset B) \equiv (\sim A \vee B)$
Negated conditional:	$\sim(A \supset B) \equiv (A \& \sim B)$
Weakening:	$A \supset (B \supset A)$
\supset Distribution:	$(A \supset (B \supset C)) \supset ((A \supset B) \supset (A \supset C))$
Pierce's Law:	$((A \supset B) \supset A) \supset A$
<i>Modus ponens</i> as formula:	$(A \& (A \supset B)) \supset B$
Curry's Formula:	$((A \& B) \supset C) \equiv (A \supset (B \supset C))$
Excluding disjuncts:	$((A \vee B) \& \sim A) \supset B$ and $((A \vee B) \& \sim B) \supset A$

Suppose $(\forall x)A$, and $(\forall x)B$ are formulas of a predicate logic language.
The following are tautologies.

Negation of quantifiers:	$\sim(\forall x)A \equiv (\exists x)\sim A$
Negation of quantifiers:	$\sim(\exists x)A \equiv (\forall x)\sim A$
Negation of quantifiers:	$\sim(\forall x)\sim A \equiv (\exists x)A$
Negation of quantifiers:	$\sim(\exists x)\sim A \equiv (\forall x)A$
Distribution of \forall over &:	$(\forall x)(A \& B) \equiv ((\forall x)A \& (\forall x)B)$
Distribution of \exists over \vee :	$(\exists x)(A \vee B) \equiv ((\exists x)A \vee (\exists x)B)$
\forall/\vee one-way conditional:	$((\forall x)A \vee (\forall x)B) \supset (\forall x)(A \vee B)$
$\exists/\&$ one-way conditional:	$(\exists x)(A \& B) \supset ((\exists x)A \& (\exists x)B)$
\forall/\supset one-way conditional:	$(\forall x)(A \supset B) \supset ((\forall x)A \supset (\forall x)B)$
$\exists/\&/\supset$ one-way conditional:	$(\exists x)(A \& B) \supset ((\exists x)A \& (\exists x)B)$

Suppose $(\forall x)A$ and B are both formulas of a predicate logic language (hence $(\exists x)A$ is also a well-formed formula). The following bi-conditionals

(called *prenexing laws*) are tautologies, *provided that* the variable x is either not in B or else is bound in B . (For example, B can be $(\exists x)Fx$.)

Prenexing law for \forall and $\&$:

$$((\forall x)A \ \& \ B) \equiv (\forall x)(A \ \& \ B)$$

Prenexing law for \exists and $\&$:

$$((\exists x)A \ \& \ B) \equiv (\exists x)(A \ \& \ B)$$

Prenexing law for \forall and \vee :

$$((\forall x)A \ \vee \ B) \equiv (\forall x)(A \ \vee \ B)$$

Prenexing law for \exists and \vee :

$$((\exists x)A \ \vee \ B) \equiv (\exists x)(A \ \vee \ B)$$

Prenexing law for \forall in antecedent:

$$((\forall x)A \supset B) \equiv (\exists x)(A \supset B)$$

Prenexing law for \exists in antecedent:

$$((\exists x)A \supset B) \equiv (\forall x)(A \supset B)$$

Prenexing law for \forall in consequent:

$$(B \supset (\forall x)A) \equiv (\forall x)(B \supset A)$$

Prenexing law for \exists in consequent:

$$(B \supset (\exists x)A) \equiv (\exists x)(B \supset A)$$

Suppose C is such that $(\forall x)(\forall y)C$ is a formula of a predicate logic language. The following are tautologies.

$$\text{Commutativity of } \forall \text{ quantifiers: } (\forall x)(\forall y)C \equiv (\forall y)(\forall x)C$$

$$\text{Commutativity of } \exists \text{ quantifiers: } (\exists x)(\exists y)C \equiv (\exists y)(\exists x)C$$

$$\text{Mixed quantifier conditional: } (\exists x)(\forall y)C \supset (\forall y)(\exists x)C$$

$$\text{Mixed quantifier conditional: } (\exists y)(\forall x)C \supset (\forall x)(\exists y)C$$

TREE PROOFS FOR PREDICATE LOGIC

3

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- Proof trees as a proof technique for predicate logic. Their rules for development and completion. Revision of rules for propositional connectives.
- The rules for quantifiers. First, particular rules (negated universal quantifier and existential quantifier, which introduce a name new to the branch), and general rules (universal quantifier and negated existential quantifier, which can be re-used with each name in the branch).
- The rules for determining if a branch is closed, or if it is open, whether it is complete.
- How to use a complete open branch in a tree to construct a model, with a domain and an interpretation for each predicate and name occurring in that branch.
- 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 predicate logic.

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.

SKILLS

These are the skills you will learn to demonstrate.

- 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 model valuation for the language (including a domain and interpretation of all predicates) that makes all formulas on the branch true.
- Detect patterns in relatively simple infinite trees, in order to specify models with infinite domains satisfying a set of formulas.

3.1 | WHY WE NEED PROOF TREES

We've seen in the previous chapter how you can use models to analyse formulas in the language of predicate logic. If a formula—like

$(\exists x)(Fx \supset (\forall y)Fy)$ —is true in all models, then it is a *tautology*. If two formulas—like $(\forall x)\sim Fx$ and $\sim(\exists x)Fx$ —are true in exactly the same models, then they are *logically equivalent*. If any model that makes the premises of an argument jointly true also makes the conclusion true (like the argument below), then that argument form is *valid*.

$$\begin{array}{l} \text{P1. } (\forall x)(Fx \supset Gx) \\ \text{P2. } (\exists x)\sim Gx \\ \hline \text{C. } (\exists x)\sim Fx. \end{array}$$

So much is clear enough, and you can go quite some way to analyse logical properties like being a tautology, or logical equivalence or validity, by directly reasoning about models. The presence of a single model that makes a formula false is enough to show that it isn't a tautology. A single model that separates two formulas (by showing that one is true and the other is false) is enough to show that they are not logically equivalent. A single model acts as a *counterexample* to an argument if it renders the premises true and the conclusion false. Such a model is enough to show that this argument is not valid. Using models to provide counterexamples to these logical properties is relatively straightforward—at least if you have come up with the right model. But using them to show that these properties *hold* is another matter. With two-valued propositional logic, it is straightforward (if a little tedious) to directly reason about models to show that a formula is true in every model. If a formula contains n different atomic propositions, then simply checking the 2^n different combinations of truth values to these atomic propositions is enough complete your task. There is no such reassurance with predicate logic.

Suppose I want to check whether or not $(\forall x)(Fx \vee Gx) \models (\forall x)Fx \vee (\exists x)Gx$. I need to verify that in every model where $(\forall x)(Fx \vee Gx)$ holds, so does $(\forall x)Fx \vee (\exists x)Gx$. How many models would I need to check, if I were to exhaustively list them? Well, first there are all of the models with one element in the domain. Then all of those with two elements in the domain. Then three, ... and ‘after’ having considered all of the models with finitely many objects, we are hardly done! There are all of the models with *infinitely* many objects. This is a *lot* of models to consider. There is no way to conduct an exhaustive search just by listing every model one-by-one. To conclusively show validity, we either need to be *smart* when considering our models—as we did in the last chapter—or we need to do something else. And one such ‘something else’ is our topic of this chapter. We’re going to expand the proof tree technique for propositional logic to construct proof trees for predicate logic. This will provide us a way to conclusively show that an argument is valid in a systematic way, which gives us a rigorous *proof* in the case where an argument is valid, and which (if you’re patient) will generate a counterexample to any invalid argument.

Well, it's not all that simple, especially when n is a large number. But at least 2^n is finite, when n is finite!

Spoiler: this is valid.

Even if we restrict ourselves to *one* possible domain of each size—since replacing one domain D by another D' , in which the identities of the objects are changed but everything else is kept the same, makes no difference to the logical properties we’re evaluating—this still leaves infinitely many possible models.



In the rest of this section, I'll step through some example trees, to motivate the rules for quantified formulas. If you are not already familiar with the rules for trees for propositional logic, it would be good to familiarise yourself with those. Chapter 2 in *Logic: Language and Information 1* is a good place to go for that. Once you're familiar with tree proofs for propositional logic, come back and join us.

★ ★ ★

Welcome back. For our first example, we're going to show that the argument from $(\forall x)(Fx \supset Gx)$ and $(\exists x)\sim Gx$ to $(\exists x)\sim Fx$ is valid, using a tree proof. To do this, we'll start the tree with the premises and the negation of the conclusion, as usual.

$$\begin{array}{l} (\forall x)(Fx \supset Gx) \\ (\exists x)\sim Gx \\ \sim(\exists x)\sim Fx \end{array}$$

Each of these formulas is either a quantified formula (the premises) or a negated quantified formula (the negated conclusion). None of the rules for propositional connectives apply here. But as usual, for each of our steps, the principle will be that each rule will allow us to move from a formula to a *simpler* formula (a *sub-formula*—a disjunct of a disjunction, an antecedent or consequent of a conditional, etc; or a negation of these) such that the results logically follow from the input of the rule, and so that all of the consequences of a formula are enough to reconstruct the starting formula. So, for example, with the conjunction rule $A \ \& \ B$, the outputs of the rule are A and B (both of which are consequences of $A \ \& \ B$, and jointly they are enough to entail $A \ \& \ B$). Similarly, with the conditional rule $A \supset B$, the outputs are two branches, one containing $\sim A$, the other B . From $A \supset B$, neither $\sim A$ nor B follows, but the disjunction (corresponding to the branching options) does follow. And in each branch, $\sim A$ is enough to entail $A \supset B$, and B entails $A \supset B$ too.

We look for the same features for rules for the quantifiers. We start with the negated existential quantified formula $(\exists x)\sim Gx$, and we reason as follows. If this formula is true in some model, then there is some object in the domain of that model to which $\sim Gx$ applies. Let us *introduce* a name for that object. The name a has not been used in any formula in this tree yet, so let us use a to name that object, and given this choice, we can add $\sim Ga$ to the tree, since in any model that makes our starting formulas to be true, so $\sim Ga$ will be true, given our choice for the interpretation of a . So this is the tree:

$$\begin{array}{l} (\forall x)(Fx \supset Gx) \\ (\exists x)\sim Gx \checkmark a \\ \sim(\exists x)\sim Fx \\ | \\ \sim Ga \end{array}$$

We processed the formula $(\exists x)\sim Gx$ and at this stage introduced the name a , so we indicate this with the tick after $(\exists x)\sim Gx$, and written the name a afterwards to show that this is the place at which the name is introduced. The formula $(\exists x)\sim Gx$ is processed with the tick (\checkmark) because the resulting formula $\sim Ga$ is enough to give us $(\exists x)\sim Gx$ back. There is no more information left in $(\exists x)\sim Gx$ that is not present in the $\sim Ga$ we have inferred.

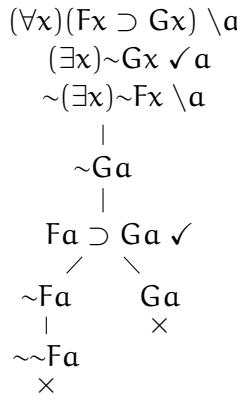
For the next step, we take $(\forall x)(Fx \supset Gx)$. Given that we have considered the object a , the general truth $(\forall x)(Fx \supset Gx)$ applies in particular to a , so we can conclude $Fa \supset Ga$. But this is *one* of the formulas that follow from $(\forall x)(Fx \supset Gx)$. The instance $Fa \supset Ga$ is not enough to recover $(\forall x)(Fx \supset Gx)$, for there may be other objects in the domain, too. If we start talking about b , c or d ,... we need to consider more instances of $(\forall x)(Fx \supset Gx)$. So, we will infer $Fa \supset Ga$, but indicate that we are not necessarily done with $(\forall x)(Fx \supset Gx)$ by marking the a -instance with a backslash, indicating that we may need to revisit this formula again, if circumstances demand it.

$$\begin{array}{c} (\forall x)(Fx \supset Gx) \setminus a \\ (\exists x)\sim Gx \checkmark a \\ \sim(\exists x)\sim Fx \\ | \\ \sim Ga \\ | \\ Fa \supset Ga \end{array}$$

Now we have the formula $Fa \supset Ga$ in the branch, and we can process this in the usual way, given the propositional rules. The second branch, containing Ga will close, and the first remains open.

$$\begin{array}{c} (\forall x)(Fx \supset Gx) \setminus a \\ (\exists x)\sim Gx \checkmark a \\ \sim(\exists x)\sim Fx \\ | \\ \sim Ga \\ | \\ Fa \supset Ga \checkmark \\ / \quad \backslash \\ \sim Fa \quad Ga \\ \times \end{array}$$

The result is the open left branch, which contains $\sim Fa$. But it also contains the unprocessed formula $\sim(\exists x)\sim Fx$. This formula says that there is *no* object x satisfying $\sim Fx$. In particular, it says that a is not an object satisfying $\sim Fx$, so we add $\sim\sim Fa$ to the branch. (In general, $\sim(\exists x)B$ in a branch tells us that there is no object satisfying B , so we can add $\sim B[x := b]$ for any name b in the branch. We are applying this rule here.) The result is a closed branch.



In this process, we similarly did not tick the formula $\sim(\exists x)\sim Fx$, because again, it may apply if we come to consider another object b, c, or whatever. We mark it with a backslash and continue the tree. However, it ends straight away, since $\sim\sim Fa$ closes with $\sim Fa$, and the left branch closes, too. The tree has all branches closed, and this shows that there is *no* way of making the premises true and the conclusion false—the argument is valid.

As another example, let's step through a tree for the argument from $(\forall x)(Fx \vee Gx)$ to the conclusion $(\forall x)Fx \vee (\exists x)Gx$. I said that this argument was valid—let's make good on that claim. We start the tree with the premise and the negation of the conclusion:

$$\begin{array}{c}
 (\forall x)(Fx \vee Gx) \\
 \sim((\forall x)Fx \vee (\exists x)Gx)
 \end{array}$$

The negated disjunction in the conclusion can be processed in the usual way. We get this:

$$\begin{array}{c}
 (\forall x)(Fx \vee Gx) \\
 \sim((\forall x)Fx \vee (\exists x)Gx) \checkmark \\
 | \\
 \sim(\forall x)Fx \\
 \sim(\exists x)Gx
 \end{array}$$

Now we have three unprocessed formulas. The premise, a universally quantified disjunction, and the two negations: a negated universally quantified formula and a negated existentially quantified formula. The universally quantified formula and the negated existentially quantified formulas are of the kind that can be applied again and again, to whatever name we wish to supply them. (We call these *general* formulas.) The negated universally quantified formula $\sim(\forall x)Fx$ will supply us with a new name: a name for the object which doesn't satisfy Fx . So, to supply the tree with a name, we will apply this rule first. We introduce the name a , new to the tree, processing the formula $\sim(\forall x)Fx$.

$$\begin{array}{c}
 (\forall x)(Fx \vee Gx) \\
 \sim((\forall x)Fx \vee (\exists x)Gx) \checkmark \\
 | \\
 \sim(\forall x)Fx \checkmark a \\
 \sim(\exists x)Gx \\
 | \\
 \sim Fa
 \end{array}$$

The tree now contains the name a , and we can apply the general formulas $(\forall x)(Fx \vee Gx)$ and $\sim(\exists x)Gx$ to apply to the object a . So, let's apply the universal quantifier rule to the premise first.

$$\begin{array}{c}
 (\forall x)(Fx \vee Gx) \setminus a \\
 \sim((\forall x)Fx \vee (\exists x)Gx) \checkmark \\
 | \\
 \sim(\forall x)Fx \checkmark a \\
 \sim(\exists x)Gx \\
 | \\
 \sim Fa \\
 | \\
 Fa \vee Ga \\
 / \quad \backslash \\
 Fa \quad Ga \\
 \times
 \end{array}$$

The result is a disjunction, which branches and the left branch, with Fa , closes. The right branch contains Ga . We can process the formula $\sim(\exists x)Gx$, and the result is $\sim Ga$ (since there is no object satisfying Gx). This closes the left branch.

$$\begin{array}{c}
 (\forall x)(Fx \vee Gx) \setminus a \\
 \sim((\forall x)Fx \vee (\exists x)Gx) \checkmark \\
 | \\
 \sim(\forall x)Fx \checkmark a \\
 \sim(\exists x)Gx \setminus a \\
 | \\
 \sim Fa \\
 | \\
 Fa \vee Ga \\
 / \quad \backslash \\
 Fa \quad Ga \\
 \times \\
 | \\
 \sim Ga \\
 \times
 \end{array}$$

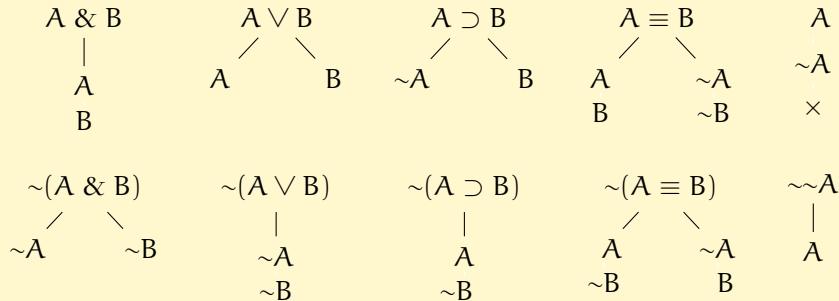
The result is another closed tree. It tells us that there is no way to make the premise true and the conclusion false. The argument is valid.

3.2 | TREE RULES FOR PREDICATE LOGIC

These are two examples which motivate the choices of the rules for tree proofs for predicate logic. In this section, we'll define the rules of

the tree proof system. We'll start by specifying the rules for the propositional connectives:

DEFINITION: The proof tree rules for propositional connectives:



These rules dictate the behaviour of the propositional connectives. The positive and negative connective rules break down each complex formula into simpler components. The closure rule (if a branch contains A and $\sim A$ then it closes— \times) dictates which of the branches are out of action. In propositional logic, a tree for a set of formulas will stay open if and only if that set is satisfiable (if it is true in some evaluation). A complete open branch in a tree will also give us a way to specify an evaluation which makes the starting formulas true. We will define rules for proof trees for predicate logic which also have these features.

As we saw in the previous section, there are two kinds of rules for quantified formulas: the *general* rules, for the universal and for negated existential quantifiers; and the *particular* rules, for the existential and the negated universal quantifiers. We start with the particular rules:

DEFINITION: The *particular* quantifier rules—

$$\begin{array}{ccc} (\exists x)A & \checkmark n & \sim(\forall x)A & \checkmark n \\ | & & | & \\ A[x := n] & n \text{ new} & \sim A[x := n] & n \text{ new} \end{array}$$

Process $(\exists x)A$ by choosing a name n new to the tree, and adding the instance $A[x := n]$ to every open branch in which the starting formula $(\exists x)A$ occurs. Mark this by ticking the formula $(\exists x)A$ and writing the name n to indicate that this is where the name was introduced.

Similarly, process $\sim(\forall x)A$ by choosing a name n new to the tree, and adding the instance $\sim A[x := n]$ to every open branch in which the starting formula $\sim(\forall x)A$ occurs. Mark this by ticking the formula $\sim(\forall x)A$ and writing the name n to indicate that this is where the name was introduced.

Why only mostly? In the rare case where you have only general formulas left, and you have *no* names so far in your tree, then you are permitted to add a new name to substitute into the general formula, to kick things off. Otherwise you would have no names and the quantified formulas would be unprocessed. (Why is this legitimate? Every model has a non-empty domain, so there is always at least one object to be named.)

These rules introduce names to the tree. The general rules, are mostly applied in cases where we substitute a name that is already present in the tree.

DEFINITION: The *general* quantifier rules—

$$\begin{array}{ccc} (\forall x)A \backslash a & & \neg(\exists x)A \backslash a \\ | & & | \\ A[x := a] \text{ any } a & & \neg A[x := a] \text{ any } a \end{array}$$

Process $(\forall x)A$ by choosing any name a not already substituted into this formula, and adding the instance $A[x := a]$ to every open branch in which the starting formula $(\forall x)A$ occurs. Mark this by writing a backslash behind the formula $(\forall x)A$, followed by the name a , to indicate that the name a has now been substituted into the formula.

Similarly, process $\neg(\exists x)A$ by choosing any n not already substituted into this formula, and adding the instance $\neg A[x := a]$ to every open branch in which the starting formula $\neg(\exists x)A$ occurs. Mark this by writing a backslash behind the formula $\neg(\exists x)A$, followed by the name a , to indicate that the name a has now been substituted into the formula.

So, let's summarise the distinctive features of these rules:

- The EXISTENTIAL QUANTIFIER and the NEGATED UNIVERSAL QUANTIFIER rule are *particular* rules.
- You use particular rules *once*, and then “tick” to indicate that they’ve been applied.
- These introduce a new name to the tree, for a *particular* object.
- The UNIVERSAL QUANTIFIER rule, and the NEGATED EXISTENTIAL QUANTIFIER rules are *general* rules.
- These rules can be used *repeatedly*.
- Don’t use a tick when applying these rules; rather, each time the rule is applied, use a backslash and then write the name used in that instantiation.
- They apply *generally* to *all* objects in the domain.

3.3 | DEVELOPMENT OF TREES FOR PREDICATE LOGIC

Given the added complexity of these proof rules, we need now to be quite careful in our account of how to properly put these rules together to form trees. First, remember that a *tree* will be represented as a downward branching structure, starting at its *root*, consisting of formulas, linked by steps, and *branching* downwards (but never *merging*). In the case of propositional logic, proof trees are always finite, and each path downward ends in a *leaf*. In the case of a tree for predicate logic, this

may not be the case. In proof trees, the notion of a *branch* in the tree is very important.

DEFINITION (A BRANCH IN A TREE): A branch in a tree is a single path from the root, downwards, which is totally ordered (for any two different formulas in the branch, one is above the other—they are never side-by-side), and maximal (it is not contained in any larger branch). In a finite tree, a branch is a complete path from the root to a leaf. In an infinite tree, a branch can start at the root and go on without end.

The great power of the proof tree technique is that not only does a closed tree give you a systematic finite *proof* of your target (whether from a tree for $\sim A$ that A is a tautology, or from a tree for a set X of formulas, that X is unsatisfiable, or from a tree for $X, \sim A$ that the argument from X to A is valid), but that a tree with a complete open branch gives you a description of a model—a model which serves as a counterexample of your target. (So, a complete open branch in a tree for $\sim A$ gives a model in which A is false, so it is *not* a tautology; a complete open branch in a tree for a set X of formulas gives you a model in which the members of X are all true, so X is satisfiable; a complete open branch in a tree for $X, \sim A$ gives you a model in which the premises X are true and the conclusion A is false, so the argument from X to A is not valid.) For all of this, we need to be very careful in being clear about when a tree is closed and when it is open, when we have a complete open branch, and how we can use complete open branches to define a model.

The first definition is standard. It is exactly the same as in trees for propositional logic.

DEFINITION (CLOSURE): A branch in a proof tree is *closed* when it contains a formula and its negation. Otherwise, it is *open*. A tree is said to be closed if and only if all of its branches are closed.

Our aim is for a branch to be closed when it is *impossible* to make all of the formulas in that branch true. But the *rule* states that we apply the closure condition in only the case which is simplest to test—the case of an explicit contradiction between A and $\sim A$. This is easy to verify by looking at the branch. Any inconsistencies other than these must be uncovered (say, between $A \& B$ and $\sim(B \& A)$; or between $(\forall x)(Fx \vee Gx)$ and $\sim(\forall x)(Gx \vee Fx)$) by way of the rules for the connectives and quantifiers. Now for how we develop formulas in a tree, from a given starting collection of formulas.

In fact, we could apply the rule in an even more restricted fashion: to only allow a closure between an atomic formula and its negation. This would, in general, make the trees longer, but closure checking would be even more efficient. This is, like many decisions in designing a proof system, a tradeoff. What suits proof trees for use by humans in working things out on a sheet of paper may not suit proof trees for computers to process large amounts of data, and what is ideal for either of those goals may not suit the theorists goal of attempting to understand the basic features of logical concepts. Proof trees are a pretty decent tradeoff between human readability and ease of understanding, efficient automation, and theoretical purity and coherence.

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

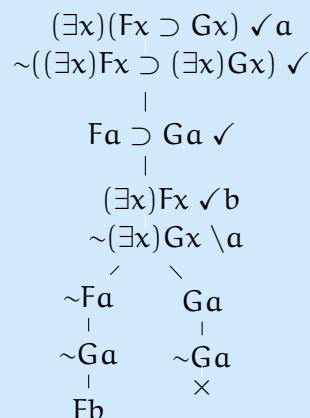
Although we don't do much with it here, the definition allows for cases in which X is a set with infinitely many members. If the tree closes, only finitely many members of X will have been used. If it is open, on the other hand, they may all be processed, and a branch be infinitely long. However, as we will see, there can be infinitely long branches even in the case where we start with a single formula.

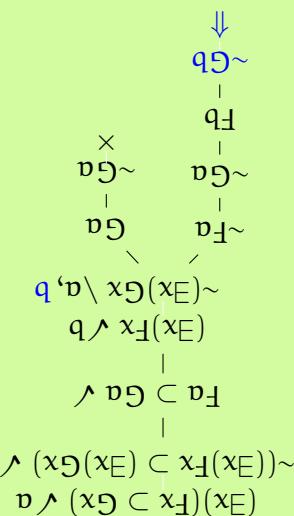
If a tree stays open, we want to use an open branch to make a model. To do this, we develop *every* complex formula, to extract all the information we have in the branch. Only then, will we be sure that there are no hidden inconsistencies waiting to be revealed, and only then will we know that an open branch is genuinely consistent. What is required for every complex formula to be developed? The answer is easy for the propositional connectives (they must be processed) and easy for *particular* rules (they must be processed, introducing a new name). But what about the *general* rules? These allow for repeated use, for substitution with any name that is available.

DEFINITION (COMPLETELY DEVELOPED BRANCHES AND TREES): A *branch* is *completely developed* if and only if every formula in the branch has been processed, and for every name occurring in the branch, that name has been substituted into every *general* formula in the branch. A *tree* is *completely developed* if and only if every branch in that tree is completely developed.

So, in a complete open branch, we substitute *every* name in that branch in each general formula, and we process every other complex formula in the branch. This is enough to generate a model. If the names in the branch name all the things there are, the instances for each general formula are enough to make those general formulas true.

FOR YOU: Is this tree fully developed?





ANSWER: No, it is not. The right branch is closed, but the left branch is not complete, because we had not substituted b in the general formula $(\exists x)Gx$. Once we make that substitution, the branch is complete, because we had not substituted b in the general formula $\sim(\exists x)Gx$.

branch is complete and open.

With complete open branches defined, we can proceed use an open branch in a tree to construct a model. Here is how.

Given a complete open branch in a tree, you can construct a model as follows:

- Its *domain* is the set of names occurring in the branch.
- Interpret each predicate F as follows:
 - If $Fa_1 \dots a_n$ is in the branch, $I(F)(a_1, \dots, a_n) = 1$.
 - If $\sim Fa_1 \dots a_n$ is in the branch, $I(F)(a_1, \dots, a_n) = 0$.
 - If neither occur, you can choose 0 or 1, freely.
 - The branch won't contain both $Fa_1 \dots a_n$ and its negation, $\sim Fa_1 \dots a_n$, as it is open, not closed.
- Interpret each name n in the branch by setting $I(n) = n$.

According to this model, *every formula* in the selected branch is true.

Let's see how this is done in practice:

FOR YOU: Take the tree from the previous example. What is the model that can be constructed from its open branch?

Finally, $I(a) = a$ and $I(b) = b$.

	0	1	b
	0	0	
	<hr/>		a
	<hr/>		I(F) I(G)

as follows:

The branch contains $\neg Fa$, Fb , $\neg Ga$ and $\neg Gb$ so we interpret F and G as follows:

ANSWER: The names in the branch are a and b alone, so $D = \{a, b\}$.

We'll end this chapter by making clear how we can use the tree technique to detect different logical properties.

To test that the argument from X to A is VALID, process a tree for $X, \neg A$. If the tree closes, the argument is valid, and we write $X \vdash A$, and we say we can *prove A* from X , or equivalently, we can *refute $X, \neg A$* .

If the tree stays open, the argument is not valid, and we write $X \not\vdash A$, and we say that there is *no proof* of A from X , and there is *no refutation* of $X, \neg A$. If we use the open branch to construct a model, this open branch gives us a *counterexample* to the argument from X to A , and a way to *satisfy $X, \neg A$* .

To test whether A is a TAUTOLOGY, process a tree for $\neg A$. If the tree closes, the formula is a tautology, and we write $\vdash A$, and we say we can *prove A*, or equivalently, we can *refute $\neg A$* .

If the tree stays open, the formula is not a tautology, and we write $\not\vdash A$, and we say that there is *no proof* of A , and there is *no refutation* of $\neg A$. If we use the open branch to construct a model, this open branch gives us a *counterexample* to formula A , and a way to *satisfy $\neg A$* .

Being a tautology is a special case of argument validity, where the set of premises is empty. We can also consider what happens when we leave out the *conclusion* from an argument, and keep only a premise. $A \vdash B$ iff a tree for $A, \neg B$ closes. $\vdash B$ iff a tree for $\neg B$ closes. In a similar way, $A \vdash$ iff a tree for A closes. This is when A is a contradiction.

To test whether A is a CONTRADICTION, process a tree for A . If the tree closes, the formula is a contradiction, and we write $A \vdash$, and we say we can *refute A*, or equivalently, we can *prove $\neg A$* .

If the tree stays open, the formula is not a contradiction, and we write $A \not\vdash$, and we say that there is *no refutation* of A , and there is *no proof* of $\sim A$. If we use the open branch to construct a model, this open branch gives us a way to *satisfy* the formula A , and a *counterexample* $\sim A$.

With these rules and definitions in hand, we can proceed to some examples.

3.4 | SIMPLE EXAMPLES

We'll start with relatively straightforward examples, illustrating the basic behaviour of the tree rules. First, a tree which will show that $\vdash (\forall x)(Fx \supset (\exists y)Fy)$, that this formula is a tautology. As we mentioned in the last section, if we wish to test whether a formula is a tautology, we start the tree with the negation of the formula.

$$\sim(\forall x)(Fx \supset (\exists y)Fy)$$

The negated universal quantifier is covered by a *particular* rule—one which introduces a new name to the tree. This formula contains no names, so any name would do. Our convention is to start at the beginning of the alphabet, and introduce names in order, so we'll start with ‘a.’ The tree then becomes

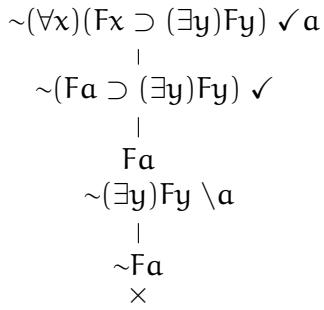
$$\begin{array}{c} \sim(\forall x)(Fx \supset (\exists y)Fy) \checkmark a \\ | \\ \sim(Fa \supset (\exists y)Fy) \end{array}$$

If I think that the tree is going to go on for longer than 5 names, I might shift to another convention, which is to use the names a_1, a_2, a_3, \dots , because by the time I have hit ‘f’, these letters begin to look less like *names* and more like *symbols* for other grammatical categories.

in which we've added a negated conditional. The usual negated conditional rule applies, to give us

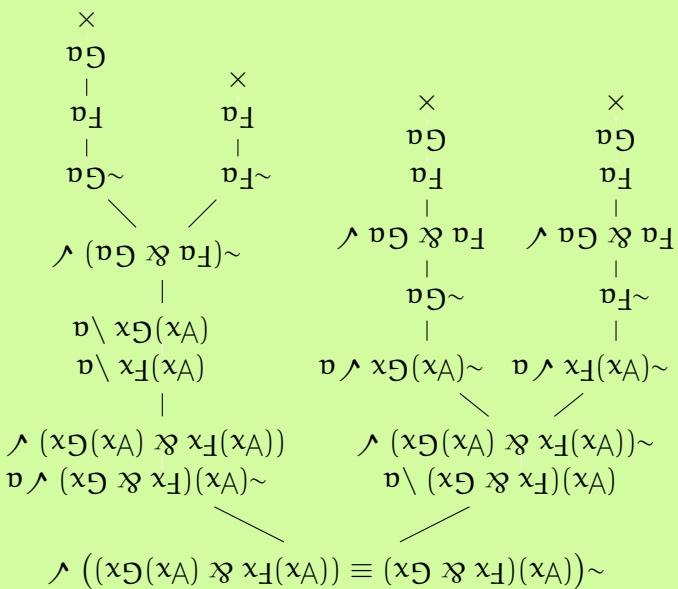
$$\begin{array}{c} \sim(\forall x)(Fx \supset (\exists y)Fy) \checkmark a \\ | \\ \sim(Fa \supset (\exists y)Fy) \checkmark \\ | \\ Fa \\ \sim(\exists y)Fy \end{array}$$

This tree now contains a negated existentially quantified formula, to which a *general* rule applies. We can substitute any name we please into this formula, and the clear candidate is ‘a’, since substituting this name results in a closed tree.



Since the tree closes, we know that the starting formula $\sim(\forall x)(Fx \supset (\exists y)Fy)$ cannot be made true, so $(\forall x)(Fx \supset (\exists y)Fy)$ cannot be made false: it is a tautology, as we had hoped to show.

FOR YOU: Complete a tree, to show that instance of the distribution of \forall over $\&$ law is a tautology: $(\forall x)(Fx \& Gx) \equiv ((\forall x)Fx \& (\forall x)Gx)$.



ANSWER: Here is an example tree. Yours might not look exactly like this, but you should get a closed tree following roughly this pattern.

These two examples have been for closed trees proofs, used to demonstrate that something is a tautology. Let's seen a example of an attempted tree proof that doesn't close. The result will be a complete open branch, and a counterexample. We'll work through a tree, aiming to evaluate the argument from $(\forall x)(Fx \vee Gx)$ to $(\forall x)Fx \vee (\forall x)Gx$. We start with the premise and the negation of the conclusion.

$$\begin{aligned} & (\forall x)(Fx \vee Gx) \\ & \sim((\forall x)Fx \vee (\forall x)Gx) \end{aligned}$$

From here, we can process the negated conclusion—a negated disjunction—using the standard propositional rule:

$$\begin{aligned} & (\forall x)(Fx \vee Gx) \\ & \sim((\forall x)Fx \vee (\forall x)Gx) \checkmark \\ & \quad | \\ & \quad \sim(\forall x)Fx \\ & \quad \sim(\forall x)Gx \end{aligned}$$

Now we have two negated universally quantified formulas. These are processed with particular rules, introducing new names. We will process the first, $\sim(\forall x)Fx$, to introduce the name a , and the formula $\sim Fa$.

$$\begin{aligned} & (\forall x)(Fx \vee Gx) \\ & \sim((\forall x)Fx \vee (\forall x)Gx) \checkmark \\ & \quad | \\ & \quad \sim(\forall x)Fx \checkmark a \\ & \quad \sim(\forall x)Gx \\ & \quad | \\ & \quad \sim Fa \end{aligned}$$

Given the name a , we now have a name to substitute into the universally quantified formula at the top of the tree. We'll make this substitution. The result is a disjunction $Fa \vee Ga$, which we'll process immediately.

$$\begin{aligned} & (\forall x)(Fx \vee Gx) \setminus a \\ & \sim((\forall x)Fx \vee (\forall x)Gx) \checkmark \\ & \quad | \\ & \quad \sim(\forall x)Fx \checkmark a \\ & \quad \sim(\forall x)Gx \\ & \quad | \\ & \quad \sim Fa \\ & \quad | \\ & \quad Fa \vee Ga \checkmark \\ & \quad / \quad \backslash \\ & \quad Fa \quad Ga \\ & \quad \times \end{aligned}$$

You don't have to do this. Why not try the tree yourself, but *don't* substitute a into the first formula yet. Try processing $\sim(\forall x)Gx$ first. How different does your tree look? Is it any longer or any shorter? However it ends up, it should remain open, just like this tree.

The left branch closes, since it contains Fa and $\sim Fa$. The right branch is open, so we will process $\sim(\forall x)Gx$ to introduce a new name, b . (We couldn't use the name a since that would be to assume something that $\sim(\forall x)Gx$ doesn't tell us—that the thing that's not F is also the thing that's not G . It *could* be that there's something that's both not F and not G , but there's no guarantee that there is such an object.) The right branch is open. After we process $\sim(\forall x)Gx$ with b to introduce $\sim Gb$, we'll substitute the name b into $(\forall x)(Fx \vee Gx)$, to introduce another disjunction, which we process, too.

$(\forall x)(Fx \vee Gx) \setminus a, b$	
$\sim((\forall x)Fx \vee (\forall x)Gx) \checkmark$	
$\sim(\forall x)Fx \checkmark a$	
$\sim(\forall x)Gx \checkmark b$	
$\sim Fa$	
$Fa \vee Ga \checkmark$	
/ \	
Fa	Ga
\times	
$\sim Gb$	
$Fb \vee Gb \checkmark$	
/ \	
Fb	Gb
\uparrow	\times

The result is a complete tree. Of the two branches introduced, the right branch closes, since it contains $\sim Gb$ and Gb . The left branch, however, is open. Scanning this branch from top to bottom, you can see that every complex formula has been processed, and the general formula (only the first formula in the tree), has been processed with a and b , all the names in the tree. So, this is a complete open branch.

The indicated branch specifies a model. We take $D = \{a, b\}$, the names in the branch. $\sim Fa$, Ga , $\sim Gb$ and Fb tell us that we can interpret F and G as follows:

	I(F)	I(G)
a	0	1
b	1	0

In this model, you can verify that $(\forall x)(Fx \vee Gx)$ is true (everything is either F or G , true enough) but $(\forall x)Fx \vee (\forall x)Gx$ is false (indeed, not everything is F — a isn't—and not everything is G — b isn't). We have a counterexample to the argument. In this model, the premise is true and the conclusion is false.

In the last tree for this section, we'll see another to test the argument from $(\exists x)(Fx \vee Gx)$ to $(\exists x)Fx \vee (\exists x)Gx$. Here is the tree.

$(\exists x)(Fx \vee Gx) \checkmark a$	
$\sim((\exists x)Fx \vee (\exists x)Gx) \checkmark$	
$\sim(\exists x)Fx \setminus a$	
$\sim(\exists x)Gx \setminus a$	
$Fa \vee Ga \checkmark$	
$\sim Fa$	
$\sim Ga$	
/ \	
Fa	Ga
\times	\times

The tree closes, and as a result we see that the argument is valid. We have $(\exists x)(Fx \vee Gx) \vdash (\exists x)Fx \vee (\exists x)Gx$.

Here are some hints and words of advice for working on trees for predicate logic.

- *Remember to always work on main connectives!* If you see a formula like $(\forall x)(A \vee B)$, you can only work on the universal quantifier, not the disjunction. To attempt to process the disjunction by branching into $(\forall x)A$ and $(\forall x)B$ cases would be to make a mistake, because $(\forall x)A \vee (\forall x)B$ does not follow from $(\forall x)(A \vee B)$.
- If you have a choice on what to process first, process a *linear* rule (one which produces no branches) over one that branches. This minimises the size of the tree (measured by number of formulas), because doing a branching rule before a linear rule typically increases the number of formulas—the results of processing the linear rule are multiplied by the number of branches.
- Process the particular rules before the general rules. Particular rules give us new names, usually associated with distinctive information. These provide the fodder for substitution into general formulas.
- Do the general rules when it looks like making a substitution is *interesting*—when it will give you an interesting formula as a result, typically something that will result in the closure of one or more branches.

Apart from the first point, which is a matter of following the rules correctly, the other rules are at most handy pieces of advice. You are free to pursue the tree rules in any way that suits you. The pieces of advice are simply means to make a tree more manageable and shorter than it might otherwise be. Following them or not is a matter of taste and judgment—not a matter of compliance with the rules of the system.

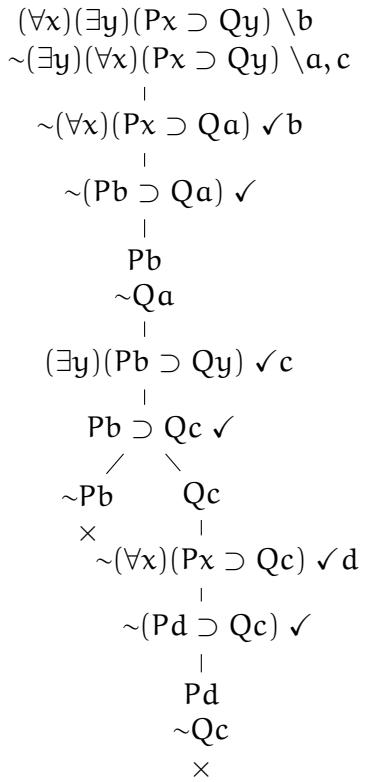
3.5 | COMPLEX EXAMPLES

The trees we have seen so far have been relatively simple. Not all proof trees in predicate logic are so straightforward. Many can be quite complex, and this complexity can arise for two distinct reasons.

- *Bookkeeping*: to do the tree systematically requires quite a lot of processing of general formulas with new names, making keeping track of the different names difficult.
- *Completion*: sometimes it can be hard to ensure that you've completed an open branch (and thus, it can be very hard to know that the branch doesn't close).

Let's look at an example of each kind of complexity. The first is a tree, testing the argument from $(\forall x)(\exists y)(Px \supset Qy)$ to $(\exists y)(\forall x)(Px \supset Qy)$. The tree closes, showing that the argument is valid.

It is a bit surprising that this closes. The slightly more general argument, from $(\forall x)(\exists y)Lxy$ to $(\exists y)(\forall x)Lxy$ is *invalid*.



This is a long and involved discussion.

Be patient with it, and read through it while you attempt to reproduce the proof tree on a blank piece of paper yourself. This will help you understand the point at which each choice is made, when you see each step in the context of the partially developed tree.

Let's step through this tree to see the choices involved. We started, as usual, with the premise and the negation of the conclusion. Both of these are general formulas, but we have no names yet, so we choose a name (we'll start with *a*), so we conclude $\sim(\forall x)(Px \supset Qa)$. The name *a* was introduced at this *general* rule step, so whatever we have learned about *a* is what holds of *every* object whatsoever. We have not learned anything distinctive about a *particular* object as of yet. The object *a* is a nondescript 'every-object.'

Our next step, however, is to process the just introduced $\sim(\forall x)(Px \supset Qa)$. The negated universal quantifier rule is a *particular* rule, so we introduce a new name, *b*, and we derive $\sim(Pb \supset Qa)$, which gives us *Pb* and $\sim Qa$. At this stage the formulas left to process are the two starting formulas. We have two names, *a* and *b* and we have substituted *a* of them into the second formula. This leaves us with three choices remaining for substitutions: *a* or *b* into the first formula, or *b* into the second.

We choose to substitute *b* into the first formula, because it will produce $(\exists y)(Pb \supset Qy)$, and *Pb* is already in our branch—when we get to the underlying $Pb \supset Qy$ (for whatever value is substituted for *y*) the *Pb* is already present, to activate the consequent *Qy*. The other branch (for $\sim Pb$) will close. This is an *interesting* substitution. (Much better than *a* in the first formula, which would leave both branches open, and make the tree more complex.) So we make the substitution, derive $(\exists y)(Pb \supset Qy)$, and process this, with a new name *c*, to give $Pb \supset Qc$. We branch, closing the $\sim Pb$ branch, to leave *Qc*.

Now we have *another* new name, and four remaining substitutions in the two starting formulas: *a* or *c* in the first formula and *b* or *c*

in the second. In this case, we have Qc at hand, so substituting c in the second formula seems most interesting, because we would get $\sim(\forall x)(Px \supset Qc)$, a negated quantified conditional with Qc in the consequent. When we get to process this quantified formula, we'll get a negated conditional with $\sim Qc$ in the consequent which will clash with the Qc we already have. So we make this substitution, to derive $\sim(\forall x)(Px \supset Qc)$.

To process $\sim(\forall x)(Px \supset Qc)$, we introduce yet another new name, since this is also a particular rule. The result is $\sim(Pd \supset Qc)$, which we process into Pd and $\sim Qc$. The $\sim Qc$ closes the tree, given that the branch already contains Qc .

This is quite a small tree, all things considered, but there were many other ways it could have been developed, given the choices of substitutions one could make into the starting formulas, many of which would generate new names, and even more formulas to process. However, all trees would eventually close. No tree for this argument can have a complete open branch.

That was one example of a tree which closed, but which required reasonably careful attention to the choice of substitution for names in formulas to make the tree of manageable size. In the next example we'll see a proof tree which does not acquiesce to this sort of treatment. This proof tree is infinitely long, no matter what substitutions you make, and no matter in what order you choose to process the formulas. This is a very simple tree, starting with the formula $(\forall x)(\exists y)Lxy$. It is a tree, designed to test whether this formula is a contradiction or not. (If the tree closes, it is a contradiction. If it stays open, an open branch can be used to construct a model in which the starting formula is true.) We start the tree with $(\forall x)(\exists y)Lxy$. It's a general formula, but we have no names in the branch, so we start with a name a_1 , and continue.

$$\begin{array}{c}
 (\forall x)(\exists y)Lxy \setminus a_1, a_2, a_3 \\
 | \\
 (\exists y)La_1y \vee a_2 \\
 | \\
 La_1a_2 \\
 | \\
 (\exists y)La_2y \vee a_3 \\
 | \\
 La_2a_3 \\
 | \\
 (\exists y)La_3y \\
 \vdots
 \end{array}$$

This gives us $(\exists y)La_1y$, which introduces a new name a_2 , and La_1a_2 , and this new name is substituted into the starting formula $(\forall x)(\exists y)Lxy$. This gives us $(\exists y)La_2y$, which gives us a new name a_3 and the formula La_2a_3 , and a_3 can be substituted into the starting formula. And you can see the pattern. (And you can perhaps see why I started with a_1 , a_2 , a_3 , etc. rather than a , b , c , etc.)

If you were not following the choices we'd made already, and our reasons for them, you might be quite worried at the number of names that are being introduced, because this multiplies the number of substitutions to make, and there may be no end in sight

Why do we say this so confidently? Because the soundness and completeness theorem result, proved in the next section, tells us that a tree for a set of formulas has a complete open branch if and only if that set has a model. This tree closes for the premise and negated conclusion, so the the argument is valid. It has no counterexample, so there is no way to make a tree stay open, yet have a complete open branch.

It should be obvious: this formula is satisfiable—choose a model with one object in the domain, and let L relate it to itself. Voila. But the idea here is to see what the tree technique does with this formula.

The tree is unending. Any complete open branch for this tree (and there is one, and only one) is infinitely long. If we look at the literals (atoms or negated atoms) occurring in the this branch, we can see the very simple pattern that recurs:

$$La_1 a_2 \ La_2 a_3 \ La_3 a_4 \ La_4 a_5 \dots$$

The domain that arises from this open branch is infinite: we have $D = \{a_1, a_2, a_3 \dots\}$. Using our conventions to read an interpretation of L from this open branch, we have

$I(L)$	a_1	a_2	a_3	a_4	a_5	\dots
a_1	1					
a_2		1				
a_3			1			
a_4				1		
\vdots						\ddots

This table could be filled out in many ways. Any way of doing so would be enough to make the formula true, because for any choice of x (any row) there is some choice of y (some column) where Lxy is true. And indeed there is. This formula is true. It would still be true if the blanks in this table were zeroed out

$I_0(L)$	a_1	a_2	a_3	a_4	a_5	\dots
a_1	0	1	0	0	0	
a_2	0	0	1	0	0	
a_3	0	0	0	1	0	
a_4	0	0	0	0	1	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

...or if the table were uniformly ones ...

$I_1(L)$	a_1	a_2	a_3	a_4	a_5	\dots
a_1	1	1	1	1	1	
a_2	1	1	1	1	1	
a_3	1	1	1	1	1	
a_4	1	1	1	1	1	
\vdots	\vdots	\vdots	\vdots	\vdots	\vdots	\ddots

... or anything in between. In all of these models, the formula turns out to be true. We have constructed a whole family of models, any of which is enough to satisfy our formula. To sum up what we have seen so far:

- The tree never ends at any step.
- The complete tree is *infinite*.
- The complete tree gives us an interpretation.
- $D = \{a_1, a_2, a_3, \dots\}$

- $\text{La}_1 \alpha_2, \text{La}_2 \alpha_3, \dots$ are all true.
- In any interpretation like this, $(\forall x)(\exists y)Lxy$ is true.

However, this seems very wasteful. $(\forall x)(\exists y)Lxy$ is true in many other models, many of which don't require an unending, infinitely large domain of objects. Not as much is needed to show how to make $(\forall x)(\exists y)Lxy$ true.

If we look at the original information we have about the interpretation of L from the open branch:

$I(L)$	α_1	α_2	α_3	α_4	α_5	\dots
α_1		1				
α_2			1			
α_3				1		
α_4					1	
\vdots						\ddots

we see something quite curious. There is *nothing* here that tells us that α_1 and α_2 have to be different objects. If we *identified* α_1 and α_2 —by replacing them by the one object α_* that was related to anything that to which α_1 or α_2 were related, and such that something related to α_* if and only if it related to α_1 or to α_2 —the result would be this:

$I(L)$	α_*	α_3	α_4	α_5	\dots
α_*	1	1			
α_3			1		
α_4				1	
\vdots					\ddots

We have α_* related to α_* since α_1 was related to α_2 in the old model, and α_* is related to α_3 since α_2 is related to α_3 in the old model. This model does just as well at making $(\forall x)(\exists y)Lxy$ true. But we don't need to stop there. There's nothing stopping us identifying α_3 with α_* . Or α_4 as well. Or *everything*. The result is much simpler.

$I(L)$	α_*
α_*	1

And this is also a model (where $D = \{\alpha_*\}$) in which $(\forall x)(\exists y)Lxy$ holds. We have simplified the model delivered to us by the open branch by aggressively revisiting the assumption that different names name different things (the assumption made when we take the domain to simply *be* the class of names), by identifying as many different objects as is consistent with the constraints on the interpretation.

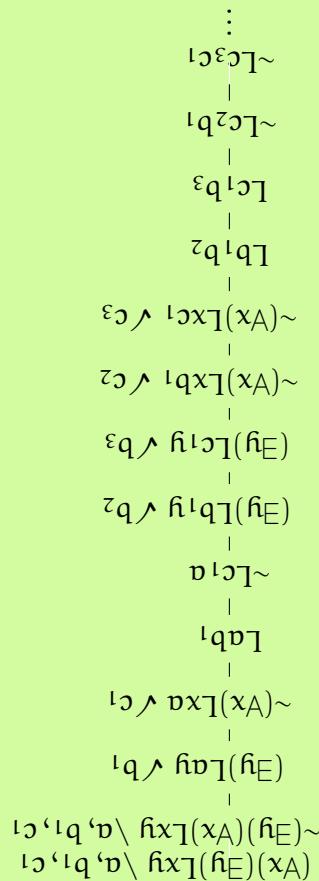
So, here is what we've found:

- Trees can go on *forever*.
- Even then, a complete open branch describes a model.
- Sometimes the model you get is bigger than you need—there might be a smaller model that does the job.

How could an identification *not* work?
If we had $I(L)$ assigning 1 to the pair $\langle \alpha_1, \alpha_2 \rangle$ but 0 to the pair $\langle \alpha_1, \alpha_3 \rangle$, then α_2 and α_3 could *not* be identified into one object α_* , because we couldn't have $I(L)$ assign both 1 and 0 to the pair $\langle \alpha_1, \alpha_* \rangle$.

- A *name* in an open branch names an *object* in the domain.
- The safe and simple assumption is that each name names a different thing. (Each name is *standard*.) This is safe (you never go wrong) but sometimes expensive (the models can get *big*).
- The tricky alternative is that names can sometimes name the same thing. This requires brains (to figure out what works) but is more economical.

FOR YOU: Process a tree for the argument from $(\forall x)(\exists y)Lxy$ to $(\exists y)(\forall x)Lxy$. It should have an infinitely long open branch. Find the pattern in this open branch and use this to describe a model, whose domain is made up of each of the names in the branch, which makes the premise true and the conclusion false. Then see if you can simplify that model into a finite one.



ANSWER: Your tree should not close. My tree goes like this:

In this tree, we use *a* for our starting name, *b_n* for names substituted into $(\exists y)Lty$ formulas, and *c_n* for names substituted into $\sim(\forall x)Lxt$ formulas.

in which we still have every row containing a 1 and every column containing a zero.

0	1	a
1	0	p
a	p	(1)I

A simpler model that does the same job would be

second column, etc.).

a	b_1	c_1	b_2	b_3	c_2	c_3	\dots
$I(L)$	1	1	1	0	1	0	\vdots
a	b_1	c_1	b_2	b_3	c_2	c_3	\vdots
a	b_1	c_1	b_2	b_3	c_2	c_3	\vdots
a	b_1	c_1	b_2	b_3	c_2	c_3	\vdots

this:

This kind of simplification, in which an infinitely large model is compressed into a finite one, is *often* possible, but it is not always available to us. There are some single formulas (or finite sets of formulas) that are satisfiable, but are *only* satisfiable in an infinite model.

Can you find any such formulas or sets of formulas?

3.6 | SOUNDNESS AND COMPLETENESS

We have seen two different ways to spell out the notion of validity. An argument is valid *according to models* if it has no counterexample (no model making the premises true and the conclusion false). An argument is valid *from the point of view of proofs* if a proof tree for that argument closes. In this section we will *show* that these two definitions of validity actually draw the boundary between validity and invalidity in the same place. This is called the *soundness* and *completeness* theorem for the tree proof technique. The soundness theorem shows that proofs



Kurt Gödel (1906–1979)
with Albert Einstein

never declare an argument valid that models don't also say is valid. (If $X \vdash A$ then $X \models A$.) The completeness theorem is the converse. It says that whenever models declare something valid (whenever there is no counterexample) then there is some proof of that fact. (If $X \models A$ then $X \vdash A$.) This is harder to show, because proofs are finite, while models may be infinite. The mere absence of a model making X true and A false does not seem like much upon which to build a proof from X to A . Yet, as a matter of fact, this result holds. The logician Kurt Gödel (1906–1979) first formulated and proved a completeness theorem for predicate logic in the 1930s. Clearly formulating and proving the completeness theorem is one of the intellectual highpoints of the 20th Century—it started off the whole domain of what we call ‘metatheory’ or ‘metalogic’, the reflection *upon* different logical techniques. It requires the realisation that we are defining related concepts (provability, validity) in different ways, and therefore that it's an open question as to whether they coincide. As a matter of fact, they do, and this fact is one of the great strengths of predicate logic and its proof systems.

As in the propositional case, our first step is will be to simplify what we'll try to show. Remember $X \vdash A$ if and only if $X, \neg A \vdash$ —if a tree for X and $\neg A$ closes. We'll use the same sort of notation for \models , validity defined in terms of models. We'll say that $X \models$ if and only if there is no model that makes every member of X true. Then it follows that $X \models A$ if and only if $X, \neg A \not\models$.

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

3.6.1 | SOUNDNESS: IF $x \not\models$ THEN $x \not\vdash$

The first fact we'll show is the *soundness* fact. That is, if there is a model that makes X true, then a tree for X stays open. This is not difficult to prove. As in the case with propositional logic, we'll start with a model M that makes X true. We'll call a set of formulas *safe* if every member of that set is true according to M . The starting set X is safe, because that's how we started with M . We'll show that any tree for X will have an entire *branch* that is safe, which means that this branch will stay open. (Why? No *closed* branch is safe, since M 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. For the propositional rules:

$$\begin{array}{cccc}
 A \& B & A \vee B & A \supset B \\
 | & / \quad \backslash & | & / \quad \backslash \\
 A & A \quad B & \sim A & B \\
 & B & & \sim B
 \end{array}
 \quad
 \begin{array}{ccccc}
 A \equiv B & & & & \sim A \\
 / \quad \backslash & & & & | \\
 A & B & \sim A & B & A
 \end{array}$$

$$\begin{array}{ccccc}
 \sim(A \& B) & \sim(A \vee B) & \sim(A \supset B) & \sim(A \equiv B) & \sim\sim A \\
 / \quad \backslash & | & | & / \quad \backslash & | \\
 \sim A & \sim A & A & \sim B & A \\
 & \sim B & \sim B & B &
 \end{array}$$

the argument is exactly as it was for propositional logic. Extending a branch with one of these rules will always result in at least one branch that continues to be safe. What is new is the rules for the *quantified formulas*.

Let's start with the *general* quantifier rules—

$$\begin{array}{ccc}
 (\forall x)A \ \backslash a & & \sim(\exists x)A \ \backslash n \\
 | & & | \\
 A[x := a] \text{ any } a & & \sim A[x := a] \text{ any } a
 \end{array}$$

If a formula $(\forall x)A$ is in our branch and it is safe, this means that $(\forall x)A$ is true in our model M . So if we extend that branch with an instance $A[x := a]$ of A , no matter what the name a , this formula $A[x := a]$ is true in M too, and so the extension to this branch remains safe. (Every instance of A is true, if $(\forall x)A$ is true in the model M). Similarly, if $\sim(\exists x)A$ is in our branch, and is safe, this means that $\sim(\exists x)A$ is true in our model M , so $(\exists x)A$ is false in M and as a result, no instance $A[x := a]$ of A is true in M . This means that for whatever name a we substitute into the original formula, the resulting formula $\sim A[x := a]$ will hold in our model M , so this extension to the branch is safe, too. Extending a branch with a *general* rule extends the safety zone.

Now consider the *particular* quantifier rules—

$$\begin{array}{ccc}
 (\exists x)A \ \checkmark n & & \sim(\forall x)A \ \checkmark n \\
 | & & | \\
 A[x := n] \ n \text{ new} & & \sim A[x := n] \ n \text{ new}
 \end{array}$$

If a formula $(\exists x)A$ is in our branch and it is safe, this means that $(\exists x)A$ is true in our model M , so there is some object d in the domain such that $A[x := d]$ holds (where d is the standard name of the object d). So if we extend that branch with an instance $A[x := a]$ of A where a is a new name, we can interpret this formula as holding in the model, because the interpretation of this name a is unconstrained by anything else. (It is *new*—it is free for us to interpret how we like). So let's extend the interpretation to the expanded language, by setting $I(a) = d$, the object for which $A[x := d]$ is true. It follows that the formula $A[x := a]$ is true in M too (since $I(a) = d$, and $A[x := d]$ is true), and so the extension to this branch remains safe. In just the same way, if $\sim(\forall x)A$ is in our branch, and is safe, this means that $\sim(\forall x)A$ is true in our model M , so

$(\forall x)A$ is false in M and as a result, there is some instance $A[x := d]$ of A , which is false in M . This if we interpret the new name a to denote d , the resulting formula $\sim A[x := a]$ will hold in our model M , so this extension to the branch is safe, too. Extending a branch with a *particular* rule also extends the safety zone.

These are all of the rules. Whenever you start with X and a model making every element in X true, then whenever you develop the tree, at least one branch remains safe, and therefore, the tree stays open. So, if $X \not\models$ then $X \not\models$.

3.6.2 | COMPLETENESS: IF $x \not\models$ THEN $x \not\models$

We want to show the converse, if $X \not\models$ then $X \not\models$ —that if a tree for X stays open, then there is some model which makes X true. If a complete tree for X is open, then we construct a model from the names and literals on the branch. We choose the domain to consist of all of the names, and interpret a predicate R as having value 1 for all of the tuples $\langle m_1, \dots, m_n \rangle$ where $Rm_1 \dots m_n$ is in the branch, and having value 0 for all of those tuples where $\sim Rm_1 \dots m_n$ is in the branch. This is consistent because the branch 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. The argument for the propositional rules is exactly the same as for propositional logic, so we will consider that done. We now look at the quantifier rules. This time, we start with the *particular rules*.

$$\begin{array}{c} (\exists x)A \quad \checkmark n \\ | \\ A[x := n] \quad n \text{ new} \end{array} \qquad \begin{array}{c} \sim(\forall x)A \quad \checkmark n \\ | \\ \sim A[x := n] \quad n \text{ new} \end{array}$$

Here, if a model makes true $A[x := n]$ for some name n , then it follows that the model makes $(\exists x)A$ true, by *definition*. (The name n is the standard name of the object n in the domain). Similarly, if a model makes true $\sim A[x := n]$ for some name n , the model does not make $(\forall x)A$ true, so it makes its negation true. For particular rules, the step from output back to input is straightforward. Let's turn to the *general rules*, which are more complex.

$$\begin{array}{c} (\forall x)A \setminus a \\ | \\ A[x := a] \quad \text{any } a \end{array} \qquad \begin{array}{c} \sim(\exists x)A \setminus a \\ | \\ \sim A[x := a] \quad \text{any } a \end{array}$$

In a complete open branch, if we have a general formula like $(\forall x)A$, it follows that *every* name in the branch has been substituted into the formula. This means that the branch contains $A[x := a]$ for *every* name a in the branch. But this means that the branch contains $A[x := a]$

for every object a in the domain of our model, and so, that $A[x := a]$ is true in the model, for every object a . It follows that each instance $A[x := a]$ of $(\forall x)A$ is true in M , which suffices to show that $(\forall x)A$ is also true in M , as desired.

Similarly, if we have $\sim(\exists x)A$ in our branch, it follows that *every* name in the branch has been substituted into the formula. This means that the branch contains $\sim A[x := a]$ for *every* name a in the branch. But this means that the branch contains $\sim A[x := a]$ for every object a in the domain of our model, and so, that $A[x := a]$ is false in the model, for every object a . It follows that each instance $A[x := a]$ of $(\exists x)A$ is false in M , which suffices to show that $(\exists x)A$ is also false in M , which makes $\sim(\exists x)A$ true in M as desired.

It follows that if we have a complete open branch in a tree for X —that is, if $X \not\vdash$ —then there is a model that makes everything in that branch true, so it makes the starting formulas X true, and hence $X \not\models$. We have proved *completeness*: if $X \not\vdash$ then $X \not\models$.

Tree proofs are a useful, powerful and elegant proof system, with a long heritage [1]. For an important introduction to proof trees, read Smullyan's *First-Order Logic* [11]. For a more recent introduction of logic using trees, consult Howson's *Logic with Trees* [5]. The tree proof technique makes the soundness and completeness argument easier than it is for other proof systems. Trees are a helpful proof technique, because an *unsuccessful* proof explicitly provides a model. For other proof techniques, like natural deduction or Hilbert proof systems, completeness can be much harder to show.

IDENTITY AND FUNCTIONS

4

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- Identity as a distinguished predicate, with semantics already available within models of predicate logic.
- Use of identity and negated identity in common counting quantifier constructs such as “at least two” or “exactly nine”.
- Additional tree rules for identity (one positive and one negative).
- First-order logic = predicate logic + identity + function symbols; complexities involved in adding function symbols.
- Extensions of first-order logic: multi-sorted logic, higher-order quantification, and non-denoting terms.

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 practise your skills and get familiar with each of these concepts.

SKILLS

These are the skills you will learn to demonstrate.

- Use of identity and negated identity to express sameness and difference, and to express common counting quantifier constructs such as “at least two” or “exactly nine”.
- Given a predicate logic formula with identity, describe a model in which the formula is true.
- Develop a proof tree for a formula or argument form with identity.

4.1 | IDENTITY: EXPRESSIVE POWER

At this stage, having on hand the syntax, semantics and proof tree method for predicate logic, we are equipped to analyse and formalise quite a lot of natural language sentences and arguments. It is certainly a great deal more than we can express within propositional logic. However, as we shall explore in this lesson, there are some things that *cannot* be expressed within predicate logic. This limitation of expressive power concerns *identity* – when two different names name the same object, and *difference* – when two similar objects are not the same. In this and the next few lessons making up this section of the course, we will see how to extend predicate logic – its syntax, semantics and tree

method – in order to be able to express notions of identity and difference.

So let's begin with a question of identity. Who is this singer? Who is this rock star, sporting classic 1980's big hair? Some clues: the year is 1983; the band is U2.



Yes, indeed. This is *Bono*, the lead singer and song-writer for the Irish rock band, U2. The band U2 has been together for now almost 40 years, with multiple generations of fans, and Bono remains a high-profile activist for social causes. So 17 year olds and 70 year olds are equally likely to have heard of U2 and of Bono.

Now, in case you didn't know, *Bono* is not the name he was given by his parents at birth. His birth certificate has the name *Paul Hewson*. Full name: Paul David Hewson, born 10 May 1960, in Dublin, Ireland. So a very obvious thing to assert is the identity sentence: Paul Hewson *is* Bono.

Let's examine the *logical form* of some identity and difference assertions.

- There are two guitarists in U2.
- Paul Hewson is Bono.
- Bono is not Dave Evans.

Start with [There are two guitarists in U2](#). We could try:

$$(\exists x)(\exists y)((Gx \ \& \ Ux) \ \& \ (Gy \ \& \ Uy))$$

where the 1-place predicate G stands for “is a guitarist” and the 1-place predicate U stands for “is in the band U2”. This formula says that there exist x and y who are both guitarists in U2. But the problem is that this formula would still be true even if there was only *one* guitarist in the band. The main guitarist in U2 is the fellow called *The Edge* and we can instantiate both the x variable and the y variable with the same name, say e for *The Edge*, to give $((Ge \ \& \ Ue) \ \& \ (Ge \ \& \ Ue))$. So we are not there yet.

The lead singer Bono is sometimes the second guitar player, so we'd like to express that he is *different* from the fellow called The Edge. The predicate logic we have seen so far does *not* allow us to say this.

The new type of expression we need to add on is: $(x \neq y)$, which means the same as the negation $\sim(x = y)$.

$$(\exists x)(\exists y)((Gx \ \& \ Ux) \ \& \ (Gy \ \& \ Uy) \ \& \ (x \neq y))$$

The form $x \neq y$ expresses that x is not identical to y , so it lets us say that there are two different guitarists in the band U2. To show that this new sentence is true, we have to instantiate two names, say e and b , where the names denote *different* objects in the domain.

Difference is the negation of an identity assertion. How about we go back to our basic fact that [Paul Hewson is the same person as Bono](#). Again, the predicate logic we have seen so far does not allow us to say this. The “*is*” relation here means “*is identical to*” or “*denotes the same object*.” Paul Hewson and Bono are two different names for the same person. The new type of expression we need to add is:

$$(p = b)$$

saying p is identical to b .

The last example is a negated identity assertion: [Bono is not Dave Evans](#). The guitarist’s mother did not name him *The Edge* when he was born; rather, his real name is Dave Evans, and this fellow is not the same as Bono, or Paul Hewson. To express this, we write:

$$(b \neq d)$$

to say b is not identical to d .

4.2 | IDENTITY: SYNTAX AND MODELS

4.2.1 | THE SYNTAX OF IDENTITY

First, the syntax. Identity is a 2-place or binary predicate, so we need two names together with the predicate symbol to put together an atomic formula. Identity as a special predicate has different conventions for writing it.

- *Identity is a binary predicate.*

YES Write “ a is identical to b ” as “ $(a = b)$ ”

NO Not as “ $= ab$ ”.

YES Write the negation of “ $(a = b)$ ” as “ $(a \neq b)$ ”

NO Not as “ $\sim(a = b)$ ”.

We do not write the identity predicate symbol first, followed by the two names, as we would with a regular binary predicate, because, well: it looks weird. Identity has historically been written, and is most naturally written, in *infix* form, with the identity predicate symbol *between* the two names. In logic, we do *not* say “ a is equal to b ”, because that could be understood as meaning “equal in some respects but not others”. Identity assertions express the stronger notion that two names denote the exact same object.

For negated identity assertions, we also use the infix notation. We write the non-identity symbol \neq in between the two names, and read

as: “ a is not identical to b ”, or “ a is different from b ”. This is to be understood as approved shorthand for the formula $\sim(a = b)$, it is not the case that a is identical to b . More precisely, the formula $(a \neq b)$ is understood as shorthand or an abbreviation for formula $\sim(a = b)$, so that the semantics of $(a \neq b)$ will be identical to the semantics of $\sim(a = b)$.

Let’s review the formation rules for predicate logic, and then see how to add on one more clause for identity atomic predicates.

- If F is a predicate_n and a_1, \dots, a_n are names, then $Fa_1 \dots a_n$ is a formula.
- If A and B are formulas, then so are $\sim A$, $(A \& B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$.
- If A is a formula and a is a name, $A[a := x]$ is the result of replacing all occurrences of a in A by x .
- If A is a formula, a is a name, and x is a variable, then $(\forall x)A[a := x]$ and $(\exists x)A[a := x]$ are formulas.
- **If a and b are names, then $(a = b)$ is a formula.**

With the shorthand convention that $(a \neq b)$ is an abbreviation of $\sim(a = b)$, we do not need to add a separate clause for non-identity.

Here’s some guidance about how to read identity claims.

- Do not read “ $(a = b)$ ” as “ a equals b .”
!! “Alice equals Bob” is ungrammatical!
 - (It make sense to say that Alice is equal to Bob in *height*, or *status* or something else.)
- But $(a = b)$ means that a and b are *equal in all respects*, since a is *the same thing* as b .

Example: reading identity formulas: To make the reading a bit more concrete, let’s suppose the binary predicate P_{yx} means “ y is as poor as x ”.

- $(\forall x)(\exists y)(\exists z)(P_{yx} \& P_{zx} \& y \neq z)$
This says: For every x , there exists y and z , different from each other, both of whom are as poor as x . More informally: Everyone has at least two people as poor as them.
- $(\exists y)(\forall x)(x = y)$
This says: There exists an object y such that, no matter how you choose an x from the domain of the model, x is identical to y . The only way this can happen is when: The model contains exactly one object.

Remember that the domain of a model must be non-empty, so it has at least one object. If the formula $(\exists y)(\forall x)(x = y)$ is true in a model, then the model must contain exactly one object.

Which of the following formulas says that there is at most one object with property F?

- (a) $(\forall x)(\forall y)(Fx \vee Fy)$
- (b) $(\forall x)(\forall y)((Fx \wedge Fy) \supset x = y)$
- (c) $(Fa \wedge Fb) \supset a = b$
- (d) Fa
- (e) $(\exists y)(\forall x)(Fx \supset x = y)$

Be careful: “*at most one* object with property F” means we need either zero or one object(s) with property F, but definitely not two or more objects with property F.

The correct answers are (b) and (e). Formula (b) can be read as saying: if there were two objects with property F, then they would be the same. This formula is true exactly when there are 0 or 1 objects with property F. The other correct answer is formula (e), which says there exists an object with the property that if there is anything with property F, then that thing is identical to y. So this object y does not necessarily have property F, because there might be nothing with property F, but if y does have property F, then nothing else different from y can have property F. So this formula is also true exactly when there are 0 or 1 objects with property F.

Note that formula (c) says if names a and b are of objects with property F, then they name the same thing, but formula (c) does not rule out there is another object different from both a and b, say with name c, such that Fc is true.

4.2.2 | MODELS: INTERPRETING IDENTITY

Our next task is to make precise the semantics of identity added to predicate logic. The extra thing we have to do to interpret identity is .. exactly nothing! We already have the ingredients within any model to interpret identity. Recall a model M is a pair consisting of a domain D and an interpretation I of the names and predicates. The atomic formula $(a = b)$ is true in a model M if and only if the object $I(a)$ interpreting name a is identical to the object $I(b)$ interpreting name b: they are the same object. Conversely, using our shorthand convention, a negated formula $(a \neq b)$ is true in a model M if and only if the object interpreting name a is different from the object interpreting name b.

Given model $M = \langle D, I \rangle$,

$(a = b)$ is true in M if and only if $I(a) = I(b)$.

Let us return to the our long-running favourite example of a model M whose domain D has three objects: Ada Lovelace, Kurt Gödel and Albert Einstein. In our model M , the 2-place predicate symbol A stands for the verb “admires”, as in Axy means “ x admires y ”. In the model M , suppose the names a and k denote Ada Lovelace and Kurt Gödel, respectively.



Model $M = \langle D, I \rangle$, where:

		$I(A)$	l	g	e
$D = \{l, g, e\}$	$I(a) = l$	l	1	0	0
	$I(k) = g$	g	1	0	1
		e	1	1	1

- Everyone admires someone else.
 $(\forall x)(\exists y)(Axy \ \& \ x \neq y)$
- If you are admired by everyone, you’re Ada.
 $(\forall x)((\forall y)Ayx \supset (x = a))$

Looking at the table $I(A)$ interpreting the “admires” predicate A , the first row of the table says Ada Lovelace admires herself but not the other two, which makes sense because she died in 1852, well before the time of Gödel and Einstein. The second row says that Kurt Gödel admires Ada Lovelace and Albert Einstein, but not himself, which is probably true: Gödel was a very troubled genius with doubts and anxieties. And the last row of the table says Albert Einstein admires everyone: Ada, Kurt and himself.

Let’s look at the truth or falsity of some formulas in this model. First, the formula expressing that everyone admires someone else:

$$(\forall x)(\exists y)(Axy \ \& \ x \neq y)$$

This is a universally quantified formula, so for it to be true, we would have to verify that each of the three instances are true in this model. But

the first instance for Ada Lovelace is false: there does not exist anyone different from Ada Lovelace who is admired by Ada Lovelace. So the formula is false in this model.

The second formula expresses that if you are admired by everyone, then you are Ada Lovelace:

$$(\forall x)((\forall y)Ayx \supset (x = a))$$

Another universally quantified formula so we have to verify that each of the three instances are true in this model. First, the instance for Ada Lovelace is true, looking down the first column of the table interpreting predicate A . The antecedent of the conditional expands out as the conjunction:

$$(All \ \& \ Agl \ \& \ Ael)$$

which is true. Then the consequent of the conditional is $(l = a)$, Lovelace is identical to a , which is true. Second, the instance for Kurt Gödel. The antecedent of the conditional here is false, because Gödel does not admire himself, hence the whole conditional is true. Third, the instance for Albert Einstein. Here the antecedent of the conditional is also false, because Lovelace does not admire Einstein, hence the whole conditional is true. We have all three instances true in the model, so this universally quantified formula is true in this model.

4.3 | IDENTITY: TREE PROOFS

4.3.1 | TREE RULES FOR IDENTITY

In this section, we'll explain how to use identity in tree proofs for predicate logic.

Remember, in tree proofs, we spell out the consequences of the statements. Sometimes this is linear, spelling out the conjuncts of a conjunction, and sometimes, with disjunction, it branches: each different branch represents one of the different ways of making our claims true together.

If any of the branches contains a contradictory pair B and $\sim B$, then that branch *closes*, because its constraints cannot be satisfied. If all of the branches close, then there is no way to satisfy our starting formulas. If one of the branches stay open, while you spell out all of the consequences of the formulas, then those formulas can be satisfied, and the open branch gives you a way to make them all true together.

Let's see how these rules should be expanded to deal with identity.

$$\begin{array}{ccc}
 & A & \\
 (\alpha \neq \alpha) & & (\alpha = b) \\
 & \times & | \\
 & & A[\alpha := b]
 \end{array}$$

The first rule is very simple. If a tree just contains ($a \neq a$), it should close, straight away, since ($a = a$) is true in every model. This is the *closure rule* for identity.

What should we do with true identity assertions? This is a little less straightforward. If we have a statement A and an identity ($a = b$), then we should be permitted to replace the name a by the name b in A , resulting in the formula $A[a := b]$, since a and b name the same object.

We'll read this rule carefully: if the formula ($a = b$) is in the branch, then we are permitted to substitute all occurrences of a with the name b – but not the reverse substitution. If we applied it backward, we would quickly get into an infinite loop.

Example (1): $\vdash (\forall x)(\forall y)(x = y \supset y = x)$

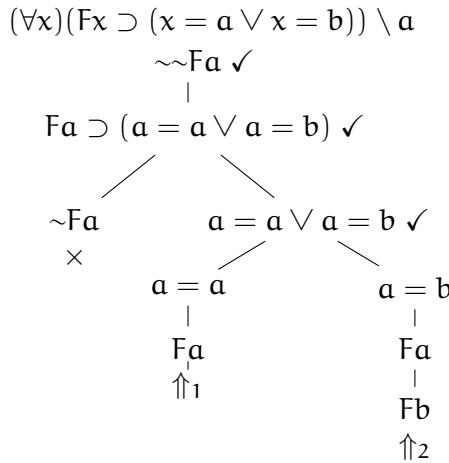
This formula expresses that the identity relation is *symmetric*.

$$\begin{array}{c}
 \sim(\forall x)(\forall y)(x = y \supset y = x) \checkmark a \\
 | \\
 \sim(\forall y)(a = y \supset y = a) \checkmark b \\
 | \\
 \sim(a = b \supset b = a) \checkmark \\
 | \\
 a = b \\
 b \neq a \\
 | \\
 b \neq b \\
 \times
 \end{array}$$

Starting with the top formula $(\forall x)(\forall y)(x = y \supset y = x)$, apply the tree rule for $\sim\forall$ and introduce a new name a . The second formula also requires the tree rule for $\sim\forall$ and we introduce another new name b , to result in the negated conditional $\sim(a = b \supset b = a)$. This is then developed with the linear rule to result in $a = b$ and $b \neq a$. The final step is the true identity development rule, taking A as $b \neq b$, so that $A[a := b]$ is the formula $b \neq b$. Applying the closure rule for negated self-identities, the single linear branch is closed.

Our second example gives a proof that $\sim Fa$ is not a provable logical consequence of the formula $(\forall x)(Fx \supset (x = a \vee x = b))$, which says that all F -things are either named a or named b .

Example (2): $(\forall x)(Fx \supset (x = a \vee x = b)) \not\vdash \sim Fa$



This tree starts with $(\forall x)(Fx \supset (x = a \vee x = b))$ and the negated conclusion (and double-negation formula) $\sim Fa$. Getting stuck into the top formula, it is a positive \forall formula, so we take an instance of the quantified variable x with the name a , to get the conditional $Fa \supset (a = a \vee a = b)$. Applying the branching rule for positive conditionals, we get a left branch with $\sim Fa$ and a right branch with $(a = a \vee a = b)$. The left branch closes immediately, as we have both $\sim Fa$ and $\sim \sim Fa$. Continuing the right branch, we break down the disjunction $(a = a \vee a = b)$ to get a left branch with $a = a$ and a right branch with $a = b$. We now go back and develop the double-negation formula $\sim Fa$, to obtain Fa on both open branches. On the right-most branch, we can apply the true identity development rule, taking A as Fa , so that $A[a := b]$ is the formula Fb . We could then go back to the first \forall formula, and take another instance of the quantified variable x with the name b , to get the conditional $Fb \supset (b = a \vee b = b)$. But this will not give us any new information, and it certainly won't do much for closing branches, so we can stop here.

From the tree with open branches, we get two different counter-models depending upon which branch we choose.

- From left open branch labelled \uparrow_1 we get a model M_1 with $D_1 = \{a, b\}$ and $I_1(Fa) = 1$, $I_1(a) = a$, and $I_1(b) = b$. There are no formulas on this branch forcing either sameness or difference between a and b .
- From right open branch labelled \uparrow_2 we get a model M_2 with $D_2 = \{a\}$ and $I_2(Fa) = 1$, $I_2(Fb) = 1$, $I_2(a) = a$, and $I_2(b) = a$. The identity $a = b$ forces the collapse to size 1 domain consisting of just a .

4.3.2 | TREE RULES FOR IDENTITY: COMPLETION

Here's the principle for *completing* proof trees with identity.

An open branch is **COMPLETELY DEVELOPED** only when we have applied the identity rule to *every* atomic or negated atomic formula A on that branch.

The restriction to atomic and negated atomic formulas is enough here, because it is what you need to do to make sure that the model is well defined. If we say Fa and $\neg Fb$ but $a = b$, we need to have that ruled out, because no model will make it true.

We could use the rule more often, with more complex formulas, but if we have applied it at least in the case of atoms and negated atoms, it's going to be enough to make a model.

- When constructing a model from an open branch, we must ensure that the identity symbol is interpreted by the identity relation as described by the open branch.

EXAMPLE: If a , b and c are the only names on the open branch, but the branch also contains $a = b$, then the domain of the model must contain one object which is named by *both* a and b , and a distinct object named by c .

That's enough for you to construct trees in the language of predicate logic with identity. Next up, we'll look at how we can use the language to express complex statements expressing counting properties.

4.4 | COUNTING

The addition of the identity predicate allows us to *count things*, in the form of *cardinality* quantifier phrases like *at least three things are F's* or *at most seven things are F's*.

4.4.1 | AT LEAST n-MANY

Let's start with an exercise: Consider the four formulas here, where the 1-place predicate T stands for "is a teacher in the subject", and where a and b are names. Which of these formulas does the best job in expressing the statement that there are at least two teachers in the subject? You take a look and make your choice.

- There are *at least* two teachers in the subject.

Which of these formulas does the best job in expressing this statement?

(a) $Ta \& Tb$ **(b)** $(\exists x)(\exists y)(Tx \& Ty)$ **(c)** $(\exists x)(\exists y)(Tx \& Ty \& x \neq y)$ **(d)** $Ta \& Tb \& a \neq b$

The first two choices, (a) and (b), won't do, because they are true even if there was only one thing that satisfies predicate T : if a and b named the same object ($Ta \& Tb$) is true if that thing is T (through, admittedly, it would be a very strange way to say something true). If there's one thing with property T then formula (b) is also true, since we can choose values for x and y (that thing with property T) but they have to be objects a and b . It is formula (c) that does the best job of it: it says that we've got *two* objects (one object x and another, y different from x) that do the job. It says not only that there has to be two things with the property – be two things with property T . The last formula goes further: it says not only that there has to be two things with the property – it's the third and fourth formulas (c) and (d) that require that there be two things with property T .

It's the third and fourth formulas (c) and (d) that require that there to make $(Tx \& Ty)$ true.

We can make this more general. For the general case of n many things, for counting numbers $n = 1, 2, 3, \dots$, we have the following characterisation:

At least n things are F

if and only if the formula below is true:

$$(\exists x_1) \cdots (\exists x_n)(Fx_1 \& \cdots \& Fx_n \& (x_1 \neq x_2) \& (x_1 \neq x_3) \& \cdots \& (x_{n-1} \neq x_n))$$

For example, for $n = 3$, we have:

At least 3 things are F

if and only if the formula below is true:

$$(\exists x_1)(\exists x_2)(\exists x_3)(Fx_1 \& Fx_2 \& Fx_3 \& (x_1 \neq x_2) \& (x_1 \neq x_3) \& (x_2 \neq x_3))$$

We need *three* non-identity constraints $x_1 \neq x_2$, $x_1 \neq x_3$ and $x_2 \neq x_3$, to ensure that the three objects are all different. For $n = 4$, we would need *six* non-identity constraints. In general, to express *at least n things*, we need a total of:

$$t_n = (n - 1) + (n - 2) + \cdots + 2 + 1 = \sum_{k=1}^{n-1} k = \frac{n(n - 1)}{2}$$

many non-identity constraints in the logic formula. (The numbers $t_n = \frac{1}{2}(n(n-1))$ are known as *triangular numbers*.) For $n = 5$, there are 10 non-identity constraints, while for $n = 6$, there are 15.

At least n things are F is an example of a statement using a *cardinality quantifier*. The quantifier phrase “*at least n many*” is like “*some*” or “*all*”. But we can express it using the quantifiers we already have on hand, making crucial use of the identity predicate. The quantifier phrase “*at most n many*” is likewise expressible in predicate logic with identity.

4.4.2 | AT MOST n -MANY

Let's do the same thing, but for "*at most n many*" things have a property, instead of "*at least n many*" things have a property.

Consider the four formulas here. Which of these formulas does the best job in expressing the statement that there are at most two teachers in the subject? You take a look and make your choice.

- There are *at most* two teachers in the subject.

Which of these formulas does the best job in expressing this statement?

- (a)** $(\exists y)(\exists z)(\forall x)(Tx \supset (x = y \vee x = z))$

(b) $(\forall x)(Tx \supset (x = a \vee x = b))$

(c) $(\exists x)(\exists y)(Tx \& Ty \& x \neq y)$

(d) $(\forall x)(\forall y)(\forall z)((Tx \& Ty \& Tz) \supset (x = y \vee x = z \vee y = z))$

The second formula (b) is close. It says that all of the teachers in the immediate family, the third formula (c) has to be wrong. I hat says "at least two", and that is completely different from "at most two". The second formula (b) is close. It says that all of the teachers in the immediate family, the third formula (c) has to be wrong. I hat says "at least two", and that is completely different from "at most two".

Our discussion of (b) hints that the first formula (a) is a good choice. It says that there are two things, y and z, such that $\exists y \exists z (Fy \wedge Fz \wedge \neg(y = z))$. That does anything with property T is one of those. This will work if y and z have property T (and nothing else has the property), or if y does something else (and nothing else does), or finally, if nothing does at all. That does what we want, so the first formula (a) is correct.

Perhaps surprisingly, the last formula (d) also works, even though it looks very different. It says that whenever I choose x, y and z, if they all have property T, the they are not all different. This also means that we have at most two things with property T, and like our translation for “*at least*”, it is nicely symmetric, with the same quantifier used for each variable.

For the general case of n many things, for counting numbers $n = 1, 2, 3, \dots$, we have the following characterisation:

At most n things are F

if and only if the formula below is true:

$$(\forall x_1) \cdots (\forall x_{n+1}) ((Fx_1 \wedge \cdots \wedge Fx_{n+1}) \supset ((x_1 = x_2) \vee (x_1 = x_3) \vee \cdots \vee (x_n = x_{n+1})))$$

The formula says: “If I make $n + 1$ choices of F things, then somewhere along the way, I must have double-counted, and two of my choices are the same”.

For example, for $n = 3$, we have:

At most 3 things are F

if and only if the formula below is true:

$$(\forall x_1)(\forall x_2)(\forall x_3)(\forall x_4) ((Fx_1 \wedge Fx_2 \wedge Fx_3 \wedge Fx_4) \supset ((x_1 = x_2) \vee (x_1 = x_3) \vee (x_1 = x_4) \vee (x_2 = x_3) \vee (x_2 = x_4) \vee (x_3 = x_4)))$$

As for the “*at least n things*” characterising formula, you need to be careful about the number of identity/non-identity constraints needed for the formula; here, we have identity constraints in the consequent of the conditional for “*at most n things*”. For example, for $n = 4$, the “*at most 4*” quantifier needs 5 quantified variables, so it will be $4+3+2+1 = 10$ identity constraints in the consequent of the formula.

With these counting quantifiers, we can do a little bit of mathematical reasoning inside logic. You could show, for example, that if you have at least 2 Fs and at least 2 Gs and nothing is both an F and a G, then you have at least 4 things that are F-or-Gs. That’s a good exercise for you to try: formalise this argument, and complete a tree for it.

At least 2 things are F.

At least 2 things are G.

Nothing is both F and G.

So, At least 4 things are either F or G.

Once you've done that, see if you can generalise it. If we have “*at least n Fs*” and “*at least m Gs*” and nothing is both an F and a G, then “*at least n + m things are either Fs or Gs*”. How does the corresponding tree work?

At least n things are F.

At least m things are G.

Nothing is both F and G.

So, At least n + m things are either F or G.

For those of you that are confident with mathematics: can you think of ways to represent multiplication, and derive some of its properties?

4.4.3 | EXACTLY ONE

We have seen how to express “*at least one*” and “*at most one*”; their conjunction expresses “*exactly one*”. In general, you can just conjoin “*at least n-many*” and “*at most n-many*” to get “*exactly n-many*” and that works fine. But can you find a formula simpler than that conjunction which expresses the same meaning? Take a look at this example. How would you formalise: “*there is exactly one teacher the subject*”? Take a look and make your choice.

- There is exactly one teacher in this subject.

Which of these formulas does the best job in expressing this statement?

(a) $(\forall x)(\forall y)((Tx \ \& \ Ty) \supset x = y)$

(b) $(\exists x)(\forall y)(Ty \supset x = y)$

(c) $(\exists x)Tx$

(d) $(\exists x)(Tx \ \& \ (\forall y)(Ty \supset x = y))$

The third option (c) won't work, because it just says that there is a teacher. That's true, even if there are many.

The first option (a) doesn't work: it says there is at most one teacher. It would be true even if there weren't any.

The second option (b) doesn't work, either. It says that there's something, where *if* anything is a teacher, it is identical to that teacher. But that is not enough: that would be true, even if nothing has property T. So, we need to add that clause in.

The last option (d) does the job.

And that's what we'll use to formalise our statement. Exactly one thing has property F can be rendered as: *Something has F and if anything has F it is identical to that first thing..*

*There is exactly one thing which is F
if and only if
 $(\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y))$
is true.*

This is historically a really important formula, because here we find a bridge between predicates and names: this statement makes the predicate F do the job of a name – of having a single thing that it *stands for*. This is important, because there are plenty of words (like names from mythology or fiction, or names arising out of mistaken identity) which behave like names which may not act like names in models of the language of predicate logic – since in models, names always have to name single things, which exist in the domain of the model. In the philosophy applications topic of this course, Chapter 7 on *Definite Descriptions*, we'll take a look at the way the language of predicate logic with identity, and the connections between predicates and names can be used to illuminate the relations between language and metaphysics.

4.5 | FUNCTIONS AND OTHER EXTENSIONS

It should come as no surprise to learn that Predicate Logic with Identity is not *all there is!* The logic commonly known as *First Order Logic* consists of predicate logic with identity together with *function symbols*. We have not included function symbols in the introductory-level course in logic as they add considerable complexity to the syntax and semantics, without gaining much expressivity when viewed from a natural language perspective. However, when viewed from the perspective of formalising mathematics in predicate logic, function symbols are extremely important. Nevertheless, as we shall see in the mathematics

applications topic in this course, Chapter 5 on *Quantifiers in Mathematics*, one can address a substantial topic in the mathematics of real numbers, namely the convergence and divergence of sequences of real numbers, without requiring a formal treatment of function symbols in the logic.

4.5.1 | ADDING FUNCTION SYMBOLS

The functions we are all most familiar with are the elementary arithmetic functions, beginning with addition, subtraction, multiplication and division. Each of these are 2-place functions, which take as input two numbers and return as output one unique number (except for division when the second number is 0, because $x \div 0$ is undefined).

A natural-language example includes the 1-place function on the domain of people, $b(x)$ for “*is the birth mother of x*”. But we could equally as well use a 2-place predicate Bxy for “*y is the birth mother of x*”, and express the uniqueness of y in relation to x with the formula:

$$(\forall x)(\exists y)(Bxy \ \& \ (\forall z)(Bxz \supset z = y)).$$

Here are a few atomic formulas involving function symbols:

$$2 + 2 = 4 \quad f(x) = x^2 + 5x - 3$$

The key points on predicate logic with functions:

- *Syntax*:
 - If a is a name and f is a one-place function symbol, then a is a TERM and $f(a)$ is a TERM.
 - If t_1, \dots, t_n are terms and f is a function_n symbol, then $f(t_1, \dots, t_n)$ is a TERM.
 - If t_1 and t_2 are terms, then $(t_1 = t_2)$ is an ATOMIC FORMULA.
 - If R is an n -place predicate symbol (other than identity), and t_1, \dots, t_n are terms, then $Rt_1 \dots t_n$ is an ATOMIC FORMULA.
- *Semantics*: interpret a function_n symbol in a model as a function_n on the domain: $I(f)$ is a function that takes as input an n -tuple of objects (d_1, d_2, \dots, d_n) from the domain, and returns as output a unique object c in the domain which is $I(f)$ applied to (d_1, d_2, \dots, d_n) .
- *Proofs*: The tree development rules get rather more complicated in the presence of functions; for example, with just one name a and a 1-place function symbol f , there may be infinitely many terms $a, f(a), f(f(a)), f(f(f(a)))$, and so on, on an open branch of a completely developed tree.
- Proof tree development rule for positive identity formulas: if branch contains formula A including term t_1 , and branch also contains $(t_1 = t_2)$ for another term t_2 , then extend the branch with the formula $A[t_1 := t_2]$.

4.5.2 | MULTI-SORTED LOGIC

It is common, when using quantified expressions, to think of different quantifiers as ranging over different *sorts* or *kinds* of objects. For example, we may want to say that for any student at the university, that student has a *student number*, and any number that is a *student number* is at least a five digit number. We could state this in a formula such as this:

$$(\forall x)((\exists n)N_{xn} \& (\forall m)(N_{xm} \supset (m > 9999)))$$

where N_{xn} states that student x has student number n , and any m that's a student number is greater than 9999. But in this case, it might make sense to restrict the first quantifier $(\forall x)$ to range over people, and $(\exists n)$ to range over *numbers*. Mathematical writing often has this feature—different quantifiers are naturally understood as ranging over different kinds of objects (points, lines, numbers, sets, etc.) and it is very rare to find quantifiers that range over a disparate collection of different kinds of things. So, we could introduce different variables as ranging over different *sorts* of objects, and then regiment the whole of the language to respect this distinction. This would mean that we have:

- Variables and names that divide into different *sorts*.
- Different sorts correspond to different *domains* of objects.
- Each predicate expects to be fed terms of the right *sort*.

Why the existential and the universal quantifier to range over the numbers?
Some students might have more than one student number, if they've enrolled more than once.

In this last condition, for example, N_{xn} expects a student in the first slot, and a number in the second slot. The ‘on’ relation in geometry might expect a *point* in the first position, and a *line* in the second, and so on.

The move to a multi-sorted first order logic is, in one sense, very convenient for many different applications of predicate logic. However, there is a sense in which it is nothing more than what can be done with predicate logic by itself. If we have a predicate for each sort, then we can mimic the restricted quantifiers of multi-sorted logic by explicitly restricting the general quantifiers with the predicates that define the desired restricted quantifier. If we were to quantify over the one, general domain, instead of the two sorted simple formula $(\forall x)((\exists n)N_{xn} \& (\forall m)(N_{xm} \supset (m > 9999)))$ we would get

$$(\forall x)(\$x \supset ((\exists n)(N_n \& N_{xn}) \& (\forall n)(N_n \supset (N_{xn} \supset (n > 9999)))))$$

which has the same effect as the original formula, but is certainly *longer*. At the very least, multi-sorted predicate logic allows for simpler formulas.

4.5.3 | HIGHER-ORDER QUANTIFICATION

An atomic formula in predicate logic has the form

$$Fa$$

consisting of a predicate and a name (or a number of names). We can quantify into the ‘name’ position to form judgments like

$$(\exists x)Fx \quad (\forall x)Fx$$

and we have seen the power of using quantifiers and the ways predicate logic goes beyond propositional logic. But the atomic formula ‘Fa’ has two positions at which a quantifier might operate. We could consider quantifying into *predicate* position.

Here is one reason why we might want to do this. Consider identity. This is a straightforward tautology:

$$(Fa \ \& \ \sim Fb) \supset a \neq b$$

If a has the property F and b does not, then a and b must be distinct objects. This is quite straightforward, and it holds of *any* pair of objects—the specifics of which a and b we choose is not important. We can generalise this, and say

$$(\forall x)(\forall y)((Fx \ \& \ \sim Fy) \supset x \neq y)$$

which is also a tautology. This is straightforward enough. But not only does it not matter which *objects* a and b we choose, it also doesn’t matter which *property* we choose, too. We could, surely, have generalised in the F position, to say

$$(\forall X)((Xa \ \& \ \sim Xb) \supset a \neq b)$$

which says that whatever property X we choose, if X holds of a and not of b , then a and b are distinct. The virtue of being able to say this is that it holds of *any* property, whether we have a way to describe that property or not—in just the same way that first order quantification (quantification into name position) allows us to talk of objects whether we have *names* for those objects or not.

Another way of saying $(\forall X)((Xa \ \& \ \sim Xb) \supset a \neq b)$ would presumably be

$$(\exists X)(Xa \ \& \ \sim Xb) \supset a \neq b$$

that if there is some property that holds of a and not of b , then a and b are distinct. *Second* order logic allows for quantification into the predicate position of formula, and it allows us to state things like this. In fact, you might think not only that if there is some property that a has that b doesn’t then a and b are distinct, but that it’s the *only* way for a and b to be distinct. You might think that *this* should be a tautology, too

$$(\exists X)(Xa \ \& \ \sim Xb) \equiv a \neq b$$

or, to much the same effect,

$$(\forall X)(Xa \equiv Xb) \equiv a = b$$

This says that a and b are identical if and only if they share all properties in common. This is either a completely straightforward tautology,

By analogy to the logical equivalence between $(\forall x)(A \supset B)$ to $(\exists x)A \supset B$, in the case where the formula B does not contain the variable x free.

or it is a matter of philosophical controversy and debate. The direction that's most controversial in the biconditional is the left-to-right direction: that if $(\forall X)(Xa \equiv Xb)$ then $a = b$. But this is a straightforward tautology if you realise that an instance of $(\forall X)(Xa \equiv Xb)$ is $a = a \equiv a = b$. (The property of ‘being identical to a ’), and since $a = a$ is a tautology, it follows from $a = a \equiv a = b$ that $a = b$, too.

The logical system that allows us to quantify into predicate position as well as name position is called *second* order predicate logic. Where there is a first and a second, there is always more. We can move to *higher* order quantification by realising that we can predicate properties (or relations) of other properties and relations. A relation is R said to be *symmetric* if and only if $(\forall x)(\forall y)(Rxy \supset Ryx)$. We could formalise the statement “ R is symmetric” as a kind of higher-order ascription of a property *to* a property.

$$\text{Sym } R$$

where ‘ Sym ’ is a level-2 property, which holds of level-1 properties, where level-1 properties hold of level-0 *objects*. *Higher* order logic allows for ascription of properties of properties of properties ... *ad infinitum*.

Could something be *both* a straightforward tautology and philosophically controversial? Surely that's possible, too.

Stewart Shapiro's book *Foundations without Foundationalism* [10] is a very clear discussion of the costs and benefits of expanding from first order predicate logic to second order predicate logic.

4.5.4 | NON-DENOTING TERMS

Another way that people have considered extending or modifying first order predicate logic is to expand the treatment of names and other terms. Some names, like ‘Pegasus’ seem to fail to refer to an object—there is no winged horse Pegasus, as there are no winged horses at all. Some function terms, like the *division* function symbol, do not always return a value. Division by zero is *undefined*.

$$\text{Pegasus does not exist.} \quad \frac{1}{0} \text{ is undefined.}$$

Traditional first-order predicate logic is ill-suited to modelling this phenomenon, since names must be defined by interpretations, and function symbols must be everywhere defined. In Chapter 7, we will take a closer look at how to modify the rules of first order predicate logic to eliminate this assumption.

4.5.5 | IS THERE A SINGLE LEVEL OF LOGICAL STRUCTURE?

Finally, it is worth asking the question as to whether there is one correct answer to the question of what logical structure is to be found in our thought and our talk, in all of the ways information might be carried, by our minds and our machines and the world around us. We have discovered propositional and predicate logic structure, and we have gestured at other forms of logical structure. Is there one single fundamental level of logical structure waiting there to be found? Or is it a matter of convention as to what logical structure we find it useful to elucidate for a given task of analysis? Given a focus on one particular level of logical analysis, is there a single way to isolate the logical

properties at hand, or are the equally good but different ways to carve out the concepts of logical consequence, consistency or equivalence? These are debates over the fundamental nature of logical concepts, and there are continuing discussions between *logical monists* and *logical pluralists* over the correct ways to understand what is at stake when we proposed different logical systems. To explore this further, we recommend Jc Beall and Greg Restall's book *Logical Pluralism* [3].

PART 11

Applications of Predicate Logic

QUANTIFIERS IN MATHEMATICS

5

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- The real number line \mathbb{R} under its ordering relation $<$ as a model of a predicate logic language.
- *Densely-ordered* and *discretely-ordered* subsets of real numbers characterised by predicate logic formulas.
- The *Continuum* or *Least Upper Bound* property of the real number line.
- Distance between real numbers via a 3-place predicate $D_{xy}\varepsilon$ meaning “the distance between x and y is less than ε ”.
- Sequences of real numbers; *convergence* and *divergence* of sequences of real numbers expressed as predicate logic formulas using the order predicate $<$ and the *distance-is-less-than* predicate D .
- Sequence convergence applied to the digitized representation of real numbers in computers.

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 practise your skills and get familiar with each of these concepts.

SKILLS

These are the skills you will learn to demonstrate.

- Translate between mathematical English sentences and predicate logic formulas expressing properties of real numbers.
- Apply the semantics of predicate logic in a mathematical context, to explain why a sentence or formula is true or is false in the standard model over \mathbb{R} .
- Determine whether a subset of real numbers is *densely-ordered* and/or *discretely-ordered*.
- Apply the *Continuum property* of the real numbers to identify whether or not a set of real numbers has a Least Upper Bound.
- Use (read and write) restricted quantifier formulas in a mathematical context.
- Given a sequence of real numbers, together with the definitions of convergence and divergence formalised in predicate logic, determine whether the sequence converges or diverges.

5.1 | MATHEMATICS AND LOGIC

There are multiple layers of relations between mathematics and logic.

- **The foundations of mathematics are built on logic.** At the foundational level, mathematics is grounded in logic. Much of contemporary mathematics can be formalised in *set theory*, and in turn, set theory can be formalised in *first-order logic*, which is predicate logic with identity and functions.
- **Logic is a branch of mathematics.** It is a sub-field within the broader discipline. Admittedly, it is one of the most abstract branches of mathematics.
- **Practically, doing and using mathematics requires logic.** At a practical level, doing and using mathematics requires logic. Each step taken within a mathematical proof must be a valid inference that could – in principle – be formalized within logic.
- **Logical tools needed to clarify and understand complex mathematical concepts.** The precision of logical constructs and tools are valuable in clarifying and understanding complex mathematical concepts. It is this aspect of the relationship between mathematics and logic that we will be exploring in this section of the course.

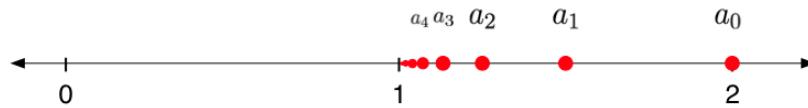
Our core example of a complex concept in need of logical clarification and insight is that of a sequence of real numbers *converging to a limit*. Consider the following sequence of numbers that is decreasing in order:

$$2, 1\frac{1}{2}, 1\frac{1}{4}, 1\frac{1}{8}, \dots, 1\frac{1}{2^n}, \dots$$

Intuitively we have a clear sense this sequence is “tending” to the value 1, getting closer and closer, but never quite getting there, even as the count index n goes to infinity. We say that the *limit* of this sequence is 1.

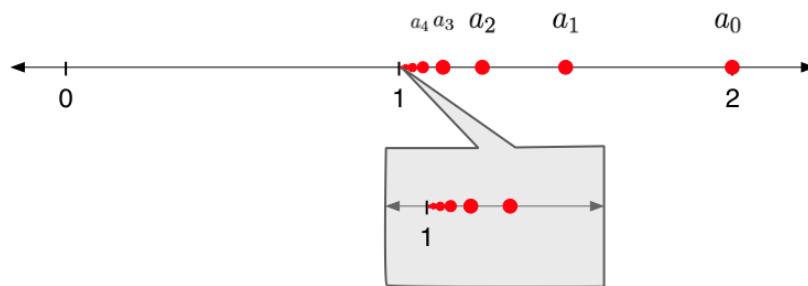
The notion of the limit of a sequence seems intuitive enough, but it is surprisingly complex. Drawn here on the number line, we see the terms in our example sequence as red dots getting closer and closer to 1. At the scale we start with, the red dots get blurry after term a_4 . The meaning of this phrase “getting closer and closer” is that we can *zoom in* or *magnify* close to the limit:

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



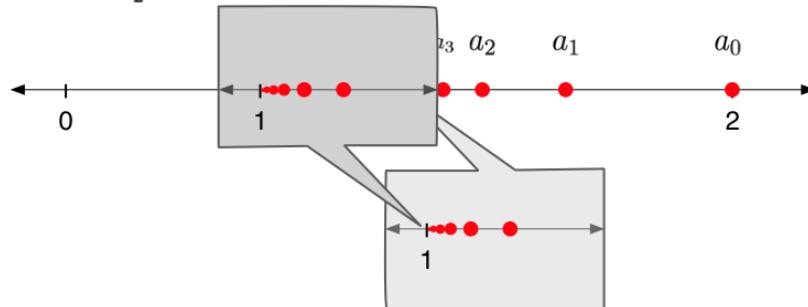
and see the same sort of behaviour of sequence terms a_n (red dots in the graph) approaching 1 from above. Again, the red dots quickly get blurry, so we do another *zoom in* or *magnify* close to the limit:

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



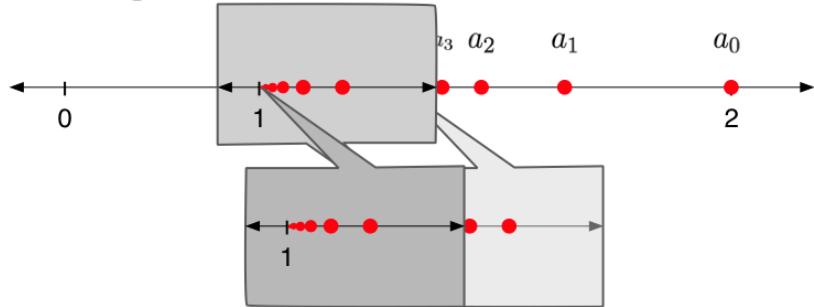
and *still* see same sort of behaviour of red dots approaching 1 from above. The red dots quickly get blurry, so we can again *zoom in* or *magnify* close to the limit:

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



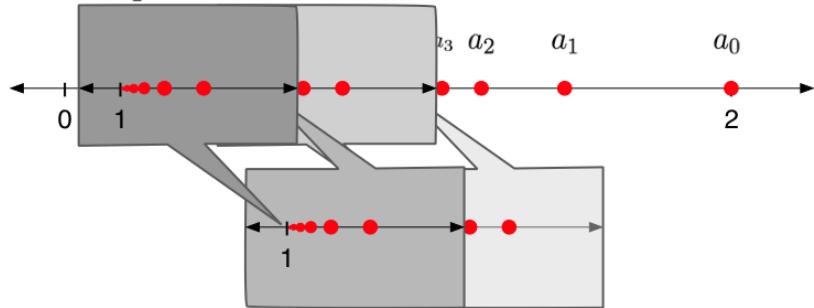
and *still* see same sort of behaviour of red dots approaching 1 from above. Once more, the red dots quickly get blurry, so we can again *zoom in* or *magnify* close to the limit:

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



and still more of the same sort of behaviour of red dots approaching 1 from above. The notion of a limit of a sequence is that we could zoom in or magnify *infinitely many times*, and each time, we would still see the same sort of behaviour of sequence terms getting closer and closer to the limit number.

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



Now consider another, quite different sequence of numbers:

$$1, -1, 1, -1, 1, -1, \dots (-1)^n, \dots$$

Here there is no limit; it bounces back and forth between 1 and -1 .

A sequence that has a limit is said to *converge*, and otherwise, the sequence is said to *diverge*.

This section of the course will work up to the point of making precise the notion of a sequence of numbers having a limit. Crucially, we need the *logical quantifiers* (“for all”, \forall , and “there exists”, \exists) to unpack the complex content of the concepts of convergence and divergence.

Historically there were a number of very important concepts which caused confusion among mathematicians until they were given precise logical definitions in the 19th century:

- (i) A function $f : \mathbb{R} \rightarrow \mathbb{R}$ being *continuous*.
- (ii) A function f having a defined slope or *derivative* $f'(x_0)$ at a point x_0 .
- (iii) An *infinite sum* having a well defined value: for instance,

$$\sum_{n=1}^{\infty} \frac{1}{2^n} = \frac{1}{2} + \frac{1}{4} + \frac{1}{8} + \dots = 1.$$

Each of these concepts depends on, or makes essential use of, the notion of a sequence of numbers **converging to a limit**. Now, do not be worried that you will need calculus to follow these lessons; there will be no further mention of continuity or derivatives. I promise: all you need is basic high school level mathematics, which I review as needed, together with predicate logic with identity as covered in the core part of this course.

Teaser: see the video

<http://youtu.be/w-I6XTVZXww>

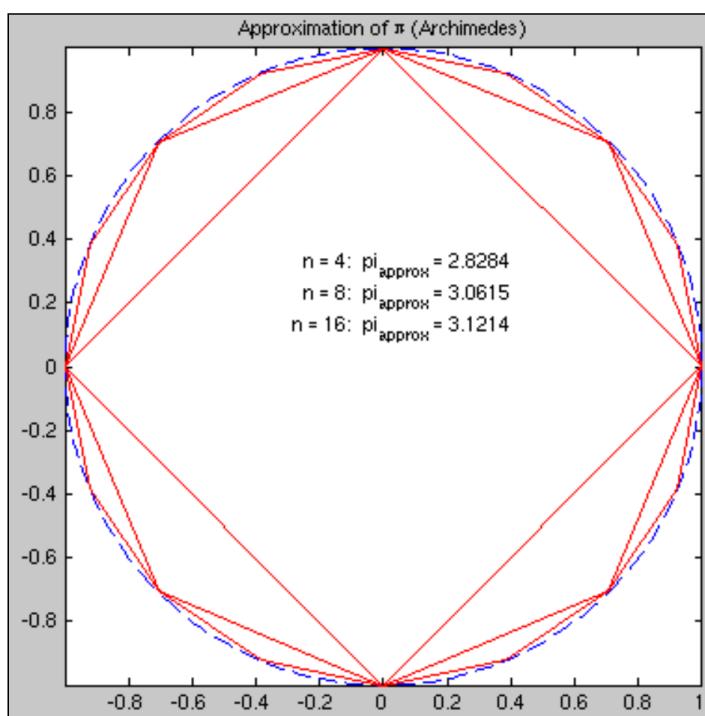
for an *astounding* “proof” that the infinite sum $1+2+3+4+\dots = -\frac{1}{12}$.

By the end of this section of the course, you will be able to explain using logic what is wrong with the supposed “proof”.

5.1.1 | HISTORY OF CONVERGENCE OF SEQUENCES

Convergence of sequences of numbers has a very long history. A famous early example is described by [Archimedes of Syracuse](#) (287-212 BC) with his approximation sequence for the number π . Remember from geometry that π is equal to the circumference or length of the perimeter of a circle of diameter 1. Geometrically, we can see that by taking polygons with more and more sides, it will get closer and closer to the circle. The image shows the start of the lower approximation sequence coming from *inscribed* polygons, with 4, 8 and 16 sides. The upper approximation sequence comes from the perimeter of *circumscribed* polygons, with the polygon on the outside of the circle.

As early as 2000 BC, people in ancient Babylon, Egypt, China and Israel had discovered that the circumference of a circle is directly proportional to the diameter.



Archimedes (or his students!) calculated by hand up to 96-sided polygons, to get the lower approximation of $3\frac{10}{71}$, which is approximately 3.1408, and the upper approximation of $3\frac{1}{7}$, which is approximately 3.1428. Since π to 6 decimal places is 3.141592, Archimedes got within accuracy $\frac{1}{100}$, and he gave a clear description of how to *continue* on the approximation sequence indefinitely, by taking polygons with more and more sides, to get more and more accurate approximations.

The earliest known sequence is that approximating $\sqrt{2}$, produced by Babylonian mathematicians using geometric reasoning in the period 1800 to 1600 BC. The Babylonian clay tablet with the code name YBC-7289 shows a sequence of calculations leading to an estimate of $\sqrt{2}$ as the fraction 30,547 over 21,600, which is accurate to 6 decimal places. The funny fractions come from working *sexagesimally*, which means in base 60. Later Babylonian artifacts show that a knowledge of a recursive formula for calculating better and better approximations: the new approximation a_{n+1} is calculated from the current approximation a_n using the formula :

$$a_{n+1} = \frac{a_n}{2} + \frac{1}{a_n}.$$

The 17th century saw the development of calculus by [Isaac Newton](#) [1642–1727] and [Gottfried Leibniz](#) [1646–1716]. Newton, in particular, was very fond of using *infinite sums*, but at that stage, they were without a rigorous basis and thus sometimes used in error. Newton's earlier work of calculus made extensive use of the notion of an *infinitesimal* quantity, that was smaller than any “normal” number but different from zero.

In the early 19th century, the German mathematician [Carl Friedrich Gauss](#) [1777–1855] started a new push to clarify the notion of convergence for sequences and infinite sums, but further work was needed.

Soon after, in 1817, the Czech (or then Bohemian) mathematician [Bernhard Bolzano](#) [1781–1848] took up Gauss' call and gave the first formulation of the modern version of sequence convergence, with his natural language formulation including the *for all-there exists-for all* pattern of quantification that we will get to a few lessons ahead. This was 60 years *before* Frege's development of predicate calculus, so the meaning and logic of the quantifiers had yet to be clarified.

Bolzano's work was not well known, and in the first half of the 19th century, it was soon eclipsed in notoriety by the work of the French mathematician, [Augustin-Louis Cauchy](#) [1789–1857]. Cauchy's better known formulation of sequence convergence (and also of continuity of functions) was framed in terms of infinitesimal quantities, as part of a research project to give Newton's infinitesimal calculus a rigorous basis. Never-the-less, Cauchy's version of sequence convergence was a step down from that of Bolzano, and led to a famous mistake by Cauchy confusing continuity and uniform continuity of functions.

Around 1860, the German mathematician [Karl Weierstrass](#) [1815–1897] gave a series of lectures in Berlin (*Introduction to Analysis* (1859–1860) and *The General Theory of Analytic Functions* (1863–1864)) which

became the “gold standard” for the rigorous development of sequence convergence, continuity of functions, and other key concepts in calculus and analysis. Weierstrass was particularly indebted to Bolzano in his formulation of sequence convergence.

Less than 20 years later, and heavily influenced by the work of Bolzano and Weierstrass in mathematical analysis, [Gottlob Frege](#) [1848–1925] presented predicate calculus, published as *Begriffsschrift* (translation: *Concept Notation: A formula language of pure thought modelled upon that of arithmetic*), 1879. An intellectual struggle in mathematics, to make precise complex some mathematical concepts, was a major impetus to the development of predicate calculus. And then in turn, the development of predicate calculus made possible the even more rigorous formulation of those concepts, as we shall see in this series of lessons, for the concept of sequence convergence.

The notion of sequence convergence is used widely within many areas of pure and applied mathematics, as well as in applications within science, engineering, economics, medicine, and other disciplines.

- dynamical system models: behaviour as time goes to infinity
- approximation methods in science and engineering
- computer representation of real numbers

We will discuss the last of these at the end of §5–5.

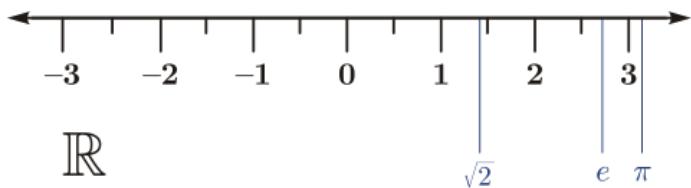
In the remainder of this section of the course, we will progress through a series of stages to get to the concept of sequence convergence formulated in predicate logic. To begin, we will review the basics of the real number line, and see how the properties of the standard ordering on the real numbers are readily formalised in predicate logic. We then focus on the notion of distance between real numbers, and study a 3-place predicate D that holds of numbers a , b and ε when the distance between a and b is less than ε . With this preparation, we can then get to the formal definition of sequence convergence using the *for all-there exists-for all* quantifier pattern. After a bunch of examples and exercises for you, we finish off with a discussion of the role of sequence convergence in the digitized representation of real numbers in computers.

5.2 | THE REAL NUMBER LINE

The task for this sub-section is to use predicate logic to deepen our understanding of the real number line and its ordering relation, with the order being what makes the number line a *line*.

5.2.1 | NUMBER SYSTEMS

Definition: The *real numbers*, written \mathbb{R} , is the set of all numbers that can be given a finite or infinite decimal expansion.



$$\sqrt{2} = 1.41421356237309504880168872420969807\dots$$

$$e = 2.71828182845904523536028747135266249\dots$$

$$\pi = 3.14159265358979323846264338327950288\dots$$

\mathbb{N} natural numbers

Start with the whole, counting numbers, from 0.



\mathbb{Z} integers

Add the negative integers, extending backwards.

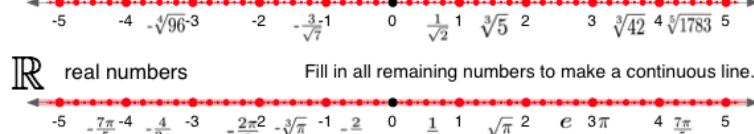


\mathbb{Q}

rational numbers $-\frac{42}{19}, -\frac{6}{7}, \frac{9}{5}$ Insert all the fractions.

$\mathbb{A}_{\mathbb{R}}$ real algebraic numbers

Insert all real roots of polynomials with integer coeffs.



\mathbb{R} real numbers

Fill in all remaining numbers to make a continuous line.



Recall the progression of number systems as you learned them at school. At first, there was just the *natural numbers*, the whole numbers, starting with zero. These are the numbers $0, 1, 2, 3, 4, 5, \dots$ and so on, increasing by 1, forever, toward infinity. The standard symbol for set of natural numbers is \mathbb{N} .

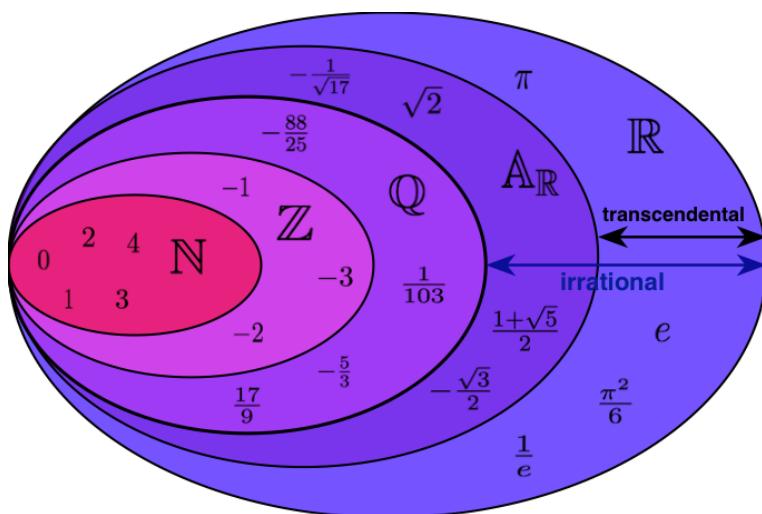
As we expanded our arithmetic skills, we came across the idea of negative numbers as well as positive numbers. By the *integers*, we mean the set of all whole numbers, positive and negative, extending forward toward infinity and backward toward negative infinity. The standard symbol for the integers is \mathbb{Z} , from German word *Zahl*, meaning “number”.

The next step up was to *fractions*. The set of *rational numbers*, written \mathbb{Q} , is the set of all numbers, positive and negative, than can be written as a fraction $\frac{m}{n}$, where m is an integer and n is a natural number greater than 0. So this means we fill in lots of the space between successive integers n and $n + 1$, but by no means all the space. The set of rational numbers is still *countably infinite*, which means that it is an infinity the same size or cardinality as the set of natural numbers.

Another step up is required when we get to algebra, and try to solve an equation like $x^2 = 2$. We get $\sqrt{2}$, the square-root of 2, which is not a rational number – as was first established by the Pythagorean School of mathematicians in Ancient Greece, about 500 BC. We don't stop at square roots, but add also cube roots and fourth roots and fifth roots, and so on. The *real algebraic numbers* $\mathbb{A}_{\mathbb{R}}$ are the set of all real numbers that are solutions to polynomial equations with integer coefficients. This fills up more of the space between rationals, but still not all the space. Indeed, the set of real algebraic numbers is also still *countably infinite*, so no bigger a degree of infinity than the rationals.

The last step to building up the real number line is then to fill in all the remaining space to create a continuous line, or *continuum*. It is only at this stage of the build up that *transcendental* numbers like π and the natural logarithm base e are added in. At the end of this lesson, we will see how to express the *Continuum property* of the real number line in predicate logic.

We can visualise the build up of the number systems in a Venn diagram, with the natural numbers \mathbb{N} the smallest, then the integers, the rationals, the real algebraic numbers, and then finally the transcendental numbers to fill out the whole of \mathbb{R} .



$$\begin{aligned}\mathbb{N} &= \text{Natural numbers} \\ &= \{0, 1, 2, 3, 4, \dots\}\end{aligned}$$

$$\begin{aligned}\mathbb{Z} &= \text{Integers} \\ &= \{\dots, -3, -2, -1, 0, 1, 2, 3, \dots\}\end{aligned}$$

$$\mathbb{Q} = \text{Rational numbers}$$

$$\mathbb{A}_{\mathbb{R}} = \text{Real algebraic numbers}$$

$$\mathbb{R} = \text{Real numbers}$$

We will use predicate logic to express the difference between:

Discrete / digital quantities $n \in \mathbb{N}$ or $n \in \mathbb{Z}$, **and**
Continuous / analog quantities $r \in \mathbb{R}$ or $r \in \mathbb{A}_{\mathbb{R}}$

where $a \in S$ means a is a member of the set S .

5.2.2 | STANDARD MODEL OF THE REAL NUMBERS

For this section of the course, we are only interested in the **standard model of the real numbers**: model $M = \langle D, I \rangle$ with domain $D = \mathbb{R}$, and interpretation I such that:

- the numerals $0, 1, 2, 3, \dots$ name the natural numbers we think they name, and their negatives $-1, -2, -3, \dots$ for the negative integers, as usual;
- usual decimal representation names for rational and real numbers, e.g. 3.14159 is 5-place approx of π , plus distinguished names e.g. $\pi, e, \sqrt{2}$;
- the 1-place predicate N will be interpreted by the natural numbers: $I(N) = \mathbb{N}$, i.e. Na is true in M iff $a \in \mathbb{N}$;
- the 2-place predicate symbol $<$ is interpreted by the standard ordering:

$$I(<) = \{ (a, b) \in \mathbb{R}^2 \mid a \text{ is strictly smaller than } b \}$$

- the basic arithmetic operations $(+, -, \times, \div)$ are the standard ones on \mathbb{R} ; we will use these to define some more predicates on \mathbb{R} .

The basic arithmetic operations of addition, subtraction, multiplication and division will be used in expressions to define more predicates on the real numbers. Now of course, to do this formally, we would need to be working in full first-order logic including function symbols, and we are not. Instead, we will use simple arithmetic expressions like $x + y$ and zx to describe predicate relationships between numbers, on the understanding that they are interpreted exactly how we expect them to be, with the meaning they have acquired since we learnt mathematics at school.

This standard model of the real numbers is what is used every day by mathematicians and scientists, as well as engineers, economists and school students. The difference is that most of them don't think of it as a model or predicate logic.

Within this standard model of the reals, we can express the usual inductive characterisation of the **natural numbers** $\mathbb{N} = \{0, 1, 2, 3, \dots\}$:

$$(\forall x)(Nx \equiv (x = 0 \vee (\exists y)(Ny \& x = y + 1))).$$

This says that x is a natural number, in \mathbb{N} , if and only if either x is 0 or else $x = y + 1$ for some number y already known to be a natural number. Notice this does *not* count as a *definition* of the natural numbers because the predicate symbol N occurs on both sides of the bi-conditional.

The **integers** \mathbb{Z} extend the natural numbers by adding the negatives of all natural numbers other than 0:

$$(\forall x)(Zx \equiv (Nx \vee (\exists y)(Ny \& y \neq 0 \& x = -y))).$$

In contrast with the formula for the predicate N , this does count as a *definition* of the 1-place predicate Z because there is no circularity, as there is no occurrence of Z on the right-hand side of the bi-conditional connective.

The *rationals* \mathbb{Q} are all the fractions $\frac{m}{n}$ with $m \in \mathbb{Z}$, $n \in \mathbb{N}$ and $n \neq 0$:

$$(\forall x)(Qx \equiv (\exists m)(\exists n)(Zm \ \& \ Nn \ \& \ n \neq 0 \ \& \ x = m \div n)).$$

Recall that every rational number has either a finite decimal expansion or an infinite repeating decimal expansion. This means we – in principle – have a *name* for every rational number, in the form of its decimal expansion. As we shall see after we get to sequence convergence, $0.\overline{9999}$, with infinitely repeating 9s, is another name for the number 1.

The *real algebraic numbers* $\mathbb{A}_{\mathbb{R}}$ are real numbers $x \in \mathbb{R}$ such that $p(x) = 0$ where p is a polynomial

$$p(x) = a_n x^n + \dots + a_2 x^2 + a_1 x + a_0$$

with integer coefficients $a_0, a_1, \dots, a_n \in \mathbb{Z}$.

5.2.3 | 2-PLACE ORDER RELATION ON THE REALS

The predicate we are particularly interested in is the *order* relation of less-than, $<$, on the set \mathbb{R} of real numbers. The order relation is such that every two numbers are *comparable*: for all $a \in \mathbb{R}$ and $b \in \mathbb{R}$,

either $a < b$ or $a = b$ or $b < a$.

For instance, $-7 < 3$ and $64.21 < 157.33$.

We can define the ‘strictly-greater than’ relation $>$ as the converse:

$$(\forall x)(x > y \equiv y < x)$$

and likewise the ‘less-than-or-equal-to’ relation:

$$(\forall x)(x \leq y \equiv (x < y \vee x = y)).$$

In virtue of the linearity of the underlying order, these additional order predicates are related by negation:

$$(\forall x)(x \leq y \equiv \sim(y > x)) \quad \& \quad (\forall x)(x \geq y \equiv \sim(x < y)).$$

We can now start building up a list of properties of the order predicates $<$ and \leq expressed as formulas of predicate logic. The following formulas are true in the standard model with domain \mathbb{R} .

- (i) $(\forall x)(\forall y)(x < y \vee x = y \vee y < x)$
- (ii) $(\forall x)(\forall y)((x \leq y) \equiv \sim(y < x))$
- (iii) $(\forall x)(\forall y)((x < y) \equiv (x \leq y \& x \neq y))$
- (iv) $(\forall x)(\forall y)(\forall z)((x < y \& y < z) \supset x < z)$
- (v) $(\forall x)(\forall y)(\forall z)((x \leq y \& y \leq z) \supset x \leq z)$
- (vi) $(\forall x)\sim(x < x)$
- (vii) $(\forall x)(x \leq x)$
- (viii) $(\forall x)(\forall y)((x \leq y \& y \leq x) \supset x = y)$
- (ix) $(\forall x)((\exists y)x < y \& (\exists z)z < x)$
- (x) $(\forall x)(\forall y)(x < y \supset (\exists z)(x < z \& z < y))$
- (xi) $(\forall x)(\forall y)(x < y \supset (x < \frac{1}{2}(x+y) \& \frac{1}{2}(x+y) < y))$
- (xii) $(\forall x)(\forall y)(x < y \equiv (y - x > 0 \& x - y < 0))$
- (xiii) $(\forall x)(\forall y)(\forall z)(x < y \equiv x + z < y + z)$
- (xiv) $(\forall x)(\forall y)(\forall z)(x \leq y \equiv x + z \leq y + z)$
- (xv) $(\forall x)(\forall y)(\forall z)(z > 0 \supset (x < y \equiv zx < zy))$
- (xvi) $(\forall x)(\forall y)(\forall z)(z > 0 \supset (x \leq y \equiv zx \leq zy))$
- (xvii) $(\forall x)(\forall y)(x < y \equiv -y < -x)$
- (xviii) $(\forall x)(\forall y)(\forall z)(z < 0 \supset (x < y \equiv zy < zx))$
- (xix) $(\forall x)(\forall y)(\forall z)(z < 0 \supset (x \leq y \equiv zy \leq zx))$

Formula (i) expresses comparability or linearity; (ii) and (iii) express relationships between $<$ and \leq . (iv) and (v) express transitivity: if x is less than y and y is less than z , then x is less than z ; (vi) irreflexivity; and (vii) reflexivity. Formula (viii) expresses anti-symmetry; (ix) bi-infinity; (x) denseness; and (xi) denseness expressed concretely with mid-point. Formula (xii) expresses a basic relationship between the order relation and the difference between numbers: for all reals x and y , x is less-than y if and only if the difference $y - x$ is positive (or greater-than 0) and $x - y$ is negative (or less-than 0). Formulas (xiii) and (xiv) express that order relations are preserved by adding the same number to both sides. Since subtracting is adding the negative, this means that order relations are preserved by subtracting the same number to both sides. Formulas (xv) and (xvi) express that order relations are preserved by multiplying the same positive number to both sides. Formula (xvii) expresses that order relations are *reversed* by taking negatives: x is less-than y if and only if $-y$ is less-than $-x$. Formulas (xviii) and (xix) express that order relations are reversed when multiplying the same negative number to both sides.

5.2.4 | DENSELY AND DISCRETELY ORDERED SUBSETS

The order predicate allows us to differentiate between *analog* versus *digital* sets of numbers. First, remember from the semantics of predicate logic that any 1-place predicate is interpreted by a set of objects in the

domain: exactly those objects of which the predicate holds true. In our setting, a 1-place predicate P will pick out a set of real numbers $I(P)$; we will also call this the P -set:

$$I(P) = \{ a \in \mathbb{R} \mid Pa \text{ is true in } M \}$$

We have seen this already with 1-place predicates: $I(N) = \mathbb{N}$, $I(Z) = \mathbb{Z}$, $I(Q) = \mathbb{Q}$.

Definition: Predicate P defines a *densely ordered* set if and only if the following formula is true:

$$\begin{aligned} (\forall x)(\forall y)((Px \& Py \& x < y) \supset \\ (\exists z)(Pz \& x < z \& z < y)) \end{aligned}$$

Definition: Predicate P defines a *discretely ordered* set if and only if the following formula is true:

$$\begin{aligned} (\forall x)((Px \& (\exists w)(Pw \& x < w)) \supset \\ (\exists y)(Py \& x < y \& \sim(\exists z)(Pz \& x < z \& z < y))) \end{aligned}$$

The P -set is *densely ordered* formula:

$$(\forall x)(\forall y)((Px \& Py \& x < y) \supset (\exists z)(Pz \& x < z \& z < y))$$

says: For all numbers x and y in P , if $x < y$, then there exists a z in P that lies strictly between x and y . Analog quantities belong in densely ordered sets.

In contrast, digital quantities (any data processed by a computer) must be discretely ordered. The P -set is *discretely ordered* formula:

$$\begin{aligned} (\forall x)((Px \& (\exists w)(Pw \& x < w)) \supset \\ (\exists y)(Py \& x < y \& \sim(\exists z)(Pz \& x < z \& z < y))) \end{aligned}$$

says: For all numbers x in P , if there is some w in P above x (so x is not the largest in P) then there is a y in P that is the next number in P above x .

Let's see some examples of assessing the order properties of 1-place predicates and sets of real numbers.

Example (1): Two P -sets that are *densely ordered*:

$P_1x \equiv (x = x)$, i.e. x is any real number and $I(P_1) = \mathbb{R}$

$P_2x \equiv Qx$, i.e. x is any rational number and $I(P_2) = \mathbb{Q}$.

This is because for any real numbers $x, y \in \mathbb{R}$ (or alternatively, for

any rationals $x, y \in \mathbb{Q}$) if $x < y$ then we can take the mid-point $z = \frac{1}{2}(x + y)$, and we get $z \in \mathbb{R}$ (or we get $z \in \mathbb{Q}$) and $x < z < y$.

Example (2): a P-set that is *discretely ordered*:

$Px \equiv (\exists m)(Zm \ \& \ x = 5m)$, i.e. x is an integer multiple of 5.

This is because if $x = 5m$, we can take $y = 5(m+1)$; then y is the *next* element in the P-set larger than x .

Example (3):

$Px \equiv (x = \sqrt{2})$, i.e. $I(P) = \{\sqrt{2}\}$, single number.

The P-set here is *both* densely ordered and discretely ordered.

With this one-element P-set, take $x := \sqrt{2}$ and $y := \sqrt{2}$ in the formula for densely ordered sets, to make antecedent of the conditional false (because $(P\sqrt{2} \ \& \ P\sqrt{2} \ \& \ \sqrt{2} < \sqrt{2})$ is false) and thus make the conditional true. There are no other instances that make the predicate P true, so we end up with the \forall -quantified densely-ordered formula true.

Likewise, taking $x := \sqrt{2}$ in the formula for discretely ordered sets will make antecedent of the conditional false (because $(\exists w)(Pw \ \& \ \sqrt{2} < w)$ is false) and thus make the conditional true. There are no other instances that make the predicate P true, so we end up with the \forall -quantified discretely-ordered formula true.

Example (4):

$Px \equiv (-3 < x \ \& \ x \leq 2)$, i.e. $I(P) = (-3, 2]$ interval.

The P-set is densely ordered. This is because for any numbers $x, y \in I(P)$, if $x < y$ then take $z = \frac{1}{2}(x + y)$, so $z \in I(P)$ and $x < z < y$.

Example (5):

$Px \equiv (-3 < x \ \& \ x \leq 2) \vee x = 5$, i.e. $I(P) = (-3, 2] \cup \{5\}$.

The P-set is *neither* densely ordered *nor* discretely ordered.

Take $x := 2$ and $y := 5$ to give a false instance of the densely ordered formula (there is nothing in P between 2 and 5), while $x := 0$ gives a false instance of the discretely ordered formula (there is some number in P above 0 but there is no *next largest* number in P above 0 because it is in the densely-ordered part, the interval $(-3, 2]$).

After all those examples illustrating the various different possibilities, here some exercises for you.

Exercise (i) For the predicate P defined by:

$$Px \equiv (x^2 \geq 16),$$

that is, $I(P) = (-\infty, -4] \cup [4, \infty)$, a disjoint union of two infinite intervals,

determine whether:

- (a) The P -set is densely ordered. (c) Both (a) and (b)
- (b) The P -set is discretely ordered. (d) Neither (a) nor (b)

Answer = (d) The P -set is neither densely ordered nor discretely ordered. This is because $x = -4$ and $y = 4$ show failure of dense-ness (there is nothing between them), and take $x = 4$ for failure of discreteness (no next largest after 4).

Exercise (ii) For the predicate P defined by:

$$Px \equiv (\exists n)(\mathbb{N}n \ \& \ n > 0 \ \& \ x = 1 + \frac{1}{n})$$

that is, $I(P) = \{2, 1\frac{1}{2}, 1\frac{1}{3}, 1\frac{1}{4}, 1\frac{1}{5}, \dots, 1\frac{1}{n} \dots\}$,

determine whether:

- (a) The P -set is densely ordered. (c) Both (a) and (b)
- (b) The P -set is discretely ordered. (d) Neither (a) nor (b)

Answer = (b) The P -set is discretely ordered. This is because for $x = 1 + \frac{1}{n}$ in P , with $x \neq 2$ and $n \neq 1$, the next largest number in P is $y = 1 + \frac{1}{n-1}$, because $x = 1 + \frac{1}{n} < 1 + \frac{1}{n-1} = y$.

5.2.5 | CONTINUUM PROPERTY OF THE REAL NUMBER LINE

Denseness in the ordering does not fully capture what is usually meant by *analog* or *continuous* quantities. The *continuity* or **no gaps** property of the real number line is expressed by the **Continuum property**, also called the **Least Upper Bound property**, which says that the real ordering is one long (indeed, bi-infinitely long) *continuous line* or **continuum**.

The ordering \leq on \mathbb{R} satisfies the **Least Upper Bound** or **Continuum property**: every non-empty set of real numbers having an

upper bound must have a *least* upper bound.

For any 1-place predicate P denoted by the set $I(P)$ of real numbers, this formula is true in the standard model with domain \mathbb{R} .

LUB-P :

$$((\exists x)Px \ \& \ (\exists y)(\forall x)(Px \supset x \leq y)) \supset \\ (\exists z)((\forall x)(Px \supset x \leq z) \ \& \ (\forall w)((\forall x)(Px \supset x \leq w) \supset z \leq w))$$

In mathematical English, the formula **LUB-P** says: if the P set is non-empty and has an upper bound (there is a number y greater than or equal to every number in P), then there exists a z that is the least upper bound of the P set (z is an upper bound, and $z \leq w$ for all upper bounds w .)

Example (1):

For the predicate P defined below, determine the consequences of **LUB-P** by answering:

- (a) Is the P -set non-empty? Is $(\exists x)Px$ true?
- (b) Is the P -set bounded above? Is $(\exists y)(\forall x)(Px \supset x \leq y)$ true?
- (c) If answer ‘YES’ to both (a) and (b) then determine LUB of P -set.

$$Px \equiv (x \geq \frac{\pi}{2} \ \& \ x < 2\pi), \text{ i.e. } I(P) = [\frac{\pi}{2}, 2\pi], \text{ interval.}$$

Answer: The P -set is non-empty and bounded above (YES to (a) and (b)) and the LUB of the P -set is 2π . In this case, the LUB does not actually belong to the P -set.

Example (2):

For the predicate P defined below, determine the consequences of **LUB-P** by answering:

- (a) Is the P -set non-empty? Is $(\exists x)Px$ true?
- (b) Is the P -set bounded above? Is $(\exists y)(\forall x)(Px \supset x \leq y)$ true?
- (c) If answer ‘YES’ to both (a) and (b) then determine LUB of P -set.

$$Px \equiv x > 1, \text{ i.e. } I(P) = (1, \infty), \text{ infinite interval.}$$

Answer: The P -set is non-empty (YES to (a)) but it is not bounded above (NO to (b)), so there is no LUB for the P -set.

Exercise (i): For the 1-place predicate P defined by:

$$Px \equiv x^2 < 2 \text{ i.e. } -\sqrt{2} < x \text{ and } x < \sqrt{2}, \\ \text{so } I(P) = (-\sqrt{2}, \sqrt{2}), \text{ open interval centred at 0.}$$

determine the consequences of **LUB-P** by answering:

- (a) Is the P -set non-empty? Is $(\exists x)Px$ true?
- (b) Is the P -set bounded above? Is $(\exists y)(\forall x)(Px \supset x \leq y)$ true?
- (c) Which of these are correct:
 [A] No LUB [B] LUB = 2
 [C] LUB = $\sqrt{2}$ [D] LUB = 0

Solution to Exercise (i): The P -set is non-empty and bounded above (YES to (a) and (b)) and the LUB of the P -set is choice [C]: $\sqrt{2}$. In this case, the LUB does not actually belong to the P -set.

Exercise (ii): For the 1-place predicate P defined by:

$$Px \equiv (\exists n)(Nn \And n > 0 \And x = 1 - \frac{1}{n}) \\ \text{so } I(P) = \{0, \frac{1}{2}, \frac{2}{3}, \frac{3}{4}, \frac{4}{5}, \dots, \frac{n-1}{n}, \dots\}$$

determine the consequences of **LUB-P** by answering:

- (a) Is the P -set non-empty? Is $(\exists x)Px$ true?
 YES NO
- (b) Is the P -set bounded above? Is $(\exists y)(\forall x)(Px \supset x \leq y)$ true?
 YES NO
- (c) Which of these are correct:
 [A] No LUB [B] LUB = 1
 [C] LUB = 0 [D] LUB = 2

Solution to Exercise (ii): The P -set is non-empty and bounded above (YES to (a) and (b)) and the LUB of the P -set is choice [B]: 1. Again, the LUB does not actually belong to the P -set.

5.3 | DISTANCE PREDICATES ON THE REAL NUMBERS

The **distance** $d(x, y)$ between two real numbers x and y is the length of the line segment connecting x and y :

$$d(x, y) = \begin{cases} y - x & \text{if } x < y \\ 0 & \text{if } x = y \\ x - y & \text{if } x > y \end{cases}$$

Now, we have developed predicate logic without using functions, but that won't be a problem because what we really need is a *bound* or *estimate* of the distance between two numbers.

We will use a 3-place predicate $Dxy\epsilon$ to mean the distance between x and y is strictly smaller than ϵ .

$$(\forall x)(\forall y)(\forall \epsilon) (Dxy\epsilon \equiv (x < y \& y < x + \epsilon) \vee (x = y \& \epsilon > 0) \vee (y < x \& x < y + \epsilon))$$

For example, the following literals (atomic formulas or their negations) are true in the standard model over \mathbb{R} (adding commas for clarity).

D 17, 13, 4.1	~D 17, 13, 4
D 17, 13, 4.00001	~D 17, 13, 0
D 3.1, 8.7, 5.6001	~D 3.1, 8.7, 5.6
D -9, 12, 22	~D -9, 12, 20

Note that $D 17, 13, 4$ is false because the distance between 17 and 13 is exactly 4, but if we replace 4 by 4.1 or 4.00001 or any number larger than 4, then we get a true atomic formula. All that is needed for the truth of $Dab\epsilon$ is that the third number for ϵ is larger than the distance between the first two numbers a and b .

The following formulas express properties of the *distance-is-less-than* predicate D and are true in the standard model with domain \mathbb{R} .

- (i) $(\forall x)(\forall y)(\forall \epsilon)(Dxy\epsilon \supset \epsilon > 0)$
- (ii) $(\forall x)(\forall y)(\exists \delta)(\delta \geq 0 \& (\forall \epsilon)(\epsilon > \delta \supset Dxy\epsilon) \& (\forall \alpha)(\alpha \leq \delta \supset \sim Dxy\alpha))$
- (iii) $(\forall x)(\forall y)((\forall \epsilon)(\epsilon > 0 \supset Dxy\epsilon) \equiv (x = y))$
- (iv) $(\forall x)(\forall y)(\forall \epsilon)(Dxy\epsilon \equiv Dyx\epsilon)$

- (v) $(\forall x)(\forall y)(\forall \varepsilon)(\forall z)(\forall \delta)(\forall \alpha)$
 $((D_{xy}\varepsilon \ \& \ D_{yz}\delta \ \& \ \varepsilon + \delta \leq \alpha) \supset D_{xz}\alpha)$
- (vi) $(\forall x)(\forall y)(\forall \varepsilon)(\forall \delta)((D_{xy}\varepsilon \ \& \ \delta > \varepsilon) \supset D_{xy}\delta)$
- (vii) $(\forall x)(\forall y)(\forall \varepsilon)((x < y \ \& \ D_{xy}\varepsilon) \supset$
 $(0 < y - x \ \& \ y - x < \varepsilon))$
- (viii) $(\forall x)(\forall y)(\forall z)(\forall \varepsilon)(D_{xy}\varepsilon \equiv D_{x+z,y+z,\varepsilon})$
- (ix) $(\forall x)(\forall y)(\forall \varepsilon)(D_{xy}\varepsilon \equiv D_{-x,-y,\varepsilon})$
- (x) $(\forall x)(\forall y)(\forall z)(\forall \varepsilon)(z > 0 \supset (D_{xy}\varepsilon \equiv D_{zx,zy,z\varepsilon}))$

Formula (i) expresses that whenever $D_{xy}\varepsilon$ is true, we must have ε greater than zero, because the distance between x and y is always zero or greater. Formula (ii) expresses that for all x and y , there is a number δ that is the actual distance between x and y , and this is shown by the fact that $D_{xy}\varepsilon$ is true for all ε strictly greater-than this δ , and $\sim D_{xy}\alpha$ is true for all α less-than-or-equal-to δ . Formula (iii) expresses that for all x and y , x is identical to y if and only if $D_{xy}\varepsilon$ is true for all ε greater than 0. Paraphrased: the distance is less than every positive number ε exactly when the two numbers are identical.

Formula (iv) expresses symmetry in the x and y arguments. Formula (v) expresses what is known as the *triangle property* for distance. If $D_{xy}\varepsilon$ and $D_{yz}\delta$, then we know we can get from x to z via y , and thus conclude that $D_{xz}\alpha$ holds whenever α is greater than the sum of ε and δ . Formula (vi) expresses that the D predicate is *monotone* in its third place: if $D_{xy}\varepsilon$ and δ is greater-than ε , then $D_{xy}\delta$ is true.

Looking again at the definition of the distance-is-less-than predicate, we see that it has an equivalent formulation using the *difference* between x and y , which is the greater of $(y - x)$ and $(x - y)$.

$$(\forall x)(\forall y)(\forall \varepsilon) (D_{xy}\varepsilon \equiv (0 < y - x \ \& \ y - x < \varepsilon) \vee (x = y \ \& \ \varepsilon > 0) \vee (0 < x - y \ \& \ x - y < \varepsilon)).$$

This formulation is derivable using property (vii) together with the symmetry property (iv).

Formula (viii) expresses that the D predicate is *invariant under addition* to the first two arguments. For any real number z , we have $D_{xy}\varepsilon$ if and only if $D_{x+z,y+z,\varepsilon}$. Formula (ix) expresses that the D predicate is *invariant under the minus operation* to the first arguments: $D_{xy}\varepsilon$ if and only if $D_{-x,-y,\varepsilon}$. The last in this list, formula (x), expresses that the D predicate *scales under multiplication by a positive real* in all three arguments. For all positive real numbers $z > 0$, we have $D_{xy}\varepsilon$ if and only if $D_{zx,zy,z\varepsilon}$.

A first basic skill is to be able to read and interpret predicate logic formulas containing D .

Example (1): Consider the following formula containing the *distance is-less-than* predicate:

$$(\forall x)(\forall y) \sim Dxy0$$

Which of these sentences of mathematical English best capture its meaning:

- (a) The distance between any two reals is greater than 0
- (b) The distance between any two reals is never less than 0.
- (c) The distance between any two reals is never equal to 0.

Answer: (b) $Dxy0$ means the distance between x and y is strictly less than 0.

Example (2): Continuing with the same predicate logic formula.

$$(\forall x)(\forall y) \sim Dxy0$$

Which of these assessments of the truth of the formula in the standard model of the real numbers is correct:

- (a) The formula is **true** because for any choice of real numbers $x := a$ and $y := b$, if $Dab0$ was true, then since $(Dab\varepsilon \supset \varepsilon > 0)$, from $\varepsilon := 0$ we would get $0 > 0$, which is a contradiction; so $Dab0$ must be false.
- (b) The formula is **false** because if we take $x := 1$ and $y := 1$, the atomic formula $D110$ is true, so $\sim D110$ is false.

Answer: (a) $\sim Dab0$ is true for all choices of $a, b \in \mathbb{R}$.

Now it is your turn to have a go.

Exercise (i): Consider the following formula:

$$(\forall x)(\forall y)(\forall \varepsilon)(Dxy\varepsilon \supset (0 < y - x \ \& \ y - x < \varepsilon))$$

“For all reals x, y and ε , if the distance between x and y is less than ε , then the difference $y - x$ is strictly between 0 and ε .”

Which of these assessments of the truth of the formula in the standard model of the real numbers is correct:

- (a) The formula is **false** because if we take $x := 1$ and $y := 1$ and $\varepsilon := 0$, then D110 is true but $(0 < 1 - 1 \ \& \ 1 - 1 < 0)$ is false.
- (b) The formula is **false** because if we take $x := 2$ and $y := 1$ and $\varepsilon := 2$, then D212 is true but $(0 < 1 - 2 \ \& \ 1 - 2 < 2)$ is false.
- (c) The formula is **true** because $Dxy\varepsilon$ means the difference between x and y is strictly less than ε and is greater than 0.

The formula is **false** and (b) is the correct answer, because if we take $x := 2$ and $y := 1$ and $\varepsilon := 2$, then D212 is true but $(0 < 1 - 2 \ \& \ 1 - 2 < 2)$ is false, because $1 - 2 = -1$ and the atomic formula $0 < -1$ is false.

Exercise (ii): Consider the similar formula:

$$(\forall x)(\forall y)(\forall \varepsilon)((Dxy\varepsilon \ \& \ x < y) \supset (0 < y - x \ \& \ y - x < \varepsilon))$$

Which of these assessments of the truth of the formula in the standard model of the real numbers is correct:

- (a) The formula is **false** because if we take $x := 2$ and $y := 1$ and $\varepsilon := 2$, then D212 is true but $2 < 1$ and $(0 < 1 - 2 \ \& \ 1 - 2 < 2)$ are both false.
- (b) The formula is **true** because for any choice of reals $x := a$ and $y := b$, and for all ε , $Dab\varepsilon$ implies $\varepsilon > 0$.
- (c) The formula is **true** because from the definition of the predicate D , for any choice of reals $x := a$ and $y := b$, and for all ε , the following conditional is true:

$$(Dab\varepsilon \ \& \ a < b) \supset (0 < b - a \ \& \ b - a < \varepsilon).$$

$$\cdot (z > a - b \ \& \ z > 0) \subset (0 < b - a \ \& \ b - a < z).$$

the answer is (c): the formula is **true** because from the definition of the predicate D , for any choice of reals $x := a$ and $y := b$, and for all ε , the following conditional is true:

$$((z > x - y \ \& \ z > 0) \subset (y > x - z \ \& \ z > y))$$

For the formula:

Next, we want to relate the distance predicate D to **densely-ordered** and **discretely-ordered** sets defined by 1-place predicates.

The conditional:

$$\begin{aligned} (D1) \quad & ((\forall x)(\forall y)((Px \ \& \ Py \ \& \ x < y) \supset (\exists z)(Pz \ \& \ x < z \ \& \ z < y)) \\ & \quad \& (\exists x)(\exists y)(Px \ \& \ Py \ \& \ x < y)) \\ & \supset (\forall x)(\forall \varepsilon)((Px \ \& \ \varepsilon > 0) \supset (\exists z)(Pz \ \& \ z \neq x \ \& \ Dxze)) \end{aligned}$$

is **true** in the standard model with domain \mathbb{R} .

The antecedent of the conditional (D1) says that the P-set is **densely ordered** and is of size ≥ 2 , while the consequent of (D1) says the P-set satisfies a *distance-dense*ness property:

for all reals x in P and for all real $\varepsilon > 0$ (no matter how small), there exists a real number z in P different from x such that the distance between x and z is less than ε .

Paraphrased: given any number x in P, we can find numbers z in P different from x that are *arbitrarily* close (as close as you want) to x .

Explanation for truth of conditional (D1): Suppose the P-set $I(P)$ is densely ordered and has at least two numbers in it. Then let $a \in \mathbb{R}$ and $e \in \mathbb{R}$ be any real numbers, and suppose $a \in I(P)$ (Pa true) and $e > 0$. We want to show that $(\exists z)(Pz \ \& \ z \neq x \ \& \ Daze)$ is true. The idea is to start with another number $b \in I(P)$ such that $a < b$ [or $b < a$ if a happens to be the largest number in the set $I(P)$], and keep searching through numbers $c \in I(P)$ such that $a < c < b$ (and hence $c \neq a$) until we find a number c such that $Dace$ is true.

Turning our attention now to distance in discretely ordered sets, consider the following conditional statement which we will label (D3): If a P-set is **discretely ordered**:

$$\begin{aligned} (\forall x)((Px \ \& \ (\exists w)(Pw \ \& \ x < w)) \supset \\ (\exists y)(Py \ \& \ x < y \ \& \ \sim(\exists z)(Pz \ \& \ x < z \ \& \ z < y))) \end{aligned}$$

then the P-set has a **distance-discreteness** property which says: **there is a number $\delta > 0$ such that any two numbers in the P-set are at least distance δ apart**:

$$(\exists \delta)(\delta > 0 \ \& \ (\forall x)(\forall y)((Px \ \& \ Py \ \& \ x \neq y) \supset \sim Dxy\delta)).$$

The conditional (D3) sounds *plausible* if we think of densely ordered sets as like the natural numbers \mathbb{N} or the integer multiples of 5 – where there is a fixed distance between any two numbers in the set.

... But conditional (D3) is **false!** Recall from §5-2, **Exercise (ii)**, the discretely ordered P-set defined by:

$$Px \equiv (\exists n)(Nn \ \& \ n > 0 \ \& \ x = 1 + \frac{1}{n})$$

so $I(P) = \{2, 1\frac{1}{2}, 1\frac{1}{3}, 1\frac{1}{4}, 1\frac{1}{5}, \dots, 1\frac{1}{n}, \dots\}$. For this set $I(P)$, we **cannot** find a positive distance $\delta > 0$ such that any two numbers in the P-set are at

least distance δ apart, because the numbers in the P-set get closer and closer to 1, and so closer and closer to each other. If we did put forth a candidate distance $\delta > 0$, it could be easily shown to fail by finding a large enough natural number $n \in \mathbb{N}$ such that $\frac{1}{n} < \delta$ – by choosing $n > \frac{1}{\delta}$. Then $a := 1 + \frac{1}{n}$ and $b := 1 + \frac{1}{n+1}$ are two different numbers in the set $I(P)$ such that $Dab\delta$ is true, and so $\sim Dab\delta$ is false.

The *converse* conditional which we will label (D4):

If a P-set has the **distance-discreteness** property:

$$(\exists \delta) (\delta > 0 \ \& \ (\forall x)(\forall y)((Px \ \& \ Py \ \& \ x \neq y) \supset \sim Dxy\delta)) .$$

then that P-set is **discretely ordered**:

$$(\forall x)((Px \ \& \ (\exists w)(Pw \ \& \ x < w)) \supset \\ (\exists y)(Py \ \& \ x < y \ \& \ \sim(\exists z)(Pz \ \& \ x < z \ \& \ z < y)))$$

is **true** in the standard model with domain \mathbb{R} .

Explanation for truth of conditional (D4): Suppose the P-set $I(P)$ has the **distance-discreteness** property. Then let $a \in \mathbb{R}$ be any real number, and suppose that $a \in I(P)$ and a is not the largest number in $I(P)$. Then there must exist another number $b \in I(P)$ such that $a < b$. Applying the distance-discreteness property, there exists a positive number $d > 0$ such that, with $x := a$ and $y := b$, we have $\sim Dabd$. Moreover, the distance-discreteness property ensures that we can choose the number b so that it is the *smallest* number $b > a$ such that $b \in I(P)$. Then we claim $\sim(\exists z)(Pz \ \& \ a < z \ \& \ z < b)$ must be true. To see why the claim must be true, suppose otherwise: suppose there exists a number $c \in I(P)$ such that $(a < c \ \& \ c < b)$. But then this contradicts our choice of b as the *smallest* number $b > a$ such that $b \in I(P)$. So no such number c can exist. Since the number $a \in I(P)$ was arbitrary, we can conclude the truth of the universally quantified formula expressing that the P-set is **discretely ordered**, as required.

5.3.1 | FUNCTIONS AND RELATIONS

This sub-section is optional and provided for those students who have studied more mathematics and are familiar with functions.

A 1-place **function** is a binary (2-place) relation R such that for every first thing a , there is exactly one (a unique) second thing b such that Rab .

A 2-place **function** is a 3-place relation S such that for every pair of things (a, b) , there is exactly one (a unique) third thing c such that the three things in order are related $Rabc$.

In a mathematical context, we need to specify the *sets* involved in a function.

Let X and Y be sets. f is a *function* from the set X to the set Y iff for every element $x \in X$, there is a unique corresponding element $f(x) \in Y$ associated with x . We write $f : X \rightarrow Y$ to indicate f is a function from X to Y .

For example, the arithmetic operations are functions on \mathbb{R} :

<i>addition</i>	$a : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$	write $x + y$ instead of $a(x, y)$
<i>subtraction</i>	$s : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$	write $x - y$ instead of $s(x, y)$
<i>multiplication</i>	$m : (\mathbb{R} \times \mathbb{R}) \rightarrow \mathbb{R}$	write xy or $x \cdot y$ instead of $m(x, y)$
<i>squaring</i>	$q : \mathbb{R} \rightarrow \mathbb{R}$	write x^2 instead of $q(x)$

Our 3-place distance-is-less-than predicate $Dxy\varepsilon$ is equivalent to $d(x, y) < \varepsilon$, where d is the distance **function** on \mathbb{R} .

Define a 2-place function $d : \mathbb{R}^2 \rightarrow \mathbb{R}$ such that for all pairs $(x, y) \in \mathbb{R}^2$, the number $d(x, y)$ satisfies:

$$d(x, y) = \begin{cases} y - x & \text{if } x < y \\ 0 & \text{if } x = y \\ x - y & \text{if } y < x \end{cases}$$

This is an instance of *definition by cases*; for a function to be well-defined in this manner, one has to ensure that the cases are *exhaustive* of all possible values of the argument (here, $(x, y) \in \mathbb{R}^2$), and *exclusive* in the sense that any one value of the argument falls in exactly one case.

The following formulas expressing properties of the distance function d for \mathbb{R} are true in the standard model with domain \mathbb{R} .

- (i) $(\forall x)(\forall y)(d(x, y) \geq 0)$
- (ii) $(\forall x)(\forall y)((d(x, y) = 0) \equiv (x = y))$
- (iii) $(\forall x)(\forall y)(d(x, y) = d(y, x))$
- (iv) $(\forall x)(\forall y)(\forall z)(d(x, z) \leq d(x, y) + d(y, z))$

Formula (iv) is known as the *triangle inequality*.

A *metric* is defined as any 2-place function d satisfying the four properties (i), (ii), (iii) and (iv). Additional properties of the standard distance function d on \mathbb{R} include the following:

- (v) $(\forall x)(\forall y)((x \leq y \supset d(x, y) = y - x) \& (x > y \supset d(x, y) = x - y))$
- (vi) $(\forall x)(\forall y)(\forall z)(d(x + z, y + z) = d(x, y))$
- (vii) $(\forall x)(\forall y)(\forall z)(z \geq 0 \supset d(zx, zy) = z \cdot d(x, y))$
- (viii) $(\forall x)(\forall y)(d(-x, -y) = d(x, y))$

5.4 | SEQUENCES OF REAL NUMBERS

We begin the section with the basic definition.

Definition: Any infinite list of real numbers,

$$a_0, a_1, a_2, \dots$$

is called a *sequence*. We use the notation $(a_n)_{n \in \mathbb{N}}$ to indicate an infinite sequence, with natural numbers $n \in \mathbb{N}$ the *index* of the entries in the sequence.

We have previously seen finite-length sequences: in giving the semantics in a model $M = (D, I)$ of an k -place predicate F with atomic formulas $F a_1 a_2 \dots a_k$, we use the length- k sequences or k -tuples of objects $(I(a_1), I(a_2), \dots, I(a_k))$ in D^k , for predicate arities $k = 1, 2, 3, \dots$

An infinite sequence is a list that *keeps on going and going and going ...*, where the entries a_n are generated according to some rule.

Example (1): The famous Fibonacci sequence,

n	0	1	2	3	4	5	6	7	...
a_n	1	1	2	3	5	8	13	21	...

is obtained by letting the first entries be $a_0 = 1$ and $a_1 = 1$, and then recursively computing according to the rule:

$$a_{n+2} = a_{n+1} + a_n.$$

The natural world exhibits Fibonacci numbers in multiple ways, such as the number of petals on a flower, or the growth pattern of branches from the main stem of a plant, or the pattern of scales in a pine-cone. The numbers in the Fibonacci sequence get larger and larger without bound; it “goes to infinity”.

Alternatively, some sequences of real numbers get closer and closer to a fixed number; they “converge to a limit”.

Example (2): Suppose we let $a_n = \frac{10^n - 1}{10^n}$ for all $n \in \mathbb{N}$. Then we would obtain the sequence beginning with:

$$(0, \frac{9}{10}, \frac{99}{100}, \frac{999}{1000}, \frac{9999}{10000}, \frac{99999}{100000}, \frac{999999}{1000000}, \dots)$$

which in decimal notation is:

$$(0, 0.9, 0.99, 0.999, 0.9999, 0.99999, 0.999999, \dots)$$

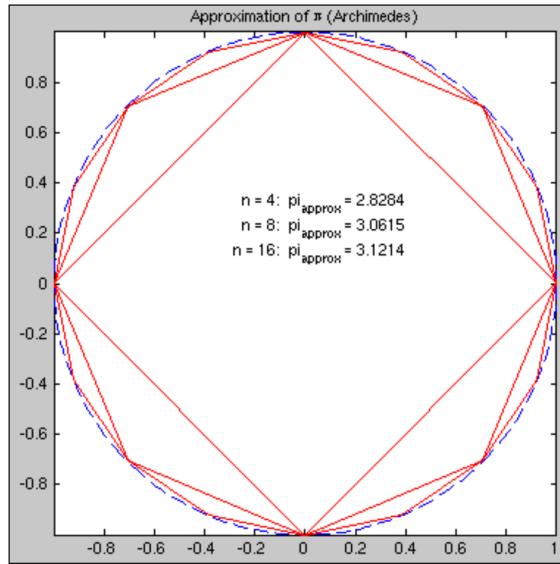
For this sequence, it is clear that the limit must be

$0.999999999999999\dots$ with infinitely repeating digit 9, written $0.\bar{9}$.

which is equal to the number 1, since $(\forall \varepsilon)(\varepsilon > 0 \supset D 0.\bar{9} 1 \varepsilon)$ is true, and apply the *distance-is-less-than* predicate property:

$$(\forall x)(\forall y)((\forall \varepsilon)(\varepsilon > 0 \supset D x y \varepsilon) \equiv (x = y))$$

to conclude that $0.\bar{9} = 1$.



Example (3): Recall the Archimedean approximation of π . As shown in the diagram above, use a circle with perimeter (circumference) of 2π when its radius is 1 and diameter is 2. An approximation sequence $(p_n)_{n \in \mathbb{N}}$ for π is obtained by the rule:

$$p_n = \frac{1}{2}(s_n \times l_n);$$

which says: p_n is equal to $\frac{1}{2}$ the perimeter of inscribed polygon with s_n -many sides of length l_n , where:

$$s_0 := 4$$

$$l_0 := \sqrt{2}$$

$$s_{n+1} := 2s_n$$

$$l_{n+1} := \sqrt{2 - \sqrt{4 - l_n^2}}$$

The sequence $(p_n)_{n \in \mathbb{N}}$ has *limit* π .

This approximation sequence sequence $(p_n)_{n \in \mathbb{N}}$ becomes accurate to 2 decimal places by stage $n = 3$, when the polygon has 32 sides (one more stage than is illustrated in the diagram).

Example (4): Consider the sequence which has $a_n = \frac{1}{n+1}$ when n is even and $a_n = -\frac{1}{n+1}$ when n is odd. Then the sequence begins with:

n	0	1	2	3	4	5	6	7	...
a_n	1	$-\frac{1}{2}$	$\frac{1}{3}$	$-\frac{1}{4}$	$\frac{1}{5}$	$-\frac{1}{6}$	$\frac{1}{7}$	$-\frac{1}{8}$...

Even though the sequence entries in **Example (4)** alternate between positive and negative, so lie to the right and to the left of 0, the entries a_n get closer and closer in distance to 0 as $n \in \mathbb{N}$ gets larger and larger.

Example (5): Consider the sequence beginning with $a_0 = 0$ and $a_1 = 1$, which continues according to the recursive rule:

$$a_{n+2} = \frac{1}{2}(a_{n+1} + a_n).$$

Each entry is the *average* of the two previous entries, so $(a_n)_{n \in \mathbb{N}}$ is a *repeated average* or *moving average* sequence.

The sequence in **Example (5)** begins with:

$$(0, 1, 0.5, 0.75, 0.625, 0.6875, 0.65625, 0.671875, \\ 0.6640625, 0.66796875, \dots)$$

as its first 10 entries (getting up to 8 decimal places).

The recursive rule $a_{n+2} = \frac{1}{2}(a_{n+1} + a_n)$ with initial conditions $a_0 = 0$ and $a_1 = 1$ describes a 2nd order difference equation with initial conditions, which can be solved to give the explicit solution:

$$a_n = \begin{cases} \frac{2}{3} \left(1 - \frac{1}{2^n} \right) & \text{if } n \in \mathbb{N} \text{ is even;} \\ \frac{2}{3} \left(1 + \frac{1}{2^n} \right) & \text{if } n \in \mathbb{N} \text{ is odd.} \end{cases}$$

Based on the explicit solution as given, together with our list of the first 10 entries, which of these options for $(a_n)_{n \in \mathbb{N}}$ seems intuitively correct (we will make this precise in the next section):

- (a)** $(a_n)_{n \in \mathbb{N}}$ approaches 1
- (b)** $(a_n)_{n \in \mathbb{N}}$ approaches $\frac{2}{3}$
- (c)** $(a_n)_{n \in \mathbb{N}}$ approaches $-\frac{2}{3}$
- (d)** $(a_n)_{n \in \mathbb{N}}$ does not have a limit.

Answer: (b) From the explicit description, we can see that the entries a_n get closer and closer in distance to $\frac{3}{2}$ as $n \in \mathbb{N}$ gets larger and larger — even though the entries alternate between being a little below $\frac{3}{2}$ and a little above $\frac{3}{2}$.

Recall the definition of the 3-place predicate $Dx\gamma\varepsilon$ expressing that the numbers x and y are less than ε apart in distance:

$$(\forall x)(\forall y)(\forall \varepsilon) (Dx\gamma\varepsilon \equiv (x < y \wedge y < x + \varepsilon) \vee (x = y \wedge \varepsilon > 0) \vee (y < x \wedge x < y + \varepsilon))$$

We will use this 3-place predicate D plus quantifiers to precisely define what it means to say an infinite sequence $(a_n)_{n \in \mathbb{N}}$ of real numbers converges to a limit $r \in \mathbb{R}$. The ε variable in atomic formulas $Da_n r \varepsilon$ will give a bound on the accuracy or error margin — since it is a bound on the distance between a sequence entry a_n and the target real number r . We will need to consider smaller and smaller values of $\varepsilon > 0$. For example, consider ε in a progression such as $\frac{1}{10}, \frac{1}{100}, \frac{1}{1000}, \frac{1}{10000}, \dots, \frac{1}{10^k}$ for larger and larger natural numbers $k \in \mathbb{N}$.

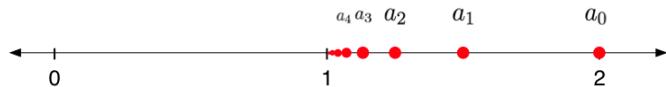
5.5 | CONVERGENCE AND DIVERGENCE OF SEQUENCES OF REALS

5.5.1 | CONVERGENCE OF SEQUENCES OF REALS

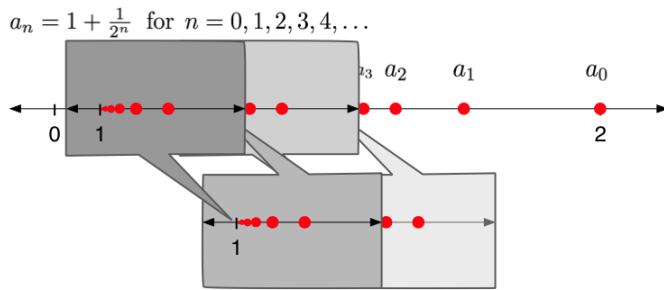
Going back to the very first example in §5-1, the sequence $(a_n)_{n \in \mathbb{N}}$ is such that:

$$a_n = 1 + \frac{1}{2^n}.$$

$$a_n = 1 + \frac{1}{2^n} \text{ for } n = 0, 1, 2, 3, 4, \dots$$



So the sequence starts at 2 and decreases: $1\frac{1}{2}, 1\frac{1}{4}, 1\frac{1}{8}, 1\frac{1}{16}, 1\frac{1}{32}$, and so on. Drawn here on the number line, we see the terms in our example sequence as red dots getting closer and closer to 1. At the scale we start with, the red dots get blurry after term a_4 . The meaning of this phrase “getting closer and closer” is that we can *zoom in* or *magnify* close to the limit ...



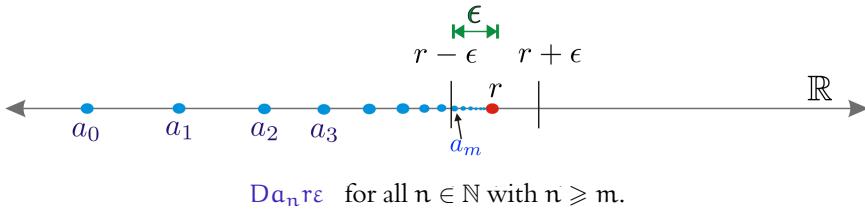
The notion of a limit of a sequence is that we could zoom in or magnify *infinitely many times*, and each time, we would still see the same behaviour of sequence terms getting closer and closer to the limit number.

Now to the definition; be warned: we need *three* alternations of quantifiers: *for all-there exists-for all*.

Definition: A real number sequence $(a_n)_{n \in \mathbb{N}}$ is said to *converge* to a real number r , and r is called its *limit* iff

$$(\forall \varepsilon)(\varepsilon > 0 \supset (\exists m)(\forall n)((\forall n & n \geq m) \supset D a_n r \varepsilon))$$

The definition reads: A real number sequence $(a_n)_{n \in \mathbb{N}}$, is said to *converge* to a real number r , and r is called the *limit* of the sequence, if and only if for all $\varepsilon > 0$, there exists a natural number $m \in \mathbb{N}$ such that for all natural numbers $n \geq m$, we have that $D a_n r \varepsilon$ is true.



To paraphrase the definition of convergence into slightly smoother English: The real number r is such that, for every positive $\varepsilon > 0$, no matter how small, we can find a cut-off index $m \in \mathbb{N}$ (depending on ε) such that from position m onwards in the sequence — for all entries a_n for $n \geq m$ — the distance between a_n and r is less than ε ; *in particular, at the cut-off index m , the entry a_m is an approximation of r to accuracy ε* .

The concept of convergence of a sequence is crucial within modern mathematics and its applications in science, engineering, economics, medicine, etc. — but note the large number of quantifiers involved: more than is typical in natural language. Indeed, some in linguistics have argued that we humans have a natural limit of three or four alternations of quantifiers: any more than that, and roughly speaking, our heads explode!

It is common for mathematics undergraduate students to struggle to follow the *logical structure* of the definition of convergence, as they receive much less logical training than students taking this course.

Using the notational shorthand for *restricted quantifiers*, that logical structure can be made even clearer.

5.5.2 | RESTRICTED QUANTIFIER NOTATION

Formulas of the form:

$$(\forall \varepsilon)(\varepsilon > 0 \supset A)$$

are instances of *restricted universal quantification* because the combination of the $(\forall \varepsilon)$ with the condition $\varepsilon > 0$ in the antecedent of a conditional means that we can understand *the \forall as ranging over a restricted subset of numbers* — here, the set of all strictly positive numbers $\varepsilon > 0$ — rather than ranging over the whole domain (real number line \mathbb{R}). This can be written more concisely in restricted quantifier notation as:

$$(\forall \varepsilon > 0) A .$$

The net effect of this combination is that the universal quantification is in fact ranging over just the positive real numbers, and not the whole real number line. If we take any instance of this formula where the ε is less than or equal to 0, then that instance formula will come out as true for the trivial reason that the antecedent of the conditional will be false. So we can safely narrow our focus to just the restricted subset of positive real numbers. Using the restricted universal quantifier notation, the formula is read as: for all ε greater-than 0, A holds.

Formulas of the form:

$$(\exists m)(Nm \ \& \ B)$$

are instances of *restricted existential quantification* because the combination of the $(\exists m)$ with the condition Nm as one half of a conjunction means that we can understand *the \exists as ranging over a restricted subset of numbers* — here, the set \mathbb{N} of all natural numbers — rather than ranging over the whole domain (real number line \mathbb{R}). This can be written more concisely in restricted quantifier notation as:

$$(\exists m \in \mathbb{N}) B .$$

The net effect of this combination is that the existential quantification is in fact ranging over just the natural numbers, and not the whole real number line. Here, if we take any instance of this formula where the m is *not* a natural number, then that instance formula will come out as false because the atom Nm will be false. So nothing in the complementary set of non-natural numbers can contribute to this existential quantification, so we can safely narrow our focus to just the restricted subset of natural numbers. Using the restricted existential quantifier

notation, the formula is read as: there exists a natural number m such that B holds.

Using the restricted quantifier notation, convergence of a sequence in \mathbb{R} , can be rewritten more concisely.

A sequence $(a_n)_{n \in \mathbb{N}}$ *converges* to the number r iff

$$(\forall \varepsilon > 0)(\exists m \in \mathbb{N})(\forall n \in \mathbb{N})(n \geq m \supset D a_n r \varepsilon).$$

5.5.3 | DIVERGENCE OF SEQUENCES OF REALS

Definition: A sequence $(a_n)_{n \in \mathbb{N}}$ is said to *diverge* iff there does not exist any real number $x \in \mathbb{R}$ such that $(a_n)_{n \in \mathbb{N}}$ converges to x .

Divergence of sequence $(a_n)_{n \in \mathbb{N}}$ is expressed by the formula:

$$\sim(\exists x)(\forall \varepsilon)(\varepsilon > 0 \supset (\exists m)(Nm \& (\forall n)(Nm \& n \geq m \supset D a_n x \varepsilon))).$$

and this is in turn equivalent to:

$$(\forall x)(\exists \varepsilon)(\varepsilon > 0 \& (\forall m)(Nm \supset (\exists n)(Nm \& n \geq m \& \sim D a_n x \varepsilon))).$$

A sequence $(a_n)_{n \in \mathbb{N}}$ is said to *diverge* iff

$$(\forall x)(\exists \varepsilon)(\varepsilon > 0 \& (\forall m)(Nm \supset (\exists n)(Nm \& n \geq m \& \sim D a_n x \varepsilon))).$$

To paraphrase: For every choice of real number $x \in \mathbb{R}$, there exists a positive distance $\varepsilon > 0$ such that for every sequence index $m \in \mathbb{N}$, there exists a larger index $n \geq m$ such that there is at least distance ε between the n -th entry a_n and the chosen number x ; this means *for all $x \in \mathbb{R}$, there exists $\varepsilon > 0$ such that there are infinitely many entries a_n in the sequence that are at least distance ε away from x* .

Likewise, divergence of a sequence in \mathbb{R} can be rewritten more concisely using restricted quantifier notation.

A sequence $(a_n)_{n \in \mathbb{N}}$ *diverges* iff

$$(\forall x)(\exists \varepsilon > 0)(\forall m \in \mathbb{N})(\exists n \in \mathbb{N})(n \geq m \& \sim D a_n x \varepsilon).$$

Now for some examples of convergence and divergence.

Example (1): Consider the sequence:

$$(a_n)_{n \in \mathbb{N}} = (1, \frac{1}{2}, \frac{1}{3}, \frac{1}{4}, \dots, \frac{1}{n+1}, \dots)$$

that is, $a_n = \frac{1}{n+1}$.

Determine whether:

- (a) $(a_n)_{n \in \mathbb{N}}$ converges to limit 1.
- (b) $(a_n)_{n \in \mathbb{N}}$ converges to limit 0.
- (c) $(a_n)_{n \in \mathbb{N}}$ converges to limit -1.
- (d) $(a_n)_{n \in \mathbb{N}}$ diverges.
- (e) None of the above.

Answer: (b) The sequence $(a_n)_{n \in \mathbb{N}}$ converges to limit 0.

As $n \in \mathbb{N}$ gets larger and larger, the entry $a_n = \frac{1}{n+1}$ gets smaller and smaller, and approaches 0. Given $\varepsilon > 0$, take $m \geq \frac{1}{\varepsilon}$ so that $\frac{1}{m+1} < \frac{1}{m} \leq \varepsilon$; then $|a_n - 0| < \varepsilon$ is true for all $n \geq m$.

Example (2): Consider the sequence:

$$(a_n)_{n \in \mathbb{N}} = (1, -1, 1, -1, \dots, (-1)^n, \dots)$$

that is, $a_n = (-1)^n$, so $a_n = 1$ if n is even, and $a_n = -1$ if n is odd.

Determine whether:

- (a) $(a_n)_{n \in \mathbb{N}}$ converges to limit 1.
- (b) $(a_n)_{n \in \mathbb{N}}$ converges to limit 0.
- (c) $(a_n)_{n \in \mathbb{N}}$ converges to limit -1.
- (d) $(a_n)_{n \in \mathbb{N}}$ diverges.
- (e) None of the above.

Answer: (d) The sequence $(a_n)_{n \in \mathbb{N}}$ diverges. For any $x \in \mathbb{R}$, if $x \geq 0$ then there is a distance of $\varepsilon \geq 1$ between x and all the odd entries $a_{2n+1} = -1$, while if $x < 0$ then there is a distance $\varepsilon \geq 1$ between x and all the even entries $a_{2n} = 1$.

Notice that the argument for divergence takes a bit of effort. This breaking into the two cases of $x \geq 0$ and $x < 0$ is typical of the reasoning required when thinking through divergence.

Now it is your turn to try some exercises.

Exercise (i): Consider the sequence:

$$(a_n)_{n \in \mathbb{N}} = (0, 1, \frac{1}{2}, \frac{2}{3}, \frac{4}{5}, \dots, \frac{b_n - 1}{b_n}, \dots)$$

where $(b_n)_{n \in \mathbb{N}}$ is the Fibonacci sequence. That is, $a_0 = 0$, $a_1 = 1$ and for $n \geq 2$, we have $a_n = \frac{b_{n-1}}{b_n}$ where $b_n = b_{n-1} + b_{n-2}$, $b_0 = 1$, $b_1 = 1$.

Determine whether:

- (a) $(a_n)_{n \in \mathbb{N}}$ converges to limit 1.
- (b) $(a_n)_{n \in \mathbb{N}}$ converges to limit 0.
- (c) $(a_n)_{n \in \mathbb{N}}$ converges to limit -1.
- (d) $(a_n)_{n \in \mathbb{N}}$ diverges.
- (e) None of the above.

Answer: (a) The sequence $(a_n)_{n \in \mathbb{N}}$ converges to limit 1. As $n \in \mathbb{N}$ gets larger and larger, the Fibonacci term b_n gets larger and larger, and hence the sequence entry $a_n = \frac{b_{n-1}}{b_n}$ gets closer and closer to 1.

Exercise (ii): Consider the sequence

$(a_n)_{n \in \mathbb{N}} = (1, -3, 5, -7, 9, -11, \dots)$, i.e. $a_n = (-1)^n(2n+1)$ so $a_n = 2n+1$ if n is even, and $a_n = -(2n+1)$ if n is odd.

Determine whether:

- (a) $(a_n)_{n \in \mathbb{N}}$ converges to limit 1.
- (b) $(a_n)_{n \in \mathbb{N}}$ converges to limit 0.
- (c) $(a_n)_{n \in \mathbb{N}}$ converges to limit π .
- (d) $(a_n)_{n \in \mathbb{N}}$ diverges.
- (e) None of the above.

Answer: (d) The sequence $(a_n)_{n \in \mathbb{N}}$ diverges. For any $x \in \mathbb{R}$, if $\epsilon \leq 1$ between x and each of the even entries $a_n = (2n+1)$. odd entries $a_n = -(2n+1)$, while if $x < 0$ then there is a distance $x \leq 0$ then there is a distance of $\epsilon \leq 1$ between x and each of the odd entries $a_n = -(2n+1)$, while if $x > 0$ then there is a distance $x \geq 0$ between x and each of the even entries $a_n = (2n+1)$.

Exercise (iii): Consider the sequence

$$(a_n)_{n \in \mathbb{N}^+} = (1, 1 + \frac{1}{4}, 1 + \frac{1}{4} + \frac{1}{9}, 1 + \frac{1}{4} + \frac{1}{9} + \frac{1}{16}, \dots)$$

That is, $a_n = \sum_{k=1}^n \frac{1}{k^2} = 1 + \frac{1}{4} + \frac{1}{9} + \dots + \frac{1}{n^2}$ for $n \geq 1$, and the sequence begins with $(1, 1.25, 1.361\bar{1}, 1.42361\bar{1}, \dots)$.

Determine whether:

- (a) $(a_n)_{n \in \mathbb{N}^+}$ converges to limit 1.
 (b) $(a_n)_{n \in \mathbb{N}^+}$ converges to limit 0.
 (c) $(a_n)_{n \in \mathbb{N}^+}$ converges to limit $\frac{\pi^2}{6}$.
 (d) $(a_n)_{n \in \mathbb{N}^+}$ diverges.
 (e) None of the above.

(d) is wrong because the sequence does in fact converge.
 (b) are wrong because the limit is definitely not either 1 or 0, and
 This one is definitely harder. By process of elimination, (a) and

Answer: (c) The sequence $(a_n)_{n \in \mathbb{N}^+}$ converges to $\frac{\pi^2}{6}$.

5.5.4 | TRICKS WITH INFINITE SUMS

The sequence $(a_n)_{n \in \mathbb{N}^+}$ such that $a_n = \sum_{k=1}^n \frac{1}{k^2}$ is an instance of a convergent *infinite sum*:

$$\sum_{n=1}^{\infty} \frac{1}{n^2} = \lim_{n \rightarrow \infty} a_n = \lim_{n \rightarrow \infty} \left(\sum_{k=1}^n \frac{1}{k^2} \right) = \frac{\pi^2}{6}.$$

A rigorous proof would show that for any positive $\varepsilon > 0$, one can compute a cut-off index m_ε (getting larger with smaller ε) such that $D a_n \frac{\pi^2}{6} \varepsilon$ is true for all $n \geq m_\varepsilon$.

A nearby infinite sum is the *Harmonic series*:

$$\sum_{n=1}^{\infty} \frac{1}{n} \quad \text{with partial sums: } a_n = \sum_{k=1}^n \frac{1}{k}$$

The Harmonic series in fact *diverges*, but its rate of divergence is rather slow, as shown in this table.

c	1	2	3	4	5	6	7	8	9
n such that $a_n \geq c$	1	4	11	83	227	616	1674	4550	12367

The partial sums sequence $(a_n)_{n \in \mathbb{N}^+}$ gets larger and larger, and heads toward infinity, but at a snail's pace. For example, to find a term a_n such that $a_n \geq 5$, we have to go to $n = 227$, while to find a term a_n such that $a_n \geq 9$, we have to go to $n = 12367$.

Divergent infinite sums admit all sorts of trickery! In §5-1, we gave a pointer to a **You Tube** video:

<http://youtu.be/w-I6XTVZXww>

for the “astounding” false claim that the *infinite sum*:

$$\sum_{n=1}^{\infty} n = 1 + 2 + 3 + 4 + \dots$$

has the finite value of $-\frac{1}{12}$. This would mean that the sequence $(a_n)_{n \in \mathbb{N}^+}$ of n -th partial sums converges to the limit $-\frac{1}{12}$, where:

$$a_n = \sum_{k=1}^n k = 1 + 2 + 3 + \cdots + n.$$

The fallacy is that at all stages of the “proof” shown on the video, it is *assumed* that the infinite sum $1 + 2 + 3 + 4 + \cdots$ does in fact *converge*, and they are just using basic high school algebra to find its value. This is of course a false assumption!! Applying the predicate logic formalisation of divergence, we can easily prove that for every choice of real number $x \in \mathbb{R}$, we can take $\epsilon = 1$ and find *infinitely many* terms a_n in the sequence of n -th partial sums that are at least distance $\epsilon = 1$ away from the given x . (For $x \leq 0$, we can take *all* the terms a_n for $n \geq 1$, while for $x > 0$, choose the first n such that $a_n \geq (x + 1)$; then for all $m \geq n$, $a_m \geq (x + 1)$ and the distance between a_m and x is at least $\epsilon = 1$.) The conclusion that the “answer” is $-\frac{1}{12}$ is the product of the principle well-known to medieval logicians: *ex falso sequitur quodlibet*, which means “from a falsity, anything follows” !!

[**Aside:** The video uses only the false assumption of convergence together with elementary algebra to arrive at its conclusion. However, the movie was motivated by a topic in mathematics with application to quantum physics concerning *summation methods* for divergent infinite sums; in particular, a method known as *zeta function regularization* via analytic continuation of a related power series. In this case, the power series is the *Riemann zeta function* of a complex variable $s \in \mathbb{C}$, given by:

$$\zeta(s) = \sum_{n=1}^{\infty} \frac{1}{n^s}.$$

The function is only defined for complex numbers s whose real part is strictly greater than 1, but can be extended to complex numbers s whose real part is strictly greater than 0. *Naively* extending the Riemann zeta function to complex numbers with non-positive real part (and in particular, the real numbers $s = -1$ and $s = 0$), some people arrive at the “conclusion” that:

$$\sum_{n=1}^{\infty} n = \zeta(-1) = -\frac{1}{12}.$$

This *advanced trickery* is still just *trickery*, in that it starts with the premise that the sequence of n -th partial sums does in fact converge, and this premise is just plain false.

Do not confuse this trickery with more substantial work on related topics. The Riemann zeta function is the subject of a famous and important open problem in complex number theory, the *Riemann Hypothesis*. It was proposed by Bernhard Riemann (1859), and is the conjecture that the nontrivial zeros of the Riemann zeta function all have

real part $\frac{1}{2}$. It is one of the open *Millennium Prize* problems, worth one million US dollars.]

Running a different line of algebraic reasoning from a false assumption, we can conclude that the “answer” ... does not exist!. Assume (falsely) that there are real numbers $a \in \mathbb{R}$ and $b \in \mathbb{R}$ such that:

$$a = \sum_{n=1}^{\infty} n = 1+2+3+4+\dots \quad \text{and} \quad b = \sum_{n=1}^{\infty} 1 = 1+1+1+1+\dots$$

Then since every number in the series summing up to b is less than or equal to the corresponding number in the series summing up to a , since $1 \leq n$ for all positive natural numbers $n \in \mathbb{N}^+$, we must have $1 < b < a$. Now, separating the infinite sum $\sum_{n=1}^{\infty} n$ into two parts, the sum of all the odd numbers $\sum_{n=1}^{\infty} (2n - 1)$, and the sum of all the even numbers $\sum_{n=1}^{\infty} 2n$, we can conclude:

$$\begin{aligned} a &= \sum_{n=1}^{\infty} n \\ &= \left(\sum_{n=1}^{\infty} (2n - 1) \right) + \left(\sum_{n=1}^{\infty} 2n \right) \\ &= 2 \left(\sum_{n=1}^{\infty} n \right) - \left(\sum_{n=1}^{\infty} 1 \right) + 2 \left(\sum_{n=1}^{\infty} n \right) \\ &= 2a - b + 2a = 4a - b \end{aligned}$$

Hence $3a - b = 0$, and so $3a = b$ and thus $a = \frac{b}{3}$. But since $b > 1$, this means $a < b$, which contradicts our earlier assessment that $1 < b < a$. So these supposed real numbers a and b cannot exist! [The Zeta function trickery on which the You Tube video was based works with solutions a and b as *negative numbers* (so an infinite sum of all positive numbers comes out as *less than 0!!*), to get $b = \zeta(0) = -\frac{1}{2}$ and $a = \zeta(-1) = -\frac{1}{12}$.]

If we were happy to assume that the two infinite sums converge and that the real numbers a and b exist, then we could also divide the sum over all positive integers into *three* parts: the sum of all numbers $3n$ (multiples of 3), the sum of all numbers $(3n - 1)$, and the sum of all numbers $(3n - 2)$. In this case, the algebra yields the equation $a = 9a - 3b$, hence $a = \frac{3b}{8}$.

Dividing the sum over all positive integers into *four* parts, the algebra yields the equation $a = 16a - 6b$, hence $a = \frac{6b}{15}$ and so $a = \frac{2b}{5}$. And dividing the sum over all positive integers into *five* parts, the algebra yields the equation $a = 25a - 10b$, hence $a = \frac{10b}{24}$ and so $a = \frac{5b}{12}$. Clearly, we can keep on going in this way. None from this collection of algebraic constraints:

$$a = \frac{b}{3}, \quad a = \frac{3b}{8}, \quad a = \frac{2b}{5}, \quad a = \frac{5b}{12}, \quad \text{and so on...}$$

can be reconciled with the condition that $1 < b < a$. This means there are multiple routes to the conclusion that these supposed real numbers a and b cannot exist!

5.5.5 | APPLICATIONS OF SEQUENCE CONVERGENCE

The notion of sequence convergence is used widely within many areas of pure and applied mathematics, as well as in applications within science, engineering, medicine, economics, and other disciplines.

A sequence of real numbers $(a_n)_{n \in \mathbb{N}}$ can be used to model the state of a **dynamic process** or **system** that may change continuously in real time but is only measured at **discrete points in time** (e.g. temperature in engine measured once every 10 milliseconds), with a_n the measured value at the **n -th time point**. The convergence or divergence behaviour of such a sequence $(a_n)_{n \in \mathbb{N}}$ reflects the **asymptotic behaviour** of the system as the time points go to infinity, with convergence indicating that, “in the long run”, the system heads toward an equilibrium state, while divergence indicates that “in the long run” the system “blows up”.

We will finish off this section with brief discussion of the *computer representation of real numbers*, and the application of sequence convergence in this context.

First note the massive discrepancy: a computer can in fact only store and represent *finitely* many numbers, because numbers are represented as *finite-length* sequences of 0s and 1s (in base 2), while the set of real numbers, or of any non-trivial interval of real numbers, is *uncountably infinite*. This means that it is of a higher degree of infinity than the set of natural numbers or the rationals, both of which are *countably* infinite.

In particular, the cardinality of the set (interval) of numbers:

$$[1, 2) = \{x \in \mathbb{R} \mid x \geq 1 \text{ \& } x < 2\}$$

is **uncountably infinite**. The finitely many different numbers represented by the *mantissa* part of computer reals in base 2 (so called *floating point numbers*) all lie in the interval $[1, 2)$.

Computer representations of real numbers use what is known as a **floating point representation** which is **scientific notation** in **base 2**:

$$\text{number} = (\text{sign value}) \times \text{mantissa} \times 2^{\text{exponent}}$$

where **sign value** $\in \{1, -1\}$, and
mantissa $= 1.b_1b_2 \dots b_m$ with binary digits (bits) $b_i \in \{0, 1\}$.

The standards for representing 32-bit (“single precision”) and 64-bit (“double precision”) numbers are as follows:

	bits in sign	bits in exponent	bits in mantissa	total bits
32-bit	1	8	23	32
64-bit	1	11	52	64

The *mantissa* part $1.b_1b_2 \dots b_m$ is between 1 and 2 in base 2 scientific notation, for the same reason that the mantissa part in base 10 decimal scientific notation is between 1 and 10.

With a 32-bit representation, there are $m = 23$ bits devoted to the mantissa, so there are 2^{23} (which is greater than 8.3×10^6) 32-bit representable numbers within the interval $[1, 2]$.

With a 64-bit representation, there are $m = 52$ bits devoted to the mantissa, so there are 2^{52} (which is greater than 4.5×10^{15}) 64-bit representable numbers within the interval $[1, 2]$.

In each case, the set of real numbers so represented is *discretely ordered* by $<$. There is a distance gap of $\varepsilon = \frac{1}{2^{23}}$ or $\varepsilon = \frac{1}{2^{52}}$ between successive *representable numbers*, with 32-bit and 64-bit representations, respectively.

Consider a 32-bit floating point representation, for which the negative number b_m is the smallest real number representable and the positive number b_M is the largest real number representable.

For any real number $c \in [b_m, b_M]$, let $dr_{32}(c)$ denote 32-bit digital representation of the number c . A real number $c \in \mathbb{R}$ is *exactly representable* by this scheme iff $c = dr_{32}(c)$.

For all real numbers $c \in [b_m, b_M]$, regardless of whether or not it is exactly representable, we will have that $D c dr_{32}(c) \varepsilon$ is true for all $\varepsilon \geq \frac{1}{2^{23}}$, because this is the distance between one 32-bit representation and the *next largest* 32-bit representation.

Suppose $(a_n)_{n \in \mathbb{N}}$ is a sequence of real numbers converging to a real number a and the entries a_n all lie in $[b_m, b_M]$, and are the n -th approximation to the exact value a . For example, the Archimedean approximation sequence for π based on inscribed polygons. Then the sequence $(dr_{32}(a_n))_{n \in \mathbb{N}}$ arising from the digital representation must eventually *become constant*: there must exist an $m \in \mathbb{N}$ such that for all $n \geq m$, we have $dr_{32}(a_n) = dr_{32}(a)$; from convergence, choose the cut-off $m \in \mathbb{N}$ such that $D a_n a \frac{1}{2^{23}}$ for all $n \geq m$.

If the real number infinite sequence $(a_n)_{n \in \mathbb{N}}$ converges then the 32-bit digital representation sequence $(dr_{32}(a_n))_{n \in \mathbb{N}}$ is effectively a *finite sequence* rather than an infinite one, because $(dr_{32}(a_n))_{n \in \mathbb{N}}$ becomes constant after – and so can be stopped at – the convergence cut-off stage m such that $D a_n a \frac{1}{2^{23}}$ for all $n \geq m$.

This brings us to the end of this chapter on the importance of quantifiers in clarifying and explicating complex mathematical concepts.

We hope you have enjoyed this foray into mathematics via logic, and get stuck into more examples within the online practice and graded quizzes.

FURTHER LEARNING

- Coursera subject **Introduction to Mathematical Thinking**, taught by Prof. Keith Devlin of Stanford University:
<https://www.coursera.org/course/maththink>
- W. Rautenberg. *A Concise Introduction to Mathematical Logic* (Third Edition), Springer (series: Universitext), New York, 2010.
- E. Mendelson. *Introduction to Mathematical Logic* (Fifth Edition), Chapman and Hall/CRC, 2010.

PREDICATE LOGIC PROGRAMMING

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 predicate logic: program clauses, logic programs and goals; Horn clause fragment of predicate logic.
- Basic PROLOG notation:
 - symbol “:-” read as “if” in program clauses, and comma as “and” in program clauses and goals;
 - identifiers for Variables start with Capital letter; names and predicates start with a lower case letter;
 - “\=” for “cannot be unified”, in the role of \neq ;
 - queries entered at ?- prompt; and
 - semi-colon ; as means to ask for more answers instantiating variables.
- How PROLOG answers queries: the resolution proof rule, unification substitution, and resolution refutation sequences.
- Incompleteness in PROLOG: when it can crash and not answer.

SKILLS

These are the skills you will learn to demonstrate.

- Determine whether or not a logic formula is logically equivalent to a logic program, or is logically equivalent to a goal.
- Represent a small knowledge base of information as a logic program, write program clauses to structure that information, and write queries to answer questions based on that information.
- Identify a resolution refutation sequence that shows why PROLOG answers the way it does to a given query of a logic program.
- Given a PROLOG program and a query, determine how PROLOG will respond to that query.

This chapter is in the application area of Computer Science. It gives an introduction to *automated reasoning* – the task of programming a computer to compute logical consequence – which is located within the field of artificial intelligence. We give brief survey of the limits of automated reasoning in predicate logic, due to the *algorithmic undecidability* of the logical consequence problem for predicate logic with predicates of arity 2 or greater.

We then introduce the framework of logic programming, which is a programming paradigm based on formal logic. Programs written in a logic programming language are sets of logic formulas, expressing facts and conditional rules about some problem domain; here, we use the language called PROLOG. To expand your knowledge base, you pose a query to the PROLOG system; this is the means to ask whether or not a formula is a logical consequence of the given logic program. In comparison with propositional logic, it takes rather more work in predicate logic to precisely demarcate the fragment of logic formulas that can be handled by PROLOG, either as program clauses or as goals of queries.

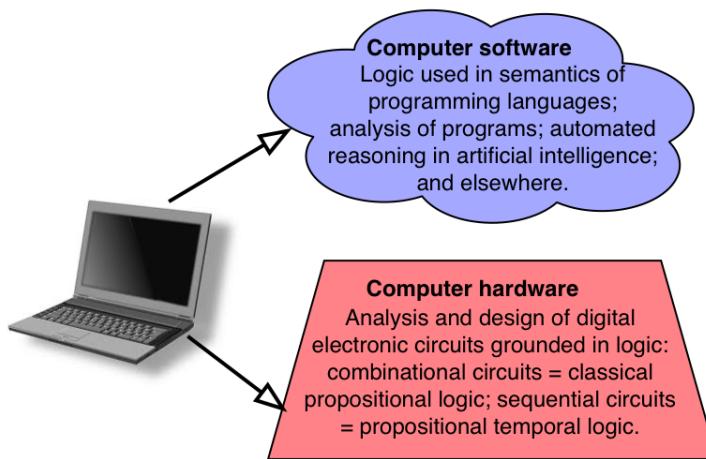
We work through several example PROLOG programs and queries, using as our sample knowledge base some family tree data from the internationally popular TV and book series, *Game of Thrones*; using that database as a testing ground, we define increasingly complex family relationship predicates within PROLOG.

We will then examine in detail the inference process used by PROLOG in answering queries, in order to understand how and why PROLOG responds to queries the way it does. We will see that the inference procedure is sound but not complete - we will see simple logic programs and queries which should be answered YES, because the query formula is a logical consequence of the logic program, but PROLOG gets stuck in a loop with a stack overflow and fails to answer.

6.1 | PREDICATE LOGIC, COMPUTERS, & AUTOMATED REASONING

6.1.1 | LOGIC AND COMPUTERS

First, a big picture view. Computers have profoundly changed the world over the last fifty 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. Add to those gates some basic memory components that allow the storage of 1 bit of information for 1 tick of a digital clock, and one can generate the class of all digital circuits. The Applications Section §9 on **Sequential Digital Circuits** and Temporal Logic gives an introduction to 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.

6.1.2 | AUTOMATED REASONING

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 consequences*.

Given a *knowledge base* represented by a list of predicate logic formulas:

$$A_1, A_2, \dots, A_n$$

we want to know whether or not we can *extend* our knowledge with the addition of further formula B by determining whether or not:

$$A_1, A_2, \dots, A_n \models B$$

The *knowledge* available to the system should not be restricted to just the information content explicitly stored in the knowledge base, but

should ideally include all the logical consequences of the knowledge base. Recall that this requires the *truth-transferring* property that for every model M in which all the formulas in the knowledge base A_1, A_2, \dots, A_n , are true, the formula B is also true in M .

A *decision procedure* for the logical consequence relation of predicate logic is an *algorithmic procedure* (implementable in a computer program) that would give a **YES** or **NO** answer to the question whether or not:

$$A_1, A_2, \dots, A_n \models B ?$$

for any input predicate logic formulas A_1, A_2, \dots, A_n and B .

In 1928, the German logician and mathematician **David Hilbert** [1862-1943] posed an equivalent problem for predicate logic tautology/validity: given a formula A of predicate logic, determine whether or not A is true in all models. This is known as *Hilbert's Entscheidungsproblem*. Note that the logical consequence decision problem as stated here is equivalent to the predicate tautology problem, and also to the satisfiability problem of truth in one model. These are equivalent problems in the sense that a decision procedure for one would give a decision procedure for the other two, since:

$$\begin{aligned} & A_1, A_2, \dots, A_n \models B \\ \text{iff } & (A_1 \& A_2 \& \dots \& A_n) \supset B \text{ is a tautology} \\ \text{iff } & (A_1 \& A_2 \& \dots \& A_n) \& \sim B \text{ is not satisfiable.} \end{aligned}$$

It is important to note that this was 1928, *before* the development of computers, so the notion of an *algorithmic procedure* was an intuitive, informal notion of a step-by-step process leading to an answer, in a finite number of steps.

Hilbert was an immensely important mathematician who posed a number of famous mathematical problems. He put forth a big batch of 23 problems in 1900 at the International Congress of Mathematicians in Paris in 1900, but he kept adding to the list.

"This conviction of the solvability of every mathematical problem is a powerful incentive to the worker. We hear within us the perpetual call: There is the problem. Seek its solution. You can find it by pure reason, for in mathematics there is no ignorabimus." D. Hilbert, "Mathematical Problems", *Bulletin of the American Mathematical Society*, vol. 8, no. 10 (1902), pp. 437-479. Earlier publications (in the original German) appeared in *Göttinger Nachrichten*, 1900, pp. 253-297, and *Archiv der Mathematik und Physik*, 3dser., vol. 1 (1901), pp. 44-63, 213-237.

In fact, we all remain *ignorabimus* in mathematics: some of Hilbert's problems remain today open and unanswered.

However Hilbert's *Entscheidungsproblem* is not one of them. By late 1930, it was known that such a procedure would be impossible, with definitive proof coming in 1936.

6.1.3 | IMPOSSIBILITY OF COMPUTING LOGICAL CONSEQUENCE

Hilbert's broader program of formalising mathematics in logic was brought to a crashing halt by a then young Austrian logician **Kurt Gödel** in late 1930, with his *Incompleteness Theorems*, published in 1931.

Just prior to that, Gödel first made a significant *positive* contribution to Hilbert's *Entscheidungsproblem*: his doctoral dissertation in 1929 (published in 1930) gives the first proof of the *Completeness* of the axiomatic proof system for predicate logic. As we know from the completeness of the proof tree method for predicate logic (in §3.6), this means the semantic problem of logical consequence can be transcribed into a syntactic problem of following a set of rules to construct a proof.



Kurt Gödel
[1906–1978]

[on left, with
Einstein, 1950.]

Gödel then specialised to the predicate logic of arithmetic on the natural numbers in an effort to advance Hilbert's program. He was attempting to prove the consistency of analysis (or, second-order arithmetic) with the resources of first-order arithmetic, and thus reduce the consistency of the former to the consistency of the latter. In his attempted proof, he needed to formalise the notion of truth. Gödel soon faced various paradoxes of self-reference (such as the Liar paradox), and had to conclude that arithmetical truth cannot be defined in arithmetic.

Gödel's *Incompleteness Theorems* (1931) can be summarised as follows: 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.

“Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I”, “On formally undecidable propositions of Principia Mathematica and related systems I”, *Monatshefte für Mathematik Physik*, 38 (1931).

For a non-technical discussion of Gödel's theorem, see the blog post by Mark Dominus under the title *World's Shortest Explanation of Gödel's Theorem* at

<http://blog.plover.com/math/Gdl-Smullyan.html> and

<http://blog.plover.com/math/Gdl.html>

These versions are inspired by the exposition by Raymond Smullyan in *5000 B.C. and Other Philosophical Fantasies* (St Martins Press, 1984).

Gödel's incompleteness theorems did not directly prove that Hilbert's *Entscheidungsproblem* could not be solved, but it was quickly

realised that Gödel's arithmetical coding of self-reference could be done in predicate logic using predicates of arity 2 or higher.

In addition to Gödel-style self-reference, the extra ingredient needed to prove the *Entscheidungsproblem* could not be solved was a formalisation of the as yet intuitive notion of what is *algorithmically computable*.



Alonzo Church
[1903-1995]

The American logician and mathematician **Alonzo Church** formalised the concept of *computable algorithm* using his *Lambda Calculus* model – which remains very much alive today as the basis of functional programming languages.

Church proved in 1935 (published 1936) the *algorithmic undecidability* of Hilbert's *Entscheidungsproblem*: there can be no algorithm to give YES/NO answers to all possible predicate logic tautology questions, or logical consequence questions.

Alonzo Church, along with Kurt Gödel, Alan Turing and Alfred Tarski, were at the forefront of 20th century logic and in particular, the great leaps forward in the 1930s. Alonzo Church founded the *Journal of Symbolic Logic* in 1936, and remained on its editorial committee until 1979.



Alan Turing
[1912-1954]

The English logician and mathematician **Alan Turing** formalised the concept of *computable algorithm* with his *Logical Computing Machine* model, subsequently called *Turing machine* model. Just months after Alonzo Church's paper came Turing's 1936 paper "On Computable Numbers" which also used Gödel-style self-reference 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.

Turing also proved that the lambda-calculus and Turing machines are equivalent models of computation, and they are also equivalent to a third class of functions whose values could be calculated by recursion, identified by Kurt Gödel and Jacques Herbrand. The Church-Turing thesis is that this one class of computable functions (with three equivalent characterisations) correctly captures the intuitive sense of what is *algorithmically computable*. Starting with the Gödel-Herbrand class of primitive recursive functions, there is now a whole branch of mathematical logic called *recursion theory* devoted to studying degrees of computability and undecidability.

For a simple to describe undecidable fragment, let \mathcal{F}_1 be the set of

all predicate logic formulas of the form:

$$(\forall x)(\exists y)(\forall z)A$$

where the body A is a propositional combination built from one binary predicate and any number of monadic predicates, but no identity predicate. Then the logical consequence and tautology problems for formulas in \mathcal{F}_1 is algorithmically undecidable. This result is due to Kahr, Moore and Hao Wang, “Entscheidungsproblem reduced to the $\forall\exists\forall$ case”, *Proc. of the National Academy of Sciences* 48 (1962), 365–377.

6.1.4 | PRACTICALLY COMPUTING LOGICAL CONSEQUENCE

It is important to note is that there are *fragments* of predicate logic with a decidable logical consequence problem, but often the *computational complexity* is very high. The *classical solvable fragments* of predicate logic are characterised by quantifier alternation patterns:

- $\exists^*\forall^*$: Bernays, Schönfinkel (1928); Ramsey (1932);
- $\exists^*\forall\exists^*$: Ackermann (1928);
- $\exists^*\forall\forall\exists^*$ without identity: Gödel (1932); Kalmár (1933); Schütte (1934).

Here Q^* (for $Q = \forall$ or \exists) means 1 or more Q quantifiers, and the body of the formulas are propositional combinations built from predicates of any arity.

The *Monadic Predicate Logic (MPL)* fragment of predicate logic has a long history and a decidable logical consequence problem whose complexity is “not too bad”. MPL is restricted to 1-place predicates only. Without binary or higher arity predicates, there is no complex nesting of quantifiers in MPL. It includes the syllogisms of Aristotelian and medieval logic (also known as *Term Logic*). There is an online *Logic calculator* for Monadic Predicate Logic to experiment with at: <http://somerby.net/mack/logic/>

The *finite model property* result for MPL says: A formula of MPL is satisfiable if and only if it is satisfiable in a model of size 2^k , where k is the number of monadic predicates in the formula (Löwenheim: 1915)

The *decidability* of the logical consequence and predicate tautology problems for MPL were established by Löwenheim (1915), Behmann (1922) and Kalmár (1929).

The computational complexity of the decision problems for MPL was established in 1980 by Harry Lewis with the result that it lies in the complexity class $\text{NTIME}(c^{n/\log(n)})$, where n measures size of input formulas and $c \geq 2$ some positive constant. Note that the complexity function $c^{n/\log(n)}$ is much smaller than a full exponential c^n , so in this sense, the complexity of MPL is in the “not too bad” exponential class. It ends up in the *non-deterministic* complexity class because the algorithm involves non-deterministically generating a bunch of candidate models, and then *testing* whether or not they are a counter-model

for the logical consequence, where the testing part is the relatively fast and easy bit.

These technical details are given to make the comparison with the complexity of the logical consequence problem for *propositional logic*, which falls in the complexity class called $\text{NTIME}(p(n))$ where $p(n)$ is a *polynomial* in size n (that is, $p(n) = cn^k$ for some power k and constant c). Again, n measures the size of the input formulas. The result was proved by Cook in 1971 – with the stronger conclusion that the decision problems for propositional logic are *NP-complete*. So the “not too bad” exponential $c^{n/\log(n)}$ for MPL is definitely larger than any polynomial $p(n)$, but not a whole lot larger. The end result: computing logical consequence for MPL *harder* than computing logical consequence for propositional logic, but not a whole lot harder.

Returning to full predicate logic with identity, here is an exercise for you, to keep you focused on the enterprise of computing logical consequence.

If A_1, A_2, \dots, A_n, B are all predicate logic formulas, which of the following are true if and only if $A_1, A_2, \dots, A_n \models B$:

- (a) The proof tree starting with A_1, A_2, \dots, A_n and $\sim B$ has all its branches closed.
- (b) The proof tree starting with A_1, A_2, \dots, A_n and B has all its branches closed.
- (c) There does not exist a model M such that A_1, A_2, \dots, A_n are all true in M , and B is false in M .
- (d) For every model M , if A_1, A_2, \dots, A_n are all true in M , then B is true in M .

- | | | |
|--|---|---|
| (a) YES | The proof tree starting with A_1, A_2, \dots, A_n and $\sim B$ has all its branches closed. | true in M , then B is true in M . |
| (b) NO | The proof tree starting with A_1, A_2, \dots, A_n and B has all its branches closed. | For every model M , if A_1, A_2, \dots, A_n are all true in M , and B is false in M . |
| (c) YES | There does not exist a model M such that A_1, A_2, \dots, A_n are all true in M , and B is false in M . | For every model M , if A_1, A_2, \dots, A_n are all true in M , then B is true in M . |
| (d) YES | For every model M , if A_1, A_2, \dots, A_n are all true in M , then B is true in M . | There does not exist a model M such that A_1, A_2, \dots, A_n are all true in M , and B is false in M . |

As we have been reminded by the exercise, for the logic as studied in the core part of this course, *predicate logic with identity*, we have soundness and completeness of the proof tree method:

$$A_1, A_2, \dots, A_n \models B \text{ if and only if } \vdash_{\text{PfTr}} ((A_1 \& A_2 \& \dots \& A_n) \supset B)$$

Of course, there is the overarching problem of *algorithmic undecidability*. We can see this directly in the form of proof trees with infinitely long branches. If we coded up proof tree rules into a computer program, we would not get any answer in any finite amount of time when the input formulas generated an infinite tree. For example, consider a proof tree for:

$$(\forall x)(\exists y)Rxy \models (\exists y)(\forall x)Rxy.$$

Since the tree begins with $(\forall x)(\exists y)Rxy$ and $\sim(\exists y)(\forall x)Rxy$, the *general* quantifier rules for \forall and $\sim\exists$ will apply infinitely often. Every time the *particular* quantifier rules for \exists and $\sim\forall$ are used developing the inner scope quantifiers, they create a new name; then that new name has to be instantiated by outer scope general quantifiers, and this repeats *ad infinitum*. The answer to this logical consequence in NO, and a counter-model can be extracted from the infinite branch. But if the tree is being developed step-wise by a computer, there will be no answer in finite time.

What we do have for the proof tree algorithm is *soundness* or correctness: if the tree method gives an answer, either YES or NO, then the answer is correct. And as humans, we can analyse the patterns in an infinite branch and work out what the associated model is going to look like. But this task is much harder for a computer program to handle, or be coded to do. A further difficulty with the proof tree method is having lots of choice in which rule to apply next; this is nice for humans but not good for computer programs. There are software implementations out there for testing logical consequence in predicate logic that are based on the proof tree method, although a more streamlined variant of proof trees called *tableaux* is typically used instead.

For the remainder of this chapter, we will be focused on a *useful* fragment of predicate logic that has a *practically computable* logical consequence problem, that will give an answer “most of the time”. The strategy is to:

- First, restrict to a simple fragment of the language. We will be working with formulas called *program clauses* and *goals*, that are associated with the *Horn clause* fragment of predicate logic.
- Second, use a proof system with *only one rule*, to keep it simple for dumb machines! Working as we are with predicate logic, this one rule, called *resolution* will have some bells and whistles attached. In particular, we need to be systematic about taking *instances* of quantified variables, using a method called *unification*.

We will be using a logic-based programming language called PROLOG, which falls within the *Logic Programming* paradigm; another such programming language is called DATALOG. Logic Programming is a direct, declarative style of computer programming using logic formulas. 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 pro-

gramming languages such as Java or C, where the program describes explicitly the steps of computation.

The predicate fragment logic used in PROLOG will still have an algorithmically undecidable logical consequence problem, but the decision procedure will be “good enough” for practical purposes.

6.2 | FRAGMENT OF PREDICATE LOGIC IN PROLOG

Our first task is to identify which formulas of *predicate logic with identity* can be handled by *basic* predicate PROLOG, so we get a clear sense of which logical consequence questions we can answer using PROLOG.

In this short introduction, we will only see *basic* predicate PROLOG; we are not able to deal with any advanced features of the language and its implementations. In particular:

- We will not use the *not-provable* operator $\text{\textbackslash}+$ [See §6-4 Negation in PROLOG from LLI 1 for a discussion.]
- But we will allow *difference assertions* $a \neq b$ as atoms.
- We will not use any of the built-in arithmetic or list manipulation functions available in PROLOG.

6.2.1 | REVISITING FORMULAS OF PREDICATE LOGIC

Recall our definition of formulas of predicate logic with identity:

- If F is a **PREDICATE_n** (of arity n) and a_1, \dots, a_n are **NAMES**, then $Fa_1 \dots a_n$ is a **FORMULA**.
- If a_1 and a_2 are **NAMES**, then $(a_1 = a_2)$ is a **FORMULA**.
- The propositional connectives \sim , $\&$, \vee , \supset , \equiv build up **FORMULAS**.
- If A is a **FORMULA**, a is a **NAME**, and x is a **VARIABLE**, then $(\forall x)A[a := x]$ and $(\exists x)A[a := x]$ are **FORMULAS**.

In our original definition of formulas, variables only appear when we want to *quantify*, and then we do a substitution replacing all occurrences of a name a with a variable x – in order to have a variable available to bind to the quantifier.

In order to describe the (program clauses + goals) fragment used in PROLOG and the restricted pattern of propositional connectives and quantifiers that will be allowed, we will slightly revise the way formulas are formed.

To begin with, we introduce two new syntactic categories, **TERMS** and ***-ATOMS**.

- If a is a **NAME** then a is a **TERM**.
- If x is a **VARIABLE** then x is a **TERM**.
- If F is a **PREDICATE_n** (of arity $n \geq 1$) and t_1, \dots, t_n are **TERMS**, then $Ft_1 \dots t_n$ is a ***-ATOM**.
- If t_1 and t_2 are **TERMS**, then $(t_1 = t_2)$ is a ***-ATOM** and $(t_1 \neq t_2)$ is a ***-ATOM**.
- If P is a **PROPOSITIONAL ATOM** (a **PREDICATE₀**) then P is a ***-ATOM**.
- Nothing else is a ***-ATOM**.

The new category of ***-ATOMS** includes all *real* atomic formulas, but also allows the occurrence of variables as well as names, without a quantifier in sight. Because of the possible presence of free, unquantified variables, ***-ATOMS** in general are not real formulas, and do not express propositions that are either true or false in any model interpreting their predicate and any names in them. We will end up with real formulas that are true or false in a model, but we take a route through ***-ATOMS** to get there.

Also notice that we include negated identities/difference assertions as ***-ATOMS**. This is a departure from the presentation of the core predicate logic in §1 and §4, but not a substantive one. As we shall see, this inclusion of difference assertions will be the only way we include negation directly in Prolog formulas.

If A is a ***-ATOM**, then:

$$\text{var}(A) = \{x \mid x \text{ is a } \text{VARIABLE} \text{ occurring in } A\}.$$

$\text{var}(A)$ is a *set* or list of variables, with no duplicates.

It follows that A is a *real* atomic **FORMULA** if and only if $\text{var}(A)$ is empty. Such formulas A are also known as *ground atoms*.

For example, if P is a 5-place predicate, then $Pxyz$ is a ***-ATOM** and $\text{var}(Pxyz) = \{x, y, z\}$, where a is a name.

With the same 5-place predicate P , if a, b and c are names, then $Pabcz$ is a ***-ATOM** and $\text{var}(Pabcz) = \{z\}$.

For our present purpose, we only need to consider *propositional* combinations of \star -ATOMS.

- If A is a \star -ATOM, then A is a \star -FORMULA.
- If A and B are \star -FORMULAS, then $\sim A$, $(A \ \& \ B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$ are \star -FORMULAS.

6.2.2 | PROGRAM CLAUSES AND THEIR QUANTIFICATION

With this preamble, we can now define a *program clause*. Program clauses are also called *definite clauses* or *positive clauses*, and are \star -FORMULAS of one of two very simple kinds.

A program clause is a \star -FORMULA of one of two kinds:

$$(A_1 \ \& \ A_2 \ \& \ \dots \ \& \ A_m) \supset B \quad \text{conditional rule}$$

$$B \quad \text{fact}$$

where $m \geq 0$ and A_1, A_2, \dots, A_m, B are all \star -ATOMS, and B is not a difference \star -ATOM ($t_1 \neq t_2$).

When convenient, we consider all program clauses as conditionals, with facts falling into the special case when $m = 0$. In this special case, the antecedent is a size-0 conjunction that behaves like the *always true* logical constant \top , and the conditional is $(\top \supset B)$, which is logically equivalent to just B itself.

For technical reasons, we need to *exclude* the case where the consequent or *head* B of a program clause is a difference \star -ATOM ($t_1 \neq t_2$), because we cannot deal with a negatively valenced consequent. So difference \star -ATOMS ($t_1 \neq t_2$) can only occur in the antecedent or *body* of a program clause.

When programming in PROLOG, the head B of a program clause is a predicate \star -ATOM where the purpose of the program clause is to *define*, or *characterise*, or asserts a *fact* about, the predicate in question.

We need to refer to the variables within a program clause C . If C is the program clause $(A_1 \ \& \ A_2 \ \& \ \dots \ \& \ A_m) \supset B$ for some $m \geq 0$, then:

$$\text{var}(C) = \text{var}(A_1) \cup \text{var}(A_2) \cup \dots \cup \text{var}(A_m) \cup \text{var}(B)$$

where \cup is the union or merge operation, with no duplicates.

The variables within program clauses will always be understood as *universally quantified*.

If C is the program clause $(A_1 \& A_2 \& \dots \& A_m) \supset B$, for $m \geq 0$, then we write $(\forall \text{var}(C))C$ to mean the **FORMULA** (a real one!) resulting from universally quantifying all of the variables occurring in the program clause C .

For example, if C is the program clause:

$$(Rxy \& Syz \& Pza) \supset Sxa$$

then $\text{var}(C) = \{x, y, z\}$ and $(\forall \text{var}(C))C$ is the **FORMULA**:

$$(\forall x)(\forall y)(\forall z)((Rxy \& Syz \& Pza) \supset Sxa).$$

For any \star -**FORMULA** C , if $\text{var}(C)$ is empty, so C is a *real FORMULA*, (also called a **GROUND FORMULA**) containing only **NAMES**, **PREDICATES** and **PROPOSITIONAL ATOMS**, then:

$(\forall \text{var}(C))C$ is just C , and $(\exists \text{var}(C))C$ is just C .

If there are no variables to quantify, then quantification will add nothing to the formula.

6.2.3 | AUTOMATED REASONING USING PROLOG

With all this lead up, we can now clearly state the class of logical consequence problems addressable in basic PROLOG.

A **logic program** \mathcal{P} is a list C_1, C_2, \dots, C_n of program clauses, whose variables are understood as *universally quantified*.

A knowledge base of information in some subject domain is represented by a list \mathcal{P} of program clauses, called a *logic program*. This is typically a mix of facts and conditional rules, with and without variables, but whatever variables there are will be *universally quantified*.

A **goal** G is a list G_1, G_2, \dots, G_k of \star -**ATOMS**, and the corresponding **goal** \star -**FORMULA** is the conjunction $D : (G_1 \& G_2 \& \dots \& G_k)$, whose variables are understood as *existentially quantified*, with

$$\text{var}(D) = \text{var}(G_1) \cup \text{var}(G_2) \cup \dots \cup \text{var}(G_k).$$

A *goal* is a list of \star -**ATOMS**, referred to as *sub-goals*. The variables in a goal are understood as *existentially quantified*; we want to know if *there exists* a way to simultaneously satisfy each of the sub-goals.

Automated reasoning task addressed by PROLOG:
determine whether or not $\mathcal{P} \models \mathcal{G}$; that is,

$$(\forall \text{var}(C_1))C_1, (\forall \text{var}(C_2))C_2, \dots, (\forall \text{var}(C_n))C_n \models (\exists \text{var}(D))D$$

Expressed as *real* formulas of predicate logic with identity, the automated reasoning task here is to determine whether or not the existential quantification of the goal conjunction D is a logical consequence of the universally quantified program clauses C_i in \mathcal{P} .

Note the quite restricted propositional combinations of \star -ATOMS allowed in program clauses and goals; there are simple conjunctions and conditionals, but negation is nowhere to be seen. The only hint of negation is in the form of the *difference* \star -formulas, ($t_1 \neq t_2$), which get to be classified as \star -ATOMS precisely so we can sneak in this much negation. (But remember the exclusion: the head of a program clause cannot be a difference \star -formula.) Because we are working with predicates, names and variables, but not with function symbols, identity and difference are not *too* complicated in PROLOG.

We will be spending the rest of this section coming to a better understanding of the class of *real* predicate logic formulas corresponding to program clauses and goals. In the next section, §6-3, we get down to business with PROLOG programming. But to keep up interest in case it is flagging, here is a sneak preview of what a PROLOG interpreter does.

- A PROLOG interpreter answers YES to a logical consequence question by providing an *existential witness* in the form of a **NAME** for each of the variables in **var(D)**. PROLOG implementations give a means to ask if there is *more than one* set of existential witnesses for the variables in the goal, using the semi-colon key “;” on the keyboard.
- A PROLOG interpreter answers NO to a logical consequence question with the response **false**; it also responds this way to the semi-colon key “;” when all the existential witnesses have been exhausted.
- PROLOG interpreters can *get stuck in a loop and crash without answer*. It is very easy to create examples of this phenomenon; we will see some in §6-4.
- When the goal formula D is a *real* FORMULA (**var(D)** is empty, so only NAMES, PREDICATES and PROPOSITIONAL ATOMS in D), then $(\exists \text{var}(D))D$ is just D, and PROLOG will directly answer with either **true** or **false** or ... *get stuck in a loop and crash without answer!*
- It is good programming practice to ensure that variables occurring in the goal are all *different* from the variables used in the logic program. This is required to ensure the correct behaviour of the *unification* algorithm used by PROLOG interpreters when taking substitution instances of program clauses to match with goals. In

terms of logical equivalence of formulas, the choice of a bound variable name does *not* matter for the meaning of the formula, or for its logical consequence status, but it does for computing.

6.2.4 | FORMULAS EQUIVALENT TO PROGRAM CLAUSES

We start by getting a better understanding of formulas that are logically equivalent to a program clause.

A formula A of predicate logic with identity is *logically equivalent to a program clause* if and only if there exists a program clause C such that:

$$A \equiv (\forall \text{var}(C))C$$

is a tautology.

A formula A of predicate logic with identity is *logically equivalent to a logic program* if and only if there exists a logic program $\mathcal{P} = (C_1, C_2, \dots, C_n)$ for $n \geq 1$ such that:

$$A \equiv ((\forall \text{var}(C_1))C_1 \ \& \ (\forall \text{var}(C_2))C_2 \ \& \ \dots \ \& \ (\forall \text{var}(C_n))C_n)$$

is a tautology.

In predicate logic, it takes a bit more work than in propositional logic to show that a formula is logically equivalent to a program clause. In the propositional case, there are some more detailed examples of reasoning through a sequence of logical equivalences in the **LL1 Course Notes**, in §1-3-3, at the end of Chapter 1. As was the case for propositional logic, it is useful to have handy a list of known tautology biconditionals to call on when reasoning through a sequence of logical equivalences. In these **LL2 Course Notes**, such a list can be found at the end of Chapter 2.

One of the more useful logical equivalences we use here is called a *Prenexing law* for the \exists -quantifier in the antecedent of a conditional:

$$((\exists x)A \supset B) \equiv (\forall x)(A \supset B)$$

is a tautology provided that the variable x is not *free* in B (that is, either x is not in B or else x is bound by a quantifiers in B, and in the latter case, it should be renamed with a different variable).

The right-hand-side of the equivalence is a formula exactly in the shape we are interested in for program clauses: a universal quantification of a conditional. The left-hand-side is a conditional with an existential quantifier in the antecedent. The equivalence can be read as

saying we can *pull* the quantifier from inner scope to outer scope, but in the process it swaps from being existential to becoming universal. This can be done *provided* there is no unintended binding of variables to quantifiers by requiring that the quantified variable x does not occur *free* in the consequent of the conditional, meaning either no x at all, or quantified x . We will in fact only see cases where there is no occurrence at all of the variable in question.

You might like to test out your proof tree skills to confirm that this bi-conditional is indeed a predicate tautology. The *dual* Prenexing law for the \forall -quantifier in the antecedent of a conditional, swaps the \exists and \forall quantifiers. There are Prenexing laws for the other propositional connectives as well, allowing a quantifier to be *pulled* from inner to outer scope, but it is only in the antecedent of a conditional that Prenexing involves swapping the flavour of the quantifier between existential and universal.

When working with the conjunction of two or more program clauses, to show a formula is logically equivalent to a logic program, one will often appeal to the logical equivalence expressing that the \forall -quantifier distributes over conjunction $\&$.

Let's see a detailed example of reasoning through a sequence of logical equivalences. We will use the following instance of the Prenexing law for exists in the antecedent of a conditional:

$$((\exists z)(A_1 \& A_2 \& A_3) \supset B) \equiv (\forall z)((A_1 \& A_2 \& A_3) \supset B)$$

provided the variable z is not *free* in B .

We start with formula $(\forall x)(Sxa \vee (\forall y)(\forall z)(\sim Rxy \vee \sim Syz \vee \sim Pza))$, and want to show that it is logically equivalent to a program clause.

$$\begin{aligned} & (\forall x)(Sxa \vee (\forall y)(\forall z)(\sim Rxy \vee \sim Syz \vee \sim Pza)) & (1) \\ \equiv & (\forall x)(Sxa \vee (\forall y)(\forall z)\sim(Rxy \& Syz \& Pza)) & (2) \\ \equiv & (\forall x)(Sxa \vee \sim(\exists y)(\exists z)(Rxy \& Syz \& Pza)) & (3) \\ \equiv & (\forall x)(\exists y)(\exists z)(Rxy \& Syz \& Pza) \supset Sxa & (4) \\ \equiv & (\forall x)(\forall y)(\exists z)(Rxy \& Syz \& Pza) \supset Sxa & (5) \\ \equiv & (\forall x)(\forall y)(\forall z)((Rxy \& Syz \& Pza) \supset Sxa) & (6) \end{aligned}$$

Formula (1) is given. The equivalence with formula (2) comes by De Morgan's law, used to rewrite the disjunction of negations as a negated conjunction. We are aiming for a conditional so we get to formula (3) by pulling the negation outwards through the \forall -quantifiers, in the process flipping them to \exists , applying the logical equivalence expressing the negation-duality of quantifiers. Formula (4) comes by the material conditional equivalence, resulting in a conditional with existential quantifiers in the antecedent. The variables quantified are y and z , and they do not occur at all in the consequent Sxa of the conditional, so we can then apply the Prenexing law for the y variable to get to formula (5), and then apply it again for the z variable, to arrive at formula (6), which is in universally-quantified program clause form.

In fact, we could also have taken a shorter route from formula (1) straight to:

$$(\forall x)(\forall y)(\forall z)(Sx \vee \neg Rx \vee \neg Sy \vee \neg Pz)$$

using the \forall -Prenexing law for \vee , which pulls the $(\forall y)$ and $(\forall z)$ from inner scope to outer scope across disjunctions. There are invariably multiple possible routes when reasoning through sequences of logical equivalences. This feature may make them seem a little daunting, but they get easier with practice. Now it is your turn. Remember that universal quantifiers *distribute* over conjunction, so breaking into a conjunction of two program clause formulas is easy.

Assume a and b are **NAMES**, x , y , z and w are **VARIABLES**, R is a 2-place predicate and P is a 3-place predicate.

Which one or more of the following formulas of predicate logic with identity is *logically equivalent* to a program clause or a conjunction of program clauses?

- (a) $(\forall x)(\forall y)(\forall z)(\forall w)(\neg Px \vee \neg Py \vee \neg Pz \vee \neg Rxy)$
- (b) $(\forall x)(\forall y)(\forall z)(Px \vee (x = y \vee Pz))$
- (c) $(\forall x)(\forall y)((\exists z)(\exists w)(Px \vee (x = w \vee Pz)) \supset (Rxy \vee Pax))$
- (d) $(\forall x)(\forall y)(\forall z)((Px \vee (x = y \vee Pz)) \supset (Rx \vee Ry))$
- (e) $(\forall x)(\forall y)(\forall w)((\exists z)(Px \vee (x = w \vee Pz)) \supset (Px \vee (x = w \vee Pw)))$

$$\begin{aligned} & (\text{q} \text{ax} \subset (\text{w} \neq \text{x} \wedge \text{z} \neq \text{x})) (\text{zA}) (\text{wA}) (\text{hA}) (\text{xA}) \equiv \\ & (\text{q} \text{ax} \subset \text{w} \neq \text{x}) (\text{zE}) (\text{wA}) (\text{hA}) (\text{xA}) \quad \text{YES} \quad (a) \\ & \text{not a program clause.} \end{aligned}$$

$$\begin{aligned} & (\text{q} \text{ax} \subset (\text{y} \neq \text{x} \wedge \text{z} \neq \text{y})) (\text{zA}) (\text{hA}) (\text{xA}) \quad \text{ON} \quad (p) \\ & (\text{q} \text{ax} \subset (\text{y} \neq \text{x} \wedge \text{z} \neq \text{y})) \text{ not a program clause.} \end{aligned}$$

$$\begin{aligned} & (\text{q} \text{ax} \subset (\text{w} \neq \text{x} \wedge \text{z} \neq \text{x} \wedge \text{w} \neq \text{z})) (\text{wA}) (\text{zA}) (\text{hA}) (\text{xA}) \wedge \\ & (\text{q} \text{ax} \subset (\text{w} \neq \text{x} \wedge \text{z} \neq \text{x} \wedge \text{w} \neq \text{z})) (\text{wA}) (\text{zA}) (\text{hA}) (\text{xA}) \equiv \\ & (\text{q} \text{ax} \subset (\text{w} \neq \text{x} \wedge \text{z} \neq \text{x} \wedge \text{w} \neq \text{z})) (\text{wE}) (\text{zE}) (\text{hA}) (\text{xA}) \quad \text{YES} \quad (c) \\ & (\text{q} \text{ax} \subset \text{h} \neq \text{x}) (\text{zA}) (\text{hA}) (\text{xA}) \wedge \\ & \text{z} \neq \text{x} (\text{zA}) (\text{hA}) (\text{xA}) \equiv \\ & ((\text{q} \text{ax} \wedge \text{h} = \text{x}) \wedge \text{z} \neq \text{x}) (\text{zA}) (\text{hA}) (\text{xA}) \quad \text{YES} \quad (q) \\ & \text{not a program clause.} \end{aligned}$$

$$\begin{aligned} & (\text{q} \text{ax} \subset (\text{w} \neq \text{x} \wedge \text{y} \neq \text{x} \wedge \text{z} \neq \text{x} \wedge \text{w} \neq \text{y} \wedge \text{y} \neq \text{z})) (\text{wA}) (\text{zA}) (\text{hA}) (\text{xA}) \equiv \\ & (\text{q} \text{ax} \subset \text{w} \neq \text{x} \wedge \text{y} \neq \text{x} \wedge \text{z} \neq \text{x} \wedge \text{w} \neq \text{y} \wedge \text{y} \neq \text{z}) (\text{wA}) (\text{zA}) (\text{hA}) (\text{xA}) \quad \text{YES} \quad (a) \\ & \text{not a program clause.} \end{aligned}$$

Of these five examples, only one fails. When you get to a disjunction in the consequent of a conditional, then there is nothing to be done; the formula simply isn't logically equivalent to a program clause. This exercise is intended to illustrate that the class of formulas logically

equivalent to a program clause or a logic program does have within it both some variation and some richness.

6.2.5 | FORMULAS LOGICALLY EQUIVALENT TO GOALS

Next, consider goal formulas occurring on the other side of the logical consequence relation symbol, \models .

A formula A of predicate logic with identity is *logically equivalent to a goal* if and only if there exists a goal $G = (G_1, G_2, \dots, G_k)$ for $k \geq 1$ (list of ***-ATOMS**) such that:

$$A \equiv (\exists \text{var}(G_1))(\exists \text{var}(G_2)) \cdots (\exists \text{var}(G_k))(G_1 \& G_2 \& \cdots \& G_k)$$

is a tautology.

Here, it is important to note that the existential quantifier \exists does *not* distribute over conjunction $\&$. The logical equivalence that is used frequently here is the negation-duality of quantifiers, so a negated-universal $\sim(\forall x)A$ is equivalent to an existential-negation $(\exists x)\sim A$. The De Morgan's equivalences and the negated material conditional equivalence are also often useful. Now its your turn.

Assume a and b are **NAMES**, x, y, z and w are **VARIABLES**, R is a 2-place predicate and P is a 3-place predicate.

Which one or more of the following formulas of predicate logic with identity is *logically equivalent* to a goal?

- (a) $\sim(\forall x)(\forall y)(\forall z)(\sim Pxyz \vee x = y \vee \sim Pazb)$
- (b) $(\exists x)(\exists y)(\exists z)(\exists w)\sim(Pxyz \supset Pazw)$
- (c) $(\exists x)(\exists y)(\exists z)\sim(Pxyz \supset (\forall w)\sim Pazw)$
- (d) $\sim(\forall x)(\forall y)(\forall z)\sim(Pxyz \& x \neq y \& Rxz \& Ryz)$
- (e) $\sim(\forall x)(\forall y)(\forall z)(\sim Pxyz \vee x \neq y \vee Pazb)$

$(\text{q}z\text{a}_d \sim \text{z} \neq x \wedge z\text{fix}_d)(z_E)(h_E)(x_E) \equiv$		
$(\text{q}z\text{a}_d \wedge h \neq x \wedge z\text{fix}_d \sim)(z_A)(h_A)(x_A) \sim$	ON	(e)
$(z \neq x \wedge z\text{fix}_d \sim)(z_E)(h_E)(x_E) \equiv$		
$(z \neq x \wedge z\text{fix}_d \sim)(z_A)(h_A)(x_A) \sim$	YES	(p)
$(\text{m}z\text{a}_d \wedge z\text{fix}_d)(m_E)(z_E)(h_E)(x_E) \equiv$		
$(\text{m}z\text{a}_d \sim m_A \subset z\text{fix}_d \sim)(z_E)(h_E)(x_E) \equiv$	YES	(o)
$(\text{m}z\text{a}_d \sim z\text{fix}_d)(m_E)(z_E)(h_E)(x_E) \equiv$		
$(\text{m}z\text{a}_d \subset z\text{fix}_d \sim)(m_E)(z_E)(h_E)(x_E) \sim$	ON	(q)
$(\text{q}z\text{a}_d \wedge h \neq x \wedge z\text{fix}_d)(z_E)(h_E)(x_E) \equiv$		
$(\text{q}z\text{a}_d \sim \wedge h = x \wedge z\text{fix}_d \sim)(z_A)(h_A)(x_A) \sim$	YES	(r)

6.2.6 | THE HORN CLAUSE FRAGMENT

Now we get to see why *program clauses* (also called *definite clauses* or *positive clauses*) are called *clauses*.

A \star -**FORMULA** A is called a *clause* if and only if A is a disjunction of *literals*, where a literal is either a \star -**ATOM** or else the negation of a \star -**ATOM**; here, we allow disjunctions of size 1 consisting of a single formula.

For example, if A_1, A_2, A_3 and A_4 are all \star -**ATOMS**, then a \star -**FORMULA** of the form:

$$(A_1 \vee \neg A_2 \vee A_3 \vee \neg A_4)$$

is a clause.

A \star -**FORMULA** A is called a *Horn clause* if and only if A is logically equivalent to a clause containing *at most one* positive literal.

A *positive Horn clause* contains *exactly one* positive literal, while a *negative Horn clause* contains *zero* positive literals, so all negative literals.

Two \star -**FORMULAS** A and B are logically equivalent if and only if $(\forall \text{var}(A))A \equiv (\forall \text{var}(B))B$ is a tautology.

A formula A of predicate logic with identity is in the *Horn clause fragment* if there is a Horn clause C such that $A \equiv (\forall \text{var}(C))C$ is a tautology.

Returning to the example above, it is *not* a Horn clause because it has two positive literals. To be in Horn clause form, the clause must have either *exactly one* positive literal (a *positive Horn clause*) or *zero* positive literals (a *negative Horn clause*), and the rest are all negative literals.

With this terminology, we see that:

- a program clause $C : ((A_1 \ \& \ A_2 \ \& \ \dots \ \& \ A_m) \supset B)$ is a Horn clause with *exactly one* positive literal, since it is logically equivalent to $(\sim A_1 \vee \sim A_2 \vee \dots \vee \sim A_m \vee B)$.
- $(\forall \text{var}(C))C$ is a universal quantification of a *positive* Horn clause.
- given a goal $D : (G_1 \ \& \ G_2 \ \& \ \dots \ G_k)$, its negation $\sim D$ is a Horn clause with *zero* positive literals, since the negation $\sim D$ is logically equivalent to the clause $(\sim G_1 \vee \sim G_2 \vee \dots \vee \sim G_k)$.
- $\sim(\exists \text{var}(D))D$ is logically equivalent to $(\forall \text{var}(D))\sim D$, a universal quantification of a *negative* Horn clause.

We end up with negated goal formulas because PROLOG uses a *refutation* approach to automated reasoning: start by assuming all the clauses of the program P are true, while goal G is false, and then try to derive a contradiction. So in the end, the class of formulas that need to be handled are universal quantifications of Horn clauses, lying in the Horn clause fragment.

6.3 | LOGIC PROGRAMMING IN PROLOG

6.3.1 | BASIC PROLOG: A MINI-MANUAL

We summarise the syntax of *basic* PROLOG before getting into examples.

Syntax: A *program clause fact* B , is written in PROLOG as:

$b.$

A *program clause conditional rule* $(A_1 \ \& \ A_2 \ \& \ \dots \ \& \ A_m) \supset B$ is written in PROLOG as:

$b \ :- \ a_1, a_2, \dots, a_m.$

where $m \geq 1$ and a_1, a_2, \dots, a_m, b are all \star -ATOMS translating A_1, A_2, \dots, A_m, B , and written according to PROLOG conventions.

First, note that PROLOG writes program clause conditionals in *reverse* order, with the consequent coming *first*. The consequent is referred to as the *head* of the program clause. It is followed by the symbol colon-dash “ $:-$ ”, read as “*IF*”, and referred to as the *neck* of the clause. Then finally follows the *body* of the clause, which is the list of \star -ATOMS in the antecedent conjunction, with the conjunction connective indicated by commas. Program clause *facts* are declarative; there is no body or neck, just a head.

The *procedural* meaning of a program clause ($m \geq 1$):

$b \ :- \ a_1, a_2, \dots, a_m.$

is: We can establish b if we first establish all of a_1, a_2, \dots, a_m .

This reverse order will make more sense when we see how Prolog works in §6-4: it is the *head* of a program clause that has to be consulted first when processing a PROLOG query.

- Every program clause must end with full-stop symbol “.”
- Identifiers for all *names* and *predicates* must start with a *lower-case letter*, then followed by lower-case letters, upper-case letters, digits or an underscore _.
- Identifiers for *Variables* must start with an *Upper-Case letter*, then followed by lower-case letters, upper-case letters, digits or an underscore “_”. E.g. *Youth*
- To write the *difference* $\star\text{-ATOM}$ ($x \neq y$) in PROLOG syntax, use the “*cannot be unified*” symbol $\backslash=$ and write: $X \backslash= Y$
- To write the *identity* $\star\text{-ATOM}$ ($x = y$) in PROLOG syntax, use the “*can be unified*” symbol $=$ and write: $X = Y$
- There are several different notions of identity or equality used in PROLOG, with the differences between them only apparent when working with functions, complex terms, and arithmetic operations. We are just using a fragment of the language here.

- Use the semi-colon ; key on your keyboard to ask PROLOG if there is more than one existential witness for each of the variables in the goal of the query. After the first witnessing substitution is provided by PROLOG, use the ; key *instead of* the RETURN key to ask “Is there more?”

Non-determinism: queries can have zero, one, two or more answers.

- The underscore symbol _ can be used as an “anonymous variable”, with the meaning “any term”. For example, if a knowledge base about geographical facts was structured by an 8-place predicate `country(Country,Region,Latitude,Longitude,Area_sqmiles,Population,Capital,Currency)`, then to ask what is the currency of Denmark, you can pose the query:

```
?- country(denmark,_,_,_,_,_,_,Currency).
Currency = krone
```

- Programs written in plain text file, with file type “.pl”; that is, names of form “*progname.pl*”.
- The *order* of \star -atoms within the body of a program clause, and the order of program clauses within a program, will really matter for how Prolog executes, which will become clear in the next section, §6-4. For example, when using identity or difference constraints, place them at the *end* of a list of \star -atoms either as the body of a program clause or in a query goal, so that they will be processed last.
- PROLOG uses the pool of *NAMES* occurring in the logic program

and the goal when taking instances of \forall -quantified program clauses and negated goals. The substitution method for assigning **NAMES** to **VARIABLES** is called *unification* in PROLOG.

- Variables occurring in the goal should be all *different* from the variables used in the logic program. This is needed to guarantee the correct behaviour of the unification algorithm. When the same variable occurs in both the goal and a program clause being instantiated, the unification algorithm can get stuck in a loop.
- Predicates are distinguished by (identifier+arity), so in addition to the 8-place predicate `country`, we could have another different 4-place predicate `country(Country,Region,Population,Capital)` that records only 4 of the original 8 arguments.

6.3.2 | EXAMPLE PROLOG PROGRAM

Our running example of a knowledge base will be family relationships between characters in the internationally popular series of books and TV shows, *Game of Thrones*. We start with the main line of the Stark family, with the data structured using the predicate

`parent(a,b)`

with the meaning that child `a` has parent `b`, or a parent of `a` is `b`.

To show the translation into formulas of predicate logic, we will use `P` as a 2-place predicate letter, and the letters `e, r, n, j, c, o, s, a, b, i` as 1-character symbols for names in logic formulas.

program clause facts	predicate logic
<code>parent(rickard_stark,edwyle_stark).</code>	<code>Pre</code>
<code>parent(ned_stark,rickard_stark).</code>	<code>Pnr</code>
<code>parent(jon_snow,ned_stark).</code>	<code>Pjn</code>
<code>parent(rob_stark,ned_stark).</code>	<code>Pon</code>
<code>parent(sansa_stark,ned_stark).</code>	<code>Psn</code>
<code>parent(arya_stark,ned_stark).</code>	<code>Pan</code>
<code>parent(bran_stark,ned_stark).</code>	<code>Pbn</code>
<code>parent(rickon_stark,ned_stark).</code>	<code>Pin</code>
<code>parent(rob_stark,catelyn_tully).</code>	<code>Poc</code>
<code>parent(sansa_stark,catelyn_tully).</code>	<code>Psc</code>
<code>parent(arya_stark,catelyn_tully).</code>	<code>Pac</code>
<code>parent(bran_stark,catelyn_tully).</code>	<code>Pbc</code>
<code>parent(rickon_stark,catelyn_tully).</code>	<code>Pic</code>

Note that identifiers for predicates and names all begin with a *lower-case letter*, and the arguments of predicates are marked with parentheses and commas between arguments when there are two or more.

Since there are multiple Stark family characters with first name begins with the letter “r”, `rob_stark` gets the 1-character name `o`, while `rickon_stark` gets the 1-character name `i`.

The knowledge base additionally includes facts about female and male members of the Stark family, using 1-place predicates `male(a)` and `female(a)`, with corresponding 1-place predicate letters M and F.

program clause facts	predicate logic
<code>male(edwyle_stark).</code>	M _e
<code>male(rickard_stark).</code>	M _r
<code>male(ned_stark).</code>	M _n
<code>male(jon_snow).</code>	M _j
<code>male(rob_stark).</code>	M _o
<code>male(bran_stark).</code>	M _b
<code>male(rickon_stark).</code>	M _i
<code>female(catelyn_tully).</code>	F _c
<code>female(sansa_stark).</code>	F _s
<code>female(arya_stark).</code>	F _a

It is useful to think in terms of the *models* M that satisfy a logic program. If \mathcal{P} is the list of program clause facts given so far, then we can build a model $M = \langle D, I \rangle$ that makes all these fact true by taking as the domain the list of 1-character names:

$$D = \{e, r, n, j, c, o, s, a, b, i\}$$

and then making the interpretations $I(P)$, $I(M)$ and $I(F)$ make true all the atomic facts listed in \mathcal{P} .

$I(P)$	e	r	n	j	c	o	s	a	b	i	$I(M)$	$I(F)$
e											e	1
r	1										r	1
n		1									n	1
j			1								j	1
c											c	1
o		1	1								o	1
s		1	1								s	1
a		1	1								a	1
b		1	1								b	1
i		1	1								i	1

The result is a *partial model*, because we only have a *partial interpretation* of the predicates P, M and F. We have positive information in the form of 1 entries, but no negative or 0 entries. On the basis of the program clauses so far, we do not yet know anything about the other entries in the tables for $I(P)$, $I(M)$ and $I(F)$. One of the intuitions with PROLOG is that by computing logical consequences of a logic program, one learns more about the models of the program. The logical structure of program clauses means that we will only learn *positive* information, and get to possibly fill in more 1s, but we cannot learn any negative information from program clauses.

Now for some *conditional rule* program clauses involving the Stark family characters from *Game of Thrones*. We introduce some additional PROLOG predicates:

`mother(a,b) and father(a,b)`

meaning that child a has mother b, and that child a has father b. For the translations into predicate logic, we use the 2-place predicate letters O and A, since M and F are already in use.

```
mother(rickon_stark,catelyn_tully) :-  
    parent(rickon_stark,catelyn_tully), female(catelyn_tully).  
  
father(rickon_stark,ned_stark) :-  
    parent(rickon_stark,ned_stark), male(ned_stark).
```

These two program clauses translate as the logic formulas:

$$(P_{ic} \& F_c) \supset O_{ic} \quad \text{and} \quad (P_{in} \& M_n) \supset A_{in}$$

So far, we haven't used variables yet; all the logic formulas have been real formulas. Looking at these two examples, we can clearly see that this pattern (mother-of if parent-of and female) is just one instance of a generalization that applies to *all* people. Likewise for the pattern (father-of if parent-of and male). This is where universally quantified variables in program clauses come into play. In PROLOG, variables must be identified by a string that starts with a Capital Letter. While it is often useful to give more descriptive identifiers to variables, X, Y and Z will also work fine.

```
mother(X,Y) :- parent(X,Y), female(Y).  
  
father(X,Y) :- parent(X,Y), male(Y).  
  
siblings(X,Y) :- parent(X,Z), parent(Y,Z), X \= Y.
```

Translated in predicate logic:

$$\begin{aligned} & (\forall x)(\forall y)((P_{xy} \& F_y) \supset O_{xy}) \\ & (\forall x)(\forall y)((P_{xy} \& M_y) \supset A_{xy}) \\ & (\forall x)(\forall y)((\exists z)(P_{xz} \& P_{yz} \& x \neq y) \supset S_{xy}) \end{aligned}$$

The last predicate `siblings`, is a step up in complexity. X is a *sibling* of Y if *there exists* a Z such that a parent of X is Z and a parent of Y is Z and X is different from Y. This is our first example of an "extra" variable in the antecedent of the conditional that is read as *existential* when within the scope is the conditional antecedent. After prenexing by being pulled outwards to the scope of the whole conditional, it becomes *universally quantified*.

Another instance of this pattern is the predicate `grandparent`: X has a *grandparent* Y if *there exists* a Z such that X has parent Z and Z has parent Y.

```
grandparent(X,Y) :- parent(X,Z), parent(Z,Y).  
  
grandmother(X,Y) :- grandparent(X,Y), female(Y).  
  
grandfather(X,Y) :- grandparent(X,Y), male(Y).
```

An important feature of PROLOG is the ease of writing *recursive* program clauses. Intuitively, X has *ancestor* Y if X has parent Y, or if X has parent Z and then Z has parent Y, or there are 3 generations, or 4 generations, or so on, from X back to Y. Written recursively, X has *ancestor* Y

if X has parent Y, or if X has parent Z and Z has *ancestor* Y. This captures the feature that ancestors are inherited down the generations.

```
ancestor(X,Y) :- parent(X,Y).
ancestor(X,Y) :- parent(X,Z), ancestor(Z,Y).
```

Here, it is important to put first the *base* case of the 1-generation link, so that the second recursive clause will “unwind” in stages: first the 2-generation links, then the 3-generation links, and so on. Without care, it is rather too easy to accidentally send PROLOG into an infinite loop. The recursive pattern as used here in the *ancestor* predicate is the most basic way of defining a recursive 2-place predicate.

```

1  /* game-of-thrones-0.pl */
2
3  /* general family relationships */
4  /* usage: parent(X,Y) means child X has parent Y, or a parent of X is Y. */
5
6  mother(X,Y)      :- parent(X,Y), female(Y).
7  father(X,Y)       :- parent(X,Y), male(Y).
8  siblings(X,Y)    :- parent(X,Z), parent(Y,Z), X \= Y.
9  brother_of(X,Y)  :- siblings(X,Y), male(Y).
10 sister_of(X,Y)   :- siblings(X,Y), female(Y).
11 grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
12 grandmother(X,Y) :- grandparent(X,Y), female(Y).
13 grandfather(X,Y) :- grandparent(X,Y), male(Y).
14 ancestor(X,Y)    :- parent(X,Y).
15 ancestor(X,Y)    :- parent(X,Z), ancestor(Z,Y).
16
17 /* facts from the family tree of the Stark family */
18 parent(rickard_stark, edwyle_stark).
19 parent(ned_stark, rickard_stark).           /* ned = edward */
20 parent(jon_snow, ned_stark).
21 parent(rob_stark, ned_stark).
22 parent(sansa_stark, ned_stark).
23 parent(arya_stark, ned_stark).
24 parent(bran_stark, ned_stark).             /* bran = brandon jnr */
25 parent(rickon_stark, ned_stark).
26 parent(rob_stark, catelyn_tully).
27 parent(sansa_stark, catelyn_tully).
28 parent(arya_stark, catelyn_tully).
29 parent(bran_stark, catelyn_tully).
30 parent(rickon_stark, catelyn_tully).
31
32 /* females among Stark family */
33 female(catelyn_tully).
34 female(sansa_stark).
35 female(arya_stark).
36
37 /* males among Stark family */
38 male(edwyle_stark).
39 male(rickard_stark).
40 male(ned_stark).
41 male(jon_snow).
42 male(rob_stark).
43 male(brandon_stark).
44 male(rickon_stark).
45

```

We can combine all our example program clauses into one logic program, which we have called *game-of-thrones-0.pl*, where the .pl part is the file extension for PROLOG programs; the file itself is just plain text. The file lists the program clauses for general family relationships, all using variables understood as universally quantified, followed

by the list of parent predicate facts, and then the male and female predicate facts.

Let \mathcal{P} be the logic program game-of-thrones-0.pl. We can now start putting forth goals \mathcal{G} , and asking PROLOG whether or not $\mathcal{P} \models \mathcal{G}$.

Before seeing a bunch of example queries to \mathcal{P} , yielding some of its logical consequences, we pause first to consider the *models* which make true all the program clause formulas for \mathcal{P} . The smallest such model $M = \langle D, I \rangle$ has 10 objects in the domain:

$$D = \{e, r, n, j, c, o, s, a, b, i\}$$

using the 1-character standard names corresponding to the 10 named persons in the logic program.

As we saw already, the atomic facts for the parent, male and female predicates fix a pattern of 1s in the table for the interpretations of the P, M and F predicates, but leave a lot of the entries blank. We can see going back the logic program clauses, there will be no *additional* atomic facts generated by the program about the parent, male and female predicates, because those three predicates do not appear in the *head* of any of the conditional program clauses. Instead, the program clauses all define *new* predicates like mother, father, ancestor, etc. By taking *instances* of these universally quantified program clauses, we can *learn* new facts about these new predicates, but they will not yield any new facts about the original parent, male or female predicates.

So back with our partial model of the game-of-thrones-0.pl logic program. The partial model can be *completed* by filling in all the blank entries 1 or 0 according to whether or not the corresponding atomic fact is a logical consequence of the program \mathcal{P} . It turns out that for this logic program \mathcal{P} , all those blank entries become 0s. In the theory of logic programming, this is what is called the *Herbrand* model of the logic program.

$I(\mathcal{P})$	e	r	n	j	c	o	s	a	b	i		$I(M)$	$I(F)$
e	0	0	0	0	0	0	0	0	0	0	e	1	0
r	1	0	0	0	0	0	0	0	0	0	r	1	0
n	0	1	0	0	0	0	0	0	0	0	n	1	0
j	0	0	1	0	0	0	0	0	0	0	j	1	0
c	0	0	0	0	0	0	0	0	0	0	c	0	1
o	0	0	1	0	1	0	0	0	0	0	o	1	0
s	0	0	1	0	1	0	0	0	0	0	s	0	1
a	0	0	1	0	1	0	0	0	0	0	a	0	1
b	0	0	1	0	1	0	0	0	0	0	b	1	0
i	0	0	1	0	1	0	0	0	0	0	i	1	0

A PROLOG interpreter essentially uses the Herbrand model of a logic program to answer queries of that program. In answering `false`. to a query, PROLOG can point to the Herbrand model as a counter-model demonstrating failure of logical consequence.

For a simple example of a program whose Herbrand model would see “extra” 1s appear in the entries left blank by facts, consider one expressing facts about *adjacency* of objects, of the form: `adjacent1(a, b)`,

together with universally quantified clauses expressing that adjacency is a *symmetric* predicate:

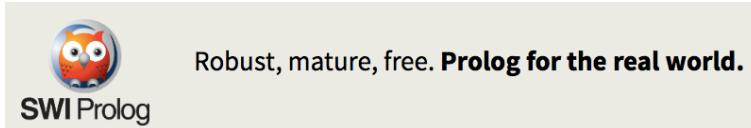
```
adjacent(X,Y) :- adjacent1(X,Y).
adjacent(X,Y) :- adjacent1(Y,X).
```

Here, a partial model for the program has interpretations for two 2-place predicates, say A_1 and A , with the atomic facts for the `adjacent1` predicate fixing 1 entries in the interpretation $I(A_1)$. Then taking the logical consequences of the program to fill out the interpretations of $I(A_1)$ and $I(A)$ in the Herbrand model, the relation $I(A)$ will be the *symmetric closure* of the relation $I(A_1)$: put in all the 1s required by the atom facts fixing $I(A_1)$, then add in all 1s for the symmetric pairs.

6.3.3 | EXAMPLE PROLOG QUERIES

There are numerous implementations of PROLOG around. We use:

<http://www.swi-prolog.org>



Copyright ©1990-2013 University of Amsterdam

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/>

Start up the SWI-Prolog software, and load in the program; you can use the menu item **File→Load** or **File→Consult**, and choose the file `game-of-thrones-0.pl`. There are 10 universally quantified clauses expressing family relationship predicates; 13 parent facts and 10 male/female facts, to give a total of 33 clauses.

```
% /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/game-of-thrones-0.pl
```

```

compiled 0.00 sec, 33 clauses
1 ?-
1 ?- siblings(jon_snow,sansa_stark).
true .
2 ?- siblings(rob_stark,rickard_stark).
false.
3 ?- grandparent(rob_stark,rickard_stark).
true ;
false.
4 ?-

```

After the first query, we hit the RETURN key on the keyboard, and get the answer `true`: Jon Snow and Sansa Stark are siblings. Hit the RETURN key again and a full stop appears, and then the query prompt returns with `2 ?-`. Our second query gets the answer `false.` and immediately the query prompt returns with `3 ?-`. Our third query about Rickard as grandparent gets the answer `true`: Rob Stark has grandparent Rickard Stark. We then try our luck with the “any more?” key “`;`” and get the answer `false.`; in this context, it means “no, there are no more answers other than `true`”.

Any string of characters starting with a *Capital letter* is treated as a *Variable*, so we can pick more meaningful identifiers, for example: `Youth`. When a goal has variables, using the “`;`” key is the means to ask PROLOG if there are any *more* existential witnesses, so it only makes sense to use it after PROLOG has answered with the first existential witnesses to a query.

Using the variable `Youth`, we can ask which youthful persons have Rickard Stark as their grandparent?

```

4 ?- grandparent(Youth,rickard_stark).
Youth = jon_snow ;
Youth = rob_stark ;
Youth = sansa_stark ;
Youth = arya_stark ;
Youth = bran_stark ;
Youth = rickon_stark ;
false.
5 ?-

```

After the first answer `Youth = jon_snow`, we repeatedly responded with the “`;`” until the PROLOG interpreter exhausts all the candidates and finally comes back with `false.` and returns to the query prompt. When you use a PROLOG implementation yourself, you will notice that the semi-colon is not actually printed on the screen; we include it here to make clear what is being entered from the keyboard to get this interaction.

Next, we experiment with a query with two variables. We want to list all the `(Youth,Elder)` pairs where the `Youth` has the `Elder` as their

grandfather.

```
5 ?- grandfather(Youth,Elder).
Youth = ned_stark, Elder = edwyle_stark ;
Youth = jon_snow, Elder = rickard_stark ;
Youth = rob_stark, Elder = rickard_stark ;
Youth = sansa_stark, Elder = rickard_stark ;
Youth = arya_stark, Elder = rickard_stark ;
Youth = bran_stark, Elder = rickard_stark ;
Youth = rickon_stark, Elder = rickard_stark ;
false.
```

6 ?-

For this query, PROLOG gets through seven (Youth,Elder) pairs related by the `grandfather` predicate, before exhausting all the candidates and finally coming back with `false`. to signify “no more”.

What about the question: “Do Jon Snow and Arya Stark have a parent in common?” One way to ask the question is to use two variables, say P1 and P2 together with an additional sub-goal `P1 = P2` requiring that P1 and P2 be unified by substituting in the same name.

```
6 ?- parent(jon_snow,P1), parent(arya_stark,P2), P1=P2.
P1 = P2, P2 = ned_stark ;
false.
```

7 ?-

The query answer is YES, witnessed by Ned Stark, but there are no other witnesses for P1 and P2.

Lets see the recursive predicate `ancestor` at work. We ask the question: Who is an ancestor of Bran Stark?

```
7 ?- ancestor(bran_stark,A).
A = ned_stark ;
A = catelyn_tully ;
A = rickard_stark ;
A = edwyle_stark ;
false.
```

8 ?-

The ancestors of Bran Stark start with his closest ancestor, his father Ned Stark, as PROLOG starts with the base clause of the `ancestor` predicate that reduces to the `parent` predicate, and in the program list order, the *first* fact that has `bran_stark` in the first argument position of the `parent` predicate is `parent(bran_stark,ned_stark)`. The second fact in program list order that has `bran_stark` in the first argument position of the `parent` predicate is `parent(bran_stark,catelyn_tully)`, so the answer `A = catelyn_tully` is next. After exhausting all the `parent` matches, PROLOG moves to the recursive clause for the `ancestor` predicate, and “unwinds” one step, to parents of parents. This yields the answer `A = rickard_stark`, but no other parents of parents. PROLOG then applies the recursive clause again, and “unwinds” one more step, to parents of grand-parents. This gets the great-grandfather Edwyle Stark, before PROLOG returns with `false`. to signify “no more”.

Because of the way the recursive `ancestor` predicate was constructed with the base clause first, and the recursive clause second, the “unwind-

ing” will start with nearest ancestor and proceeds backwards through generations.

To compare, lets run the ancestors of Bran Stark query again, but after the first answer for `A = ned_stark`, we just hit the RETURN key instead of the semi-colon.

```
8 ?- ancestor(bran_stark,A).  
A = ned_stark .  
9 ?-
```

Then PROLOG goes straight back to the query prompt, interpreting this as meaning the user is happy to stop with the answers already given.

```
9 ?- ancestor(catelyn_tully,A).  
false.  
10 ?- ancestor(bran_stark,A), A=edwyle_stark .  
A = edwyle_stark ;  
false.  
11 ?-
```

There is no data in the logic program `game-of-thrones-0.pl` about parents of Catelyn Tully, so a query asking for her ancestors returns with `false`.

To test if two different names belong to the same person (which they don’t, in our sample knowledge base), we obviously use identity.

```
11 ?- A = bran_stark, B = rickon_stark, A = B.  
false.  
12 ?- bran_stark = A, B = rickon_stark, A = B.  
false.  
13 ?- bran_stark = rickon_stark.  
false.  
14 ?- bran_stark \= rickon_stark.  
true .  
15 ?-
```

Now it is your turn to write some PROLOG code. For this exercise, note that (first) `cousins` share a grandparent but don’t share a parent; if they share a parent, they are `siblings`.

Which of the following PROLOG program clauses correctly express that `X` and `Y` are *cousins*?

- (a) `cousins(X,Y) :- parent(X,W), parent(W,Z), parent(Y,V), parent(V,Z), X \= Y.`
- (b) `cousins(X,Y) :- parent(X,W), parent(W,Z), parent(Y,V), parent(V,Z), X \= Y, W \= V.`
- (c) `cousins(X,Y) :- grandparent(X,Z), grandparent(Y,Z), X \= Y.`
- (d) `cousins(X,Y) :- parent(X,W), parent(Z,W), parent(Y,V), parent(Z,V), X \= Y, W \= V.`

```

Λ =\ M , Λ =\ X , Λ =\ Y , parent(Y,V) , parent(Z,V) , parent(Z,W) ,
    NO   cousins(X,Y) :- parent(X,M) , parent(Z,W) ,  

          grandparent(Y,Z) , X =\ Y .  

(c)   NO   cousins(X,Y) :- grandparent(X,Z) ,  

          parent(Y,V) , parent(V,Z) , X =\ Y .  

Λ =\ M , Λ =\ X , Λ =\ Y , parent(Y,V) , parent(X,W) , parent(W,Z) ,
    YES  cousins(X,Y) :- parent(X,W) , parent(W,Z) ,  

          parent(Y,V) , parent(V,Z) , X =\ Y .  

(d)   NO   cousins(X,Y) :- parent(X,W) , parent(W,Z) ,  

          parent(Y,V) , parent(V,Z) , X =\ Y .
(e)

```

The `cousins` predicate won't be much fun with the data in our first example logic program, but there is a larger knowledge base called `game-of-thrones-1.pl` available for download, in addition to the original logic program `game-of-thrones-0.pl`. There is a full listing of the larger knowledge base `game-of-thrones-1.pl` at the end of this chapter. It has a bunch more *Game of Thrones* family relationship facts as well as few more family relationship predicates. It has a total of 113 program clauses, and includes data for the Lannister family of characters, as well as a bit more of the Stark family tree, so there will be many more *Game of Thrones* facts to discover!

6.3.4 | QUERIES AND RESPONSES IN PROLOG

Let \mathcal{P} be a **logic program**: a list C_1, C_2, \dots, C_n of program clauses. Let \mathcal{G} be a **goal**: a list of \star -**ATOMS** after the query prompt:

`?- g1, g2, ..., gk.`

Let the \star -formula D be the conjunction of the sub-goal g_j 's.

Automated reasoning task addressed by PROLOG: determine whether or not $\mathcal{P} \models \mathcal{G}$; that is,

$(\forall \text{var}(C_1))C_1, (\forall \text{var}(C_2))C_2, \dots, (\forall \text{var}(C_n))C_n \models (\exists \text{var}(D))D$

To get PROLOG to answer the question whether or not $\mathcal{P} \models \mathcal{G}$,

- load the program file containing \mathcal{P} into PROLOG (use `File→Consult...` or `File→Load...` in some versions).
- enter the goal \star -atoms in \mathcal{G} in a list after the query prompt `?-`
- end list with full-stop `.` and then press RETURN key;
- then PROLOG answers with either:
 - *existential witnesses* for the variables in \mathcal{G} , or
 - `true` or `false`, or ...
 - there is no answer when PROLOG gets stuck in a loop and crashes!

The proof method is a *refutation* procedure: in effect, we suppose the program \mathcal{P} is true but the initial goal \mathcal{G} is false, and then PROLOG tries to derive a contradiction. Unlike the proof tree method, there will be only *one* inference rule, called *resolution*.

We will discuss some of the details of a PROLOG implementation in the next section, **§6-4 How PROLOG Answers Queries**.

Note to students from LLI1: Our project example in propositional PROLOG was toy sized 4×4 *Sudoku* puzzles, and we noted there that it would be much easier to tackle such puzzles using predicate PROLOG. A few remarks:

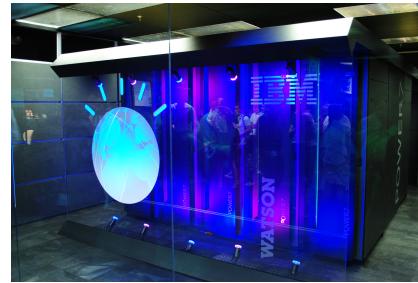
- We could use predicate PROLOG (plus some more advanced features required to write to files) to generate the propositional PROLOG program code that solves a given 4×4 Sudoku puzzle. But we won't, given the length constraints of this course and these notes. The predicate structure in a Sudoku puzzle is less interesting than that found in family relationships, so we have gone for the latter.
- The most concise way to solve a 4×4 or a full-sized 9×9 Sudoku in PROLOG is using list manipulation operations, including the `all_distinct` test. For example, see one PROLOG solution at: <http://programmablelife.blogspot.com.au/2012/07/prolog-sudoku-solver-e.html>

6.4 | HOW PROLOG ANSWERS QUERIES

6.4.1 | PROLOG USED IN CURRENT AI SYSTEMS

But before we get into details about how PROLOG goes about answering queries, we start with a brief discussion of the use of PROLOG in current AI systems. You might be wondering whether PROLOG is mostly of academic or theoretical interest, and perhaps is not really of much practical use.

Among programming languages, PROLOG is certainly much more specialised in its use than well-known general-purpose imperative languages like C or Java. PROLOG is used for natural language processing, expert systems, automated question-answering, knowledge representation and information ontologies. An extension called *constraint logic programming* is particularly good for large-scale combinatorial optimisation problems, such as production scheduling and timetabling.



A recent use of Prolog is within IBM's top-of-the-line artificial intelligence computer called *Watson*. Watson's capabilities were publicly demonstrated in 2011 when it competed on the US TV game show *Jeopardy*. The game show *Jeopardy* has the twist that questions are posed in the form of a natural language sentence, and the competitors' task is to provide the question that the sentence answers. Armed with the entire contents of Wikipedia and a good deal else in its knowledge database (but not hooked up to the internet), and using PROLOG for natural language processing and pattern matching, Watson consistently out-performed the champion human competitors on the game show. IBM's current projects with Watson include clinical decision making in medicine, and another called *Project Lucy*, aimed at addressing development problems in countries across Africa, beginning with healthcare and education.

6.4.2 | HISTORY OF PROLOG

Our explanation of *how* and *why* PROLOG works starts with its history.

- **John Alan Robinson** [1928–]:

In 1965, the American philosopher (by training), mathematician and computer scientist, John Robinson, developed the proof method of *resolution* with *unification* for the Horn clause fragment of predicate logic. We will learn about these in this lesson.

- **Robert Kowalski** [1941–]:

In 1971, the American/British logician and computer scientist Robert Kowalski developed with Donald Kuehner a *procedural* interpretation of Robinson's resolution with Horn clauses, resulting in what is known as *Selective Linear Definite clause resolution*, or *SLD resolution*. This is the inference engine of Prolog.

- **Alain Colmerauer** [1941–] and **Phillipe Roussel** [1945–]:

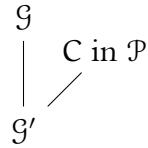
The language Prolog was developed by the French computer scientist Alain Colmerauer and first implemented by Colmerauer and Phillippe Roussel in 1972. The name "Prolog" was chosen by Roussel as an abbreviation for "*programmation en logique*" (French for "*programming in logic*").

6.4.3 | RESOLUTION PROOF METHOD USED BY PROLOG

From the previous section, we know how to write program clauses and logic programs in predicate PROLOG notation, and how to ask PROLOG

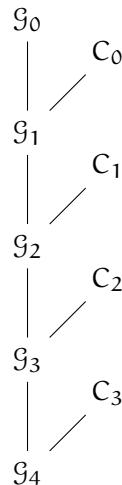
whether or not an existentially quantified goal is a logical consequence of the universally quantified program clauses in a logic program. Now, we get inside PROLOG to find out how and why it gives the answers it does.

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 C in a program \mathcal{P} .



The proof method constructs a chain or sequence of goals with repeated applications of the rule. As noted in the brief history, the rule implemented in PROLOG is called *Selective Linear Definite clause resolution* or *SLD resolution*.

The *Linear* part of the rule name arises because we can use a “linear tree” diagram to represent a sequence of rule applications, with program clauses C from logic program \mathcal{P} appearing as side inputs to the linear sequence of goals \mathcal{G}_i . *Definite clause* is alternative terminology for *program clause*, and these serve as side inputs to the linear sequence of goals.



Practically, in the concrete examples we will see, the tree diagrams quickly get too hard to draw, so we will use an annotated linear list instead.

The *S* in *SLD* resolution is for the principle of *selection* used to choose program clauses. While more complex alternative selection principles can be devised, PROLOG uses the very simple selection principle of “take the first matching program clauses that has not already been tried”. We will see this in action in examples.

Now to the *resolution proof method*. We include among the goals \mathcal{G}_i the *empty goal*, written “ \square ”, which is the empty list of atoms. Reaching the empty goal \square will signify that a *contradiction* has been reached.

The proof method implemented in PROLOG is a *refutation* procedure, which 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, and then PROLOG applies resolution steps with the aim of deriving the empty goal \square .

Supposing goal \mathcal{G} is false means supposing the *universal quantification* of the *negation* of the goal conjunction is true. This in turn means that under *all ways* of instantiating the variables occurring in the goal \mathcal{G} , at least one of the sub-goals g_i is false.

The **LLI1 Course Notes** §6-4 covers the resolution proof method in the simpler setting of propositional PROLOG. The extra ingredients in predicate PROLOG are quantifiers and instantiation of variables. Because we are working with program clauses and the negations of goals, we have universally quantified Horn clauses. In the predicate logic semantics as used in the core sections of the course, this means we are free to instantiate any variable with any **NAME**. In our present setting, we have revised the way formulas are formed to allow variables as well as names in the class of **TERMS**. In giving an answer to a PROLOG query, each variable needs to end up being instantiated concretely with a **NAME**. But along the way, in the course of processing a query, we will be instantiating variables with **TERMS**, which means names or variables.

The process for assigning terms to variables in Prolog is called *unification*. We will start with a simple example. The program \mathcal{P} is our larger collection of popular culture trivia, `game-of-thrones-1.pl`. For the goal \mathcal{G} , we have the query about who are the grandchildren of Rickard Stark, where the variable is the identifier `Youth`.

```
?- grandparent(Youth,rickard_stark).
```

Now, the *first* clause in program list order in \mathcal{P} whose head is the same predicate `grandparent` is:

```
C : grandparent(X,Y) :- parent(X,Z), parent(Z,Y).
```

with universally quantified variables `X`, `Y`, and `Z`. The active sub-goal `grandparent(Youth,rickard_stark)` and the clause C can be *unified* or *matched* by the substitution:

$$\theta : \{X = \text{Youth}, Y = \text{rickard_stark}\}$$

which assigns variable `X` the term `Youth` (which is a variable), and assigns variable `Y` the term `rickard_stark` (which is a name). Applying the substitution to the program clause C results in the substitution instance:

```
 $\theta C$ : grandparent(Youth,rickard_stark) :-  
parent(Youth,Z), parent(Z,rickard_stark).
```

The substitution *unifies* them because the head of clause θC is now a *direct match* with the sub-goal `grandparent(Youth,rickard_stark)`, and the resolution step can be applied (explained below, and as before, for those who have done propositional PROLOG in **LLI1**).

Note the direction of *substitutions*; they are mappings *from* a variable *to* a term (a name or a variable). We will always list them in that order, with the variable on the left-hand side of the $=$, and the term on the right-hand side. Also note that every substitution instance of a universally quantified program clause in \mathcal{P} is a logical consequence of program \mathcal{P} , and so can be used in computing logical consequences of a program \mathcal{P} .

One further point: substitutions do not have any affect on formulas that don't contain the variables that are the subject of the substitution. For example, if C is a PROLOG formula with *no* occurrences of variables X and Y , and θ is a substitution that assigns $X = t_1$ and $Y = t_2$, where t_1 and t_2 are terms, then θC is just C , with no change.

In this short introduction to predicate PROLOG, we cannot give a detailed treatment of unification. Instead, we will illustrate the ideas by examples. Here is a second example, a bit more general than the first.

Suppose p is a 5-place predicate in PROLOG, and our goal G consists of the single \star -ATOM $p(A, B, a, b, c)$, with variables A and B in the first two argument positions, and names a , b , and c . So the query is asking do there exist objects A and B such that $p(A, B, a, b, c)$ holds.

This active sub-goal $p(A, B, a, b, c)$ can be *unified* with any program clause C whose head:

- (i) is the same predicate p , and all five of the arguments of p in the head of C are variables; or
- (ii) is the same predicate p , and the first two arguments of p in the head of C are variables, and the remaining three arguments of p are the names a , b , and c , in that order.

In case (i), a program clause such as:

$$C_1 : p(X, Y, Z, W, V) :- r(X, Y, Z), s(Z, W, V).$$

can be *unified* with $p(A, B, a, b, c)$ by the substitution:

$$\theta_1 : \{X = A, Y = B, Z = a, W = b, V = c\}$$

resulting in the substitution instance:

$$\theta_1 C_1 : p(A, B, a, b, c) :- r(A, B, a), s(a, b, c).$$

In case (ii), a program clause such as:

$$C_2 : p(X, Y, a, b, c) :- r(a, b, X), s(Y, b, c).$$

can be *unified* with $p(A, B, a, b, c)$ by the substitution:

$$\theta' : \{X = A, Y = B\}$$

resulting in the substitution instance:

$\theta_2 C_2: p(A, B, a, b, c) :- r(a, b, A), s(B, b, c).$

In the theory of unification, it can be proved that for any substitution problem of this kind, there exists a *most general unifier* that does the best job. The performance criterion *does the best job*, and the modifier *most general*, can be formalised precisely (in terms of composition of substitutions considered as functions), but we will not do that here. The intuitive meaning is a substitution that will minimise the total number of substitution steps needed to reach the final objective of mapping all variables to names. In the setting of full PROLOG with functions, the final objective is mapping all variables to so called *ground terms*, which are function term expressions built up from names.

In the examples we have seen and will see, the *most general unifier* will be the most obvious substitution.

So, what is this resolution rule? Rather than try to describe resolution in full generality, considering \star -atoms with n -many variables and m -many names, and writing all that out, we will state the rule a bit more concretely, in the case of two variables, A and B , occurring in the first sub-goal being processed. The version for one or three or more variables requires only the obvious modification. In the case that the sub-goal being processed does not have any variables at all, unification may still be required in order to find a program clause C in \mathcal{P} whose head matches that sub-goal. We will see this in examples.

So suppose we are given a goal \mathcal{G} with first sub-goal a predicate p with arguments including some variables A and B , and perhaps some other sub-goals g_1, g_2, \dots, g_k following.

SLD resolution rule:

Given a goal of the form: $\mathcal{G} : p(A, B), g_1, g_2, \dots, g_k$. with $k \geq 0$, PROLOG will start work on the first sub-goal $p(A, B)$ in \mathcal{G} , and will search for the first program clause C in \mathcal{P} whose head has the same predicate p as the active sub-goal, and for which there exists a substitution θ unifying the clause head of C with the active sub-goal $p(A, B)$.

If C is a *fact*: $p(X, a)$.

then use the *most general unifier* θ of $p(A, B)$ with $p(X, a)$,

namely $\theta : \{A = X, B = a\}$,

to form the *resolvent* of \mathcal{G} with clause C under θ :

$\mathcal{G}' = \theta g_1, \theta g_2, \dots, \theta g_k$.

which simply deletes $p(X, a)$ from $\theta \mathcal{G}$.

If \mathcal{G} is just $p(A, B)$, so $k = 0$, then \mathcal{G}' is empty, \square .

The new goal \mathcal{G}' resulting from this resolution step is called the *resolvent* of goal \mathcal{G} with clause C under substitution θ . It consists of all the remaining sub-goals under the θ substitution, $\theta g_1, \theta g_2, \dots, \theta g_k$. This is the same as applying substitution θ to the whole goal \mathcal{G} and then simply deleting the first sub-goal $p(X, a) = \theta p(A, B)$. The resolution step is said to be *resolving on* the atom $p(X, a)$.

Under the refutation interpretation, we are supposing that \mathcal{G} is false and all the clauses in \mathcal{P} are true. Supposing \mathcal{G} is false means that the *universally quantified* $\sim D$ is true, where D is the conjunction of the sub-goals in \mathcal{G} . This means that for every substitution θ , at least one of the sub-goals in $\theta \mathcal{G}$ is false, where $\theta \mathcal{G}$ is $\theta p(A, B), \theta g_1, \theta g_2, \dots, \theta g_k$. Since $(\forall \text{var}(C))C$ is true for all clauses in \mathcal{P} , this includes the fact clause $C : p(X, a)$. Under the substitution θ , the first sub-goal $\theta p(A, B)$ exactly matches the fact $p(X, a)$. This means that sub-goal can be simply deleted from consideration, on the grounds that it cannot be this sub-goal that is false under substitution θ . We can thus turn our attention to the next sub-goal on the list, if there is one.

In summary, resolving with facts is *good!* It shortens the length of the list of sub-goals. If the sub-goal being processed is the only one in \mathcal{G} , then the final resolvent is the empty goal, shown by the symbol \square .

The second case of resolution is when the program clause C that can be unified with the first sub-goal is a conditional.

If C is a *conditional rule* or *proper program clause*, with $m \geq 1$:

$C : p(X, Y) :- a_1(X, Y), a_2(X, Y), \dots, a_m(X, Y).$

then use the *most general unifier* θ of $p(A, B)$ with $p(X, Y)$

namely $\theta : \{X = A, Y = B\}$,

to form the *resolvent* of \mathcal{G} with clause C under θ :

$\mathcal{G}' = a_1(A, B), a_2(A, B), \dots, a_m(A, B), \theta g_1, \theta g_2, \dots, \theta g_k.$

which replaces $\theta p(A, B)$ in $\theta \mathcal{G}$ with body atoms $\theta a_i(X, Y) = a_i(A, B)$ for $i = 1, 2, \dots, m$ from instance θC of $(\forall \text{var}(C))C$.

The new goal \mathcal{G}' resulting from this resolution step is the *resolvent* of goal \mathcal{G} with clause C under substitution θ . It consists of all the body \star -atoms of the clause C under the substitution θ , followed by the other sub-goals g_1 up to g_k , they also under the substitution θ . This will become clearer when we see examples.

Under the refutation interpretation, we are supposing that \mathcal{G} is false and all the clauses in \mathcal{P} are true. Supposing \mathcal{G} is false means that the *universally quantified* $\sim D$ is true, where D is the conjunction of the sub-goals in \mathcal{G} . This means that for every substitution θ , at least one of the sub-goals in $\theta \mathcal{G}$ is false, where $\theta \mathcal{G}$ is $p(A, B), \theta g_1, \theta g_2, \dots, \theta g_k$, since $\theta p(A, B)$ is just $p(A, B)$, matching clause head $\theta p(X, Y)$. Since $(\forall \text{var}(C))C$ is true, every instance θC of the conditional program clause C must be true. If it is $p(A, B)$ that is false, then this is the head or consequent of θC , so we can thus apply the principle of *modus tollens* or “denying the consequent”, and infer that the antecedent of θC must

be false, and thus one of the body atoms $\theta a_i(X, Y) = a_i(A, B)$, for $i = 1, 2, \dots, m$, must be false. Thus we get the resolvent \mathcal{G}' as described, with the resolution step said to be *resolving on* the atom $\theta p(A, B)$

Note that in the resolvent \mathcal{G}' , the first in the list is now $a_1(A, B)$, the result of applying θ to $a_1(X, Y)$, the first \star -atom in the body of conditional clause C, so it will become the active sub-goal within the next resolution step.

In our class of \star -ATOMS, we include both difference and identity formulas. We now deal separately with the cases where the first sub-goal being processed is a difference or identity atom.

Difference atom rule:

Given a goal of the form: $\mathcal{G} : a \setminus= b, g_1, g_2, \dots, g_k$. with $k \geq 0$, where a and b are *distinct names*, and $a = b$ is *not* a logical consequence of the logic program \mathcal{P} ,

then form the *resolvent* $\mathcal{G}' : g_1, g_2, \dots, g_k$.

obtained by simply deleting the difference atom sub-goal.

Under the refutation approach, we are assuming that at least one of the sub-goals in \mathcal{G} is false. When the first of those sub-goals is $a \setminus= b$, where a and b are *distinct names*, and $a = b$ is *not* a logical consequence of \mathcal{P} (the PROLOG interpreter spawns a separate sub-query to answer this), then this sub-goal $a \setminus= b$ will in fact be true. It can thus be deleted from consideration, on the grounds that it cannot be this sub-goal that is false in this case, so we can turn attention to the next sub-goal on the list, if there is one.

For the PROLOG fragment we are using, we need two rules for resolving identity sub-goals: one for *self-identity* sub-goals, and another for identities describing *substitutions*.

Self-identity rule:

Given a goal of the form: $\mathcal{G} : t = t, g_1, g_2, \dots, g_k$. with $k \geq 0$, where t is any *term* (variable or name),

then form the *resolvent* $\mathcal{G}' : g_1, g_2, \dots, g_k$.

obtained by simply deleting the self-identity sub-goal.

Under the refutation approach, we are assuming that at least one of the sub-goals in \mathcal{G} is false. When the first of those sub-goals is a self-identity assertion $t = t$, where term t is any variable or name, then this sub-goal will in fact be true. It can thus be deleted from consideration, on the grounds that it cannot be this sub-goal that is false in this case, so we can turn attention to the next sub-goal on the list, if there is one.

Identity substitution rule:

Given a goal of the form:

$$G : X = t, g_1, g_2, \dots, g_k. \quad \text{or} \quad G : t = X, g_1, g_2, \dots, g_k.$$

with $k \geq 0$, where X is a variable and t is any *term* (variable or name) different from X ,

then form the *substitution*: $\theta : \{X = t\}$

and then form the *resolvent* $G' : \theta g_1, \theta g_2, \dots, \theta g_k.$

obtained by applying the obvious substitution θ and then deleting the identity sub-goal describing that substitution.

In this rule, an identity sub-goal is simply transformed into a substitution which is applied to the remaining sub-goals. It can thus be deleted from consideration, on the grounds that it is not this sub-goal that is false in this case, so we can turn attention to the next sub-goal on the list, if there is one.

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 G_0, G_1, \dots, G_n via resolution with program clauses C_0, C_1, \dots, C_{n-1} from \mathcal{P} , together with substitutions θ_i for $i < n$ unifying the first sub-goal of G_i with clause C_i , do the following:

- Mark a goal G_i as a *choice point* if and only if there are two different pairs (C_i, θ_i) and (C'_i, θ'_i) of program clauses and substitutions such that θ unifies the first sub-goal of G_i with clause C_i , and θ'_i unifies the first sub-goal of G_i with clause C'_i , and both are different from any previous (program clause, substitution) pairs that have been tried on this goal G_i before.
- Mark a goal G_i as a *dead-end* if and only if the first sub-goal g of G_i is such that there are *no* pairs (C, θ) such that θ unifies the head of C with g and (C, θ) has not already been explored in earlier resolution steps G_j for $j < i$.
In this case, *backtrack* through the sequence to locate the nearest goal 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 G_i .
- Continue to extend the sequence from G_n to a resolvent G_{n+1} , in a *depth-first* manner, whenever G_n is not \square , G_n is not a dead-end, and either the **Difference atom rule** or the **Self-identity rule** or the **Identity substitution rule** is

applicable, or else there exists a program clause C_n in \mathcal{P} with head unifiable with the first sub-goal of \mathcal{G}_n by some substitution θ_n , such that the pair (C_n, θ_n) has not yet been explored.

Three cases of dead-end goals \mathcal{G}_n are:

- When the first sub-goal g of in \mathcal{G}_n is such that there are *no* pairs (C, θ) – either none at all, or none that have not yet been explored through earlier choice points – where C is a clause in \mathcal{P} with head unifiable with the first sub-goal in \mathcal{G}_n by substitution θ .
- When the first sub-goal in \mathcal{G}_n is $a \backslash= a$ where a is a name.
- When the first sub-goal in \mathcal{G}_n is **false**.

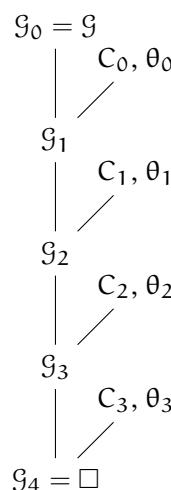
The first case of dead-end goals is when there are no matching pairs (C, θ) to unify with the first sub-goal of the goal. This is the most common situation of dead-ends in predicate PROLOG resolution searches.

The second case of dead-end goals is when the first sub-goal is a clearly false difference assertion $a \backslash= a$ where a is a name.

The third case of dead-end goals is more common in propositional PROLOG, where “dummy” clauses $p :- \text{false}$ are needed for any propositional atom p that otherwise has no clause in the program with it as head, and these “dummy” clauses lead by resolution to the constant **false** appearing in a goal.

When a program \mathcal{P} is compiled, a PROLOG interpreter will usually complain if there are any predicates occurring in the body of a rule clause in \mathcal{P} for which there are no program clauses (rules or facts) in \mathcal{P} with that predicate as head. If the problem predicate is p and, for example, has arity 3, then the PROLOG interpreter will treat p as “undefined” and produce an error message like the following:

ERROR: Undefined procedure: $p/3$



The proof method of resolution with unification precedes in a sequence of steps. At each resolution step, the goal \mathcal{G}_i and the clause C_i together with a substitution θ_i are combined to produce a new goal

\mathcal{G}_{i+1} . Depicted as a tree diagram, a *resolution refutation sequence* starting from \mathcal{G} and ending with \square by resolving with clauses C_i in logic program \mathcal{P} can be depicted as above. Note that the final existential witness substitution for all variables in \mathcal{G} is extracted from the substitutions $\theta_0, \theta_1, \theta_2$ and θ_3 used within the resolution refutation sequence.

Let \mathcal{P} be a logic program loaded into a PROLOG interpreter, and suppose goal \mathcal{G} is g_1, g_2, \dots, g_k , where each of the predicates in the sub-goals g_j occur in the head of at least one program clause within \mathcal{P} . In response to the query:

`?- g1, g2, ..., gk.`

that is, the question whether or not $\mathcal{P} \models \mathcal{G}$, the PROLOG interpreter will have one of three responses:

- it answers with existential witnesses for each of the variables in \mathcal{G} , or just `true` or `yes` if there are no variables in \mathcal{G} ;
- 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` or `yes` to query \mathcal{G}

`?- g1, g2, ..., gk.`

that is, the question whether or not $\mathcal{P} \models \mathcal{G}$, and will give a substitution for all variables in \mathcal{G} ,

if and only if PROLOG finds a sequence of goals $\mathcal{G}_0, \mathcal{G}_1, \dots, \mathcal{G}_n$ such that:

- $\mathcal{G}_0 = \mathcal{G}$;
- for each $i < n$, goal \mathcal{G}_{i+1} is the resolvent of \mathcal{G}_i , either by application of an identity or difference rule, or because \mathcal{G}_{i+1} results from \mathcal{G}_i by resolution with a pair (C_i, θ_i) such that substitution θ_i is a most general unifier for the first sub-goal in \mathcal{G}_i and the program clause C_i in \mathcal{P} ;
- $\mathcal{G}_n = \square$.

Such a sequence is called a *resolution refutation sequence* for the query.

We return to our knowledge base of popular culture trivia with the logic program \mathcal{P} : `game-of-thrones-1.pl`, the full text of which can be found at the end of this chapter.

```
% /Users/jdavoren/prolog/Prolog-code/game-of-thrones-1.pl
compiled 0.00 sec, 113 clauses
1 ?- mother(sansa_stark,M).
M = catelyn_tully.
2 ?-
```

Our query asks PROLOG to name the person M who is the mother of Sansa Stark. The PROLOG interpreter comes back with $M = \text{catelyn_tully}$,, but no other answers, so it then returns to the query prompt. The resolution and unification steps taken by the PROLOG interpreter are as follows.

```

 $\mathcal{G}_0 = \text{mother}(\text{sansa\_stark}, M).$ 
   $C_0: \text{mother}(X, Y) :- \text{parent}(X, Y), \text{female}(Y).$ 
   $\theta_0: \{X = \text{sansa\_stark}, Y = M\}$ 
 $\mathcal{G}_1 = \text{parent}(\text{sansa\_stark}, M), \text{female}(M).$  Choice point
   $C_1: \text{parent}(\text{sansa\_stark}, \text{ned\_stark}).$ 
   $\theta_1: \{M = \text{ned\_stark}\}$ 
 $\mathcal{G}_2 = \text{female}(\text{ned\_stark}).$  dead-end: backtrack to  $\mathcal{G}_1$ 
 $\mathcal{G}_1 = \text{parent}(\text{sansa\_stark}, M), \text{female}(M).$  Second choice
   $C_1: \text{parent}(\text{sansa\_stark}, \text{catelyn\_tully}).$ 
   $\theta_1: \{M = \text{catelyn\_tully}\}$ 
 $\mathcal{G}_2 = \text{female}(\text{catelyn\_tully}).$ 
   $C_2: \text{female}(\text{catelyn\_tully}).$ 
 $\mathcal{G}_3 = \square.$ 

```

In program listing order, the first program clause C_0 with its head matching \mathcal{G}_0 is the definition clause for `mother`, and the obvious substitution θ_0 is to map variable X to name `sansa_stark` and variable Y to variable M . The new goal \mathcal{G}_1 is the resolvent consisting of the body of clause instance $\theta_0 C_0$ in place of the original sub-goal, resulting in `parent(sansa_stark, M), female(M)`.

The goal \mathcal{G}_1 is marked as a *choice point* because there is more than one pair (C_1, θ_1) of program clauses and substitutions that will match the first sub-goal `parent(sansa_stark, M)`. In program listing order, the first such program clause C_1 is the fact asserting that `sansa_stark` has parent `ned_stark`, with substitution θ_1 mapping variable M to name `ned_stark`.

We see that resolution with facts is *good* because it shortens the length of the list of sub-goals. The new goal \mathcal{G}_2 is now just the atomic formula `female(ned_stark)`, and it has no variables in it. But then we are in trouble: goal \mathcal{G}_2 is declared a *dead-end* because there are *no* program clauses with a head matching this sub-goal. Upon hitting a dead-end, PROLOG *backtracks* to the last *choice point*, which was at the goal \mathcal{G}_1 . So we erase back to \mathcal{G}_1 , and try again.

Excising out the dead-end, we get the *resolution refutation sequence* justifying the answer $M = \text{catelyn_tully}$.

```

 $\mathcal{G}_0 = \text{mother}(\text{sansa\_stark}, M).$ 
   $C_0: \text{mother}(X, Y) :- \text{parent}(X, Y), \text{female}(Y).$ 
   $\theta_0: \{X = \text{sansa\_stark}, Y = M\}$ 
 $\mathcal{G}_1 = \text{parent}(\text{sansa\_stark}, M), \text{female}(M).$ 
   $C_1: \text{parent}(\text{sansa\_stark}, \text{catelyn\_tully}).$ 
   $\theta_1: \{M = \text{catelyn\_tully}\}$ 
 $\mathcal{G}_2 = \text{female}(\text{catelyn\_tully}).$ 
   $C_2: \text{female}(\text{catelyn\_tully}).$ 
 $\mathcal{G}_3 = \square.$ 

```

The final resolution refutation sequence comes from trying the second choice option for the resolution of goal G_1 . This uses the program clause C_1 asserting the fact that `sansa_stark` has parent `catelyn_tully`, together with substitution θ_1 mapping M to name `catelyn_tully`. After resolution with a fact, the new goal G_2 is `female(catelyn_tully)`. Then the first matching program clause C_2 is `female(catelyn_tully)`, and the final resolvent is then the empty goal, \square . Within this resolution refutation sequence, we can see how the substitution θ_1 tells PROLOG the name of the existential witness $M = catelyn_tully$ in the original goal G_0 .

This is what happens when PROLOG answers affirmatively, and delivers a substitution of names for variables. When we enter a semi-colon key “ ; ” asking for more existential witnesses for the variables in the goal, PROLOG continues searching for more resolution refutation sequences, keeping track of paths in the search tree with branching generated by choice-points. Eventually, after some number of semi-colon requests for more, PROLOG will return the answer `false`. In that context, it means *there are no more*’ existential witnesses for the variables in the goal.

The next scenario is when PROLOG answers `false`. immediately, in response to the query of whether or not the existentially quantified goal is a logical consequence of the universally quantified program clauses in a logic program.

PROLOG will answer `false`. to query \mathcal{G}

`?- g1, g2, ..., gk.`

that is, the question whether or not $\mathcal{P} \models \mathcal{G}$,

if and only if a systematic *depth-first* search 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} .

Note the contrast: the refutation approach within the *proof tree method* has the added bonus that a `NO` answer brings with it a model which makes premises true and conclusion false. This is *not* the case for the resolution method used in PROLOG. A false or `NO` answer here does not bring with it the useful semantic information as provided by a proof tree with an open branch.

For an illustrative example, we again use our knowledge base of popular culture trivia with logic program \mathcal{P} : `game-of-thrones-1.pl`.

```
% /Users/jdavoren/prolog/Prolog-code/game-of-thrones-1.pl
compiled 0.00 sec, 113 clauses
1 ?- siblings(tyrion_lannister,lancel_lannister).
false.
2 ?- cousins(tyrion_lannister,lancel_lannister).
```

```
true .
3 ?- siblings(tyrion_lannister,S).
S = cersei_lannister ;
S = jaime_lannister ;
S = cersei_lannister ;
S = jaime_lannister ;
false.
4 ?-
```

This time, we ask about siblings. Our first question is: Are Tyrion Lannister and Lancel Lannister siblings? The PROLOG interpreter comes back with `false.` and immediately returns to the query prompt. The follow-on queries and their answers confirm that Tyrion Lannister and Lancel Lannister are cousins, and that the siblings of Tyrion Lannister are Cersei Lannister and Jaime Lannister.

In giving the answer `false.` to the question whether Tyrion Lannister and Lancel Lannister are siblings, the PROLOG search for a resolution refutation sequence proceeds as follows.

```
G0 = siblings(tyrion_lannister,lancel_lannister).
  C0: siblings(X,Y) :- parent(X,Z), parent(Y,Z), X \= Y.
  θ0: {X = tyrion_lannister, Y = lancel_lannister}
G1 = parent(tyrion_lannister,Z), parent(lancel_lannister,Z),
      tyrion_lannister \= lancel_lannister. Choice point
  C1: parent(tyrion_lannister,tywin_lannister).
  θ1: {Z = tywin_lannister}
G2 = parent(lancel_lannister,tywin_lannister),
      tyrion_lannister \= lancel_lannister.
dead-end: backtrack to G1
G1 = parent(tyrion_lannister,Z), parent(lancel_lannister,Z),
      tyrion_lannister \= lancel_lannister. Second choice
  C1: parent(tyrion_lannister,joanna_lannister).
  θ1: {Z = joanna_lannister}
G2 = parent(lancel_lannister,joanna_lannister),
      tyrion_lannister \= lancel_lannister.
dead-end: but nowhere to backtrack to! Searched failed.
```

PROLOG looks for the first program clause C_0 in the program with a head matching G_0 . It is the clause defining `siblings`, and the substitution maps the variables X and Y to the two names for Tyrion and Lancel.

The new goal G_1 has first sub-goal `parent(tyrion_lannister,Z)`. PROLOG finds two matches, one for each of two parents of Tyrion named in the knowledge base, so this is marked as a choice point. The first try for Z is Tyrion's father, Tywin Lannister.

The next, shorter, goal G_2 has as its first sub-goal to be processed the atom `parent(lancel_lannister,tywin_lannister)`. This goal is declared a dead-end because there are no matches in the program: the character Lancel Lannister has parents listed as `kevan_lannister` and `dorna_swift`, but not `tywin_lannister`. So PROLOG backtracks to the last choice point, which was goal G_1 .

The second choice for goal G_1 tries with Z as Tyrion's mother, Joanna Lannister. Then the new, shorter, goal G_2 has first sub-goal

`parent(lancel_lannister,joanna_lannister)`. This goal is also declared a dead-end because there are no matches in the program for the parent of `lancel_lannister`. At this point, there are no further choice points to explore, so the PROLOG interpreter declares a failed search and returns with the answer `false`.

In response to a query, the last remaining possibility is that PROLOG crashes and does not produce any answer at all. This will be the case when \mathcal{P} contains an *implication loop* and the loop is accessible via resolutions and unifications from the goal \mathcal{G} .

The following logic program gives a simple instance of this phenomenon.

```
/* loop-002.pl */
p(X) :- q(X).
q(a) :- p(a).
q(a).

% /Users/jdavoren/Prolog/Prolog-code/loop-002
compiled 0.00 sec, 4 clauses
1 ?- p(a).
ERROR: Out of local stack
Exception: (3,722,701) q(a) ?    creep
Exception: (3,722,699) q(a) ?
```

Let \mathcal{P} be the program listed in `loop-002.pl` and let \mathcal{G} be the goal consisting of just the atom `p(a)`. Then $\mathcal{P} \models \mathcal{G}$: the atom `p(a)` 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:

$$\begin{aligned}\mathcal{G}_0 &= p(a). \\ C_0 : p(X) &\text{ :- } q(X)., \quad \theta_0 : \{X = a\} \\ \mathcal{G}_1 &= q(a). \\ C_1 : q(a) &. \\ \mathcal{G}_2 &= \square\end{aligned}$$

The problem is that PROLOG gets stuck in the loop flipping between the instance `p(a) :- q(a).` of clause C_0 and the converse program clause `q(a) :- p(a).`, and it never gets to find the fact `q(a)`. So the resolution refutation sequence exists, but the PROLOG interpreter is not able to find it, and so is unable to return any answer at all to the query `?- p(a)`. Thus 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.

Now it is your turn to pretend to be a PROLOG interpreter. For this exercise, and ones like them in the quizzes online, you are *not* to use PROLOG software to answer the question. Rather, the purpose is for you to think through the resolution steps taken by a PROLOG interpreter in answering queries.

```
/* exercise-002.pl */

s(a).
s(c).
s(e).
p(a,b).
p(b,c).
p(c,d).
p(d,e).
q(X,Y) :- p(X,Y).
q(X,Y) :- p(X,Z), p(Z,Y).
r(X,Y) :- s(X), s(Y), q(X,Y).
```

Which option best describes how PROLOG will answer the query:

?- **r(A1,A2)**.

which asks “Are there objects A1 and A2 that are r-related?”

- (a)** A1 = c, A2 = e ; **false**.
- (b)** **false**.
- (c)** A1 = a, A2 = c ; A1 = c, A2 = e ; **false**.
- (d)** **true** .
- (e)** Does not answer.

The correct answer is option **(c)**. PROLOG will first find the pair of A1 as a and A2 as c, because of facts s(a), s(c), p(a,b), and p(b,c). Then it will get the second pair with A1 as c and A2 as e, because of facts s(c), s(e), p(c,d), and p(d,e).

6.4.4 | WHY RESOLUTION WORKS: CLAUSAL FORM

In clausal form, **program clauses** include *exactly one* positive literal (the head b), and the rest are negative literals. Written in clausal form:

$$C : (b \vee \neg a_1 \vee \neg a_2 \vee \dots \vee \neg a)$$

In clausal form, negated goal formulas $\neg D$ have *zero* positive literals, as all literals are negative. Written in clausal form:

$$\neg D : (\neg g_1 \vee \neg g_2 \vee \dots \vee \neg g_k)$$

When PROLOG attempts to answer the query $?- \mathcal{G}$ after loading logic program \mathcal{P} , whose clauses are assumed *true*, the *refutation* approach supposes goal \mathcal{G} is *false* and attempts to derive the empty goal \square .

If goal \mathcal{G} in negated clausal form is $(\sim p \vee \sim g_1 \vee \sim g_2 \vee \dots \vee \sim g_k)$ and program clause C is a *conditional rule* $(p \vee \sim a_1 \vee \sim a_2 \vee \dots \vee \sim a_m)$, and substitution θ is such that $\sim \theta p$ and θp match, then the *resolvent* \mathcal{G}' of \mathcal{G} with C under θ is goal

$$(\sim \theta a_1 \vee \sim \theta a_2 \vee \dots \vee \sim \theta a_m \vee \sim \theta g_1 \vee \sim \theta g_2 \vee \dots \vee \sim \theta g_k)$$

obtained by taking the disjunction of $\theta \mathcal{G}$ with θC , and cancelling the complementary literals in $\sim \theta p \vee \theta p$.

If goal \mathcal{G} in negated clausal form is $(\sim p \vee \sim g_1 \vee \sim g_2 \vee \dots \vee \sim g_k)$ and program clause C is a *fact* p , and substitution θ is such that $\sim \theta p$ and θp match, then the *resolvent* \mathcal{G}' of \mathcal{G} with C under θ is goal $(\sim \theta g_1 \vee \sim \theta g_2 \vee \dots \vee \sim \theta g_k)$, obtained by taking the disjunction of $\theta \mathcal{G}$ with θC , and cancelling the complementary literals in $\sim \theta p \vee \theta p$. If $k = 0$ then \mathcal{G}' is \square .

6.4.5 | CORRECTNESS OF PREDICATE PROLOG

We finish off with a summary of the soundness or correctness of predicate PROLOG.

Given a logic program \mathcal{P} loaded into PROLOG together with a goal \mathcal{G} , all of whose sub-goals have predicates occurring as heads of clauses in \mathcal{P} :

- If PROLOG answers with a substitution for variables in the goal, or with *true* to the query $?- \mathcal{G}$ then it is the case that $\mathcal{P} \models \mathcal{G}$.
- If PROLOG answers *false* to the query $?- \mathcal{G}$ then $\mathcal{P} \cup \{(\forall \text{var}(D)) \sim D\}$ is satisfiable in the Herbrand model of \mathcal{P} , where D is the conjunction of \star -atoms in \mathcal{G} , so $\mathcal{P} \not\models \mathcal{G}$.
- **But ... incompleteness:** PROLOG can fail to answer when it really is the case that $\mathcal{P} \models \mathcal{G}$.

This brings us to the end of this chapter on predicate logic programming in PROLOG. We hope you have enjoyed this foray into computer science via logic, and get stuck into more examples within the online practice and graded quizzes.

FURTHER LEARNING

- Ulle Endriss, *Lecture Notes: An Introduction to Prolog Programming*, Institute for Logic, Language and Computation, the University

of Amsterdam, 2007; <http://staff.science.uva.nl/~ulle/teaching/prolog/prolog.pdf>

- *Learn Prolog Now!* Online tutorials and manual.
<http://www.learnprolognow.org>
- *Prolog Tutorial Program*, CS1104, Department of Computer Science, Virginia Tech (Virginia Polytechnic Institute); <http://courses.cs.vt.edu/~cs1104/TowerOfBabel/Prolog/prolog.samples.html>

SAMPLE PROLOG PROGRAM

```
/* game-of-thrones-1.pl */

/* general family relationships */
/* usage: parent(X,Y) means child X has parent Y, or a parent of X is Y. */

mother(X,Y)  :- parent(X,Y), female(Y).
father(X,Y)  :- parent(X,Y), male(Y).
siblings(X,Y)  :- parent(X,Z), parent(Y,Z), X \= Y.
brother_of(X,Y)  :- siblings(X,Y), male(Y).
sister_of(X,Y)  :- siblings(X,Y), female(Y).
grandparent(X,Y)  :- parent(X,Z), parent(Z,Y).
grandmother(X,Y)  :- grandparent(X,Y), female(Y).
grandfather(X,Y)  :- grandparent(X,Y), male(Y).
cousins(X,Y)  :- parent(X,W), parent(W,Z), parent(Y,V), parent(V,Z), X \= Y, W \= V.
aunt(X,Y)  :- parent(X,Z), sister_of(Z,Y).
uncle(X,Y)  :- parent(X,Z), brother_of(Z,Y).
ancestor(X,Y)  :- parent(X,Y).
ancestor(X,Y)  :- parent(X,Z), ancestor(Z,Y).

sibling_parents(Y,Z)  :- parent(Y,W), parent(Z,W), parent(X,Y), parent(X,Z), Y \= Z.
sibling_parents(X,Y,Z)  :- parent(Y,W), parent(Z,W), parent(X,Y), parent(X,Z), Y \= Z.

/* facts from the family tree of the Stark family */
parent(rickard_stark,edwyle_stark).
parent(brandon_stark_snr,rickard_stark).
parent(ned_stark,rickard_stark). /* ned = eddard */
parent(benjen_stark,rickard_stark).
parent(lyanna_stark,rickard_stark).
parent(jon_snow,ned_stark).
parent(rob_stark,ned_stark).
parent(sansa_stark,ned_stark).
parent(arya_stark,ned_stark).
parent(bran_stark,ned_stark). /* bran = brandon jnr */
parent(rickon_stark,ned_stark).
parent(rob_stark,catelyn_tully).
parent(sansa_stark,catelyn_tully).
parent(arya_stark,catelyn_tully).
parent(bran_stark,catelyn_tully).
parent(rickon_stark,catelyn_tully).
```

```

/* facts from the family tree of the Lannister family */
parent(tywin_lannister,tytos_lannister).
parent(kevan_lannister,tytos_lannister).
parent(tygett_lannister,tytos_lannister).
parent(gerion_lannister,tytos_lannister).
parent(genna_lannister,tytos_lannister).
parent(cersei_lannister,tywin_lannister).
parent(jaime_lannister,tywin_lannister).
parent(tyrion_lannister,tywin_lannister).
parent(cersei_lannister,joanna_lannister).
parent(jaime_lannister,joanna_lannister).
parent(tyrion_lannister,joanna_lannister).
parent(joffrey_baratheon,cersei_lannister).
parent(myrcella_baratheon,cersei_lannister).
parent(tommen_baratheon,cersei_lannister).
parent(joffrey_baratheon,jaime_lannister).
parent(myrcella_baratheon,jaime_lannister).
parent(tommen_baratheon,jaime_lannister).
parent(lancel_lannister,kevan_lannister).
parent(willem_lannister,kevan_lannister).
parent(martyn_lannister,kevan_lannister).
parent(janei_lannister,kevan_lannister).
parent(lancel_lannister,dorna_swift).
parent(willem_lannister,dorna_swift).
parent(martyn_lannister,dorna_swift).
parent(janei_lannister,dorna_swift).
parent(tyrekk_lannister,tygett_lannister).
parent(tyrekk_lannister,darlessa_marbrand).
parent(joy_hill,gerion_lannister).
parent(cleos_frey,genna_lannister).
parent(lyonel_frey,genna_lannister).
parent(tion_frey,genna_lannister).
parent(red_walder_frey,genna_lannister).
parent(cleos_frey,emmon_frey).
parent(lyonel_frey,emmon_frey).
parent(tion_frey,emmon_frey).
parent(red_walder_frey,emmon_frey).
parent(tywin_frey,cleos_frey).
parent(willem_frey,cleos_frey).
parent(tywin_frey,jeyne_darry).
parent(willem_frey,jeyne_darry).

/* females among Stark family */
female(lyanna_stark).
female(catelyn_tully).
female(sansa_stark).
female(arya_stark).

/* females among Lannister family */
female(joanna_lannister).
female(gerion_lannister).

```

```

female(genna_lannister).
female(cersei_lannister).
female(myrcella_baratheon).
female(dorna_swift).
female(janei_lannister).
female(darlessa_marbrand).
female(joy_hill).
female(jeyne_darry).

/* males among Stark family */
male(edwyle_stark).
male(rickard_stark).
male(brandon_stark_snr).
male(ned_stark).
male(benjen_stark).
male(jon_snow).
male(rob_stark).
male(bran_stark).
male(rickon_stark).

/* males among Lannister family */
male(tytos_lannister).
male(tywin_lannister).
male(kevan_lannister).
male(tygett_lannister).
male(jaime_lannister).
male(tyrion_lannister).
male(joffrey_baratheon).
male(tommen_baratheon).
male(lancel_lannister).
male(willem_lannister).
male(martyn_lannister).
male(tyrek_lannister).
male(emmon_frey).
male(cleos_frey).
male(lyonel_frey).
male(tion_frey).
male(red_walder_frey).
male(tywin_frey).
male(willem_frey).

```


DEFINITE DESCRIPTIONS AND FREE LOGIC



WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- The difference between *existence* and *subsistence* in Meinong's philosophy of objects.
- The difference between a *name* and a *description*.
- The difference between *definite* and *indefinite* descriptions.
- The quantificational syntax of definite descriptions.
- The difference between *inner* and *outer* negations when applied to definite descriptions.
- Russell's analysis of definite descriptions in the language of first order logic.
- The treatment of existence and uniqueness assumptions as pre-suppositions of a definite description claim, rather than as a part of the claim.
- The treatment of natural language names as *quantifiers*.
- The syntax of *free logic*, with the addition of an *existence predicate* E!
- Models for (positive) free logic including an inner domain, the extenstion of E!, over which the quantifiers range.
- Tree proofs for (positive) free logic.

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.

SKILLS

These are the skills you will learn to demonstrate.

- Formalising natural language sentences into a formal language distinguishing the scopes of definite descriptions (the $(Ix)(Fx, Gx)$ notation) and using this language to distinguish the different readings of inner and outer negations and the possibilities for positioning other logical connectives; recognising the difference this makes in the interpretation of those sentences.
- Translation of definite description vocabulary into the language of first order predicate logic with identity, using Russell's analysis of definite description, and use of this analysis to distinguish the different truth conditions for inner and outer negations.

- Evaluation of arguments and analysis of statements in the language of positive free logic, including the construction of tree proofs and the interpretation of formulas in models of positive free logic.

The assumption that every proposition is either true or false is sometimes called the assumption of *bivalence*.

Philosophy and *logic* are connected in many different ways. One connection is to look at the *philosophy of logic*—to critically examine the assumptions involved in a particular logical technique, to explore alternatives, and to see whether those assumptions hold water or whether they should be revised in the face of evidence. In *Logic: Language and Information 1* we did this by looking at the assumption that every proposition is either true or false—an assumption fundamental to the two-valued models used in classical propositional or predicate logic. We looked at this assumption alongside the manifest *vagueness* which fills the world around us, and we asked ourselves whether this assumption could be maintained in the presence of vagueness, or whether it should be rejected in favour of a different understanding of truth values. This view of the relationship between philosophy and logic can be repeated in topics beyond bivalence—as philosophers examine the foundations of the entire logical enterprise, and ask difficult questions about whether the guiding assumptions are correct or could be altered in one way or another. *Philosophy of logic* is a fruitful exercise in many different ways.

However, we will *not* do philosophy of logic in this way here—at least, not in the first part of this chapter. Instead, we will give a very different example of how philosophy and logic can relate. For the first part of this chapter, we will instead show how tools from logic can shed light on difficult debates in *philosophy*. In the next sections, we will see how first order predicate logic can be used to answer questions in *metaphysics*, and clarify certain puzzles about the nature of *existence*.

7.1 | WHAT DOES EXIST, AND WHAT DOESN'T?

Philosophers study many things—one important part of philosophy is the study of what *exists*, and the *nature of existence*. This area of study is variously called *ontology* (which centres on the classification and analysis of different categories of existence, substances, qualities, properties, tropes, relations, etc.) and *metaphysics* (which is harder to define sharply, but which is centred on the attempt to describe ultimate reality at its fundamental level). In discussions in metaphysics and ontology, philosophers are prone to make many different claims about what *exists* and what *doesn't*. When we venture into this area, it seems that we are prone to come perilously close to contradicting ourselves.

See Peter van Inwagen's entry "Metaphysics" in the *Stanford Encyclopedia of Philosophy*, at <http://plato.stanford.edu/entries/metaphysics/> for an attempt to explain why metaphysics as it is currently practiced, is very difficult to define.



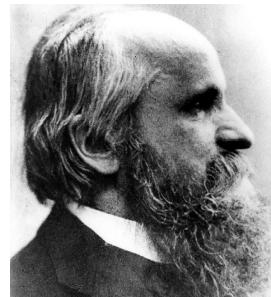
When we make claims of non-existence, these can take a number of forms, depending on the kind of thing we're taking to fail to

exist. Here is a straightforward example: In the 17th and 18th Centuries, *phlogiston* was thought to be a substance present within objects that could burn—and which was released during combustion. After experimental results could be explained in other ways, by the end of the 18th Century, scientists eventually rejected this theory of combustion. As a result, we no longer think that *phlogiston exists*. In this case, the non-existence claim is relatively straightforward. There is no *phlogiston*. This has the form $\sim(\exists x)Px$, where ‘P’ is a one-place predicate for ‘*phlogiston*.’ It turns out that there is nothing that has the property of being *phlogiston*—of being the substance that is present within all combustible objects, and which is released when they burn. If there is no such object, then the nonexistence claim is straightforward: it is a negatively existentially quantified sentence $\sim(\exists x)Px$, and this is straightforwardly true or false in an interpretation depending on whether there are objects in the interpretation of the predicate P.

So far, so good. But not all non-existence claims have this form. You and I could be talking about Socrates, and you might ask me: did he *really* exist? Now, if we were to say “no, he didn’t,” then what are we saying is the logical structure of *this* claim? If “Socrates” features as a name, then $\sim E!s$ might work, where “E!” is a special predicate for *existence*, but if then how would this “E!” predicate work? If s is interpreted as an element of our domain (as it is in models of predicate logic) then members of this domain exist equally and on a par with each other. It makes little sense for some to exist while others do not. But how, then, are we to understand claims of the form “Socrates didn’t exist” and other claims like it?

Such questions came to the forefront of the attention of the Austrian philosopher, Alexius Meinong. He was most influential for his theory of *objects*, and *intentionality*. For Meinong, and the other phenomenologists, thought is to be distinguished by its *direction of fit*—we can think of things, and sometimes those things we contemplate *exist* externally to our thoughts, and sometimes they don’t [6]. (We don’t always get things *right* when we make an inventory of the world.) Thought, when it works, reaches out and touches the world as it is. If the thought fails to refer, it can still make sense, for it *still* is directed outside oneself to an object—but in this case it is an *object of thought*. All thoughts have objects—some exist in the external world, while others merely *subsist*. This was the case for the mythical *Golden Mountain* (a mountain made of solid gold). There is no Golden Mountain, so the Golden Mountain does not exist. However, we can still think about the Golden Mountain, so it is an object of our thoughts. It *subsists* without *existing*.

This is an ingenious attempt to begin make sense of claims of nonexistence, while holding onto the idea that thoughts have to refer to an object outside themselves for them to be thoughts at all. At this relatively superficial level, there seem to be no contradictions. However, troubling questions come very close to the surface when we think more deeply about the view. For example: how many objects are there that



Alexius Meinong (1853–1920)



The Golden Mountain?

What then, of claims of the non-existence of *phlogiston*? Does this mean that there is nothing that has the property of being *phlogiston*? Or does it mean that there is nothing that is *phlogiston* that *exists*? Does it mean that all *phlogiston* merely subsists and does not exist?

...at least in the case where it is 5cm across. I suppose an object can be a circle and a square if it is 0cm across—and it's then a single *point*. This is why we have the restriction to a non-zero size.

This paradox launched an industry of those wanting to weaken the so-called ‘characterisation postulate’ (the claim that “the F has property F,” which is central to much discussion of these cases). The simple case of inconsistent descriptions is enough to show that descriptions do not always work as one might hope for. There are a number of proposals for how to weaken the characterisation postulate to avoid.

merely subsist? Is there anything that *doesn't even subsist*? The Golden Mountain does not exist because the world doesn't supply any mountains made of gold. But we can think about such a mountain, we can head off in search of it, and we can be disappointed that we did not find it. Similarly, we can ask ourselves about the 5cm round square—this is a plane figure that is simultaneously a circle 5cm across, and a square 5cm across. It has a circular circumference with one uniform curved edge and no straight lines, and four sides which are straight lines—with no curve at all. Such a figure is inconsistent. It both has and does not have straight sides. It both has and does not have a curved perimeter. The 5cm round square cannot exist. Its nonexistence is much more severe than the nonexistence of the Golden Mountain. While the Golden Mountain *could have* existed had the geology of the earth gone differently, and while the Golden Mountain could yet exist (were we to put our minds to it, and acquire enough gold to construct such a thing), the 5cm round square *cannot* exist. It is logically impossible, given that its characterising properties (being wholly round, and being wholly a square) are inconsistent with each other. Now, ask yourself this: Why does the Golden Mountain merely subsist without existing? Presumably it is because it is *golden*, and it is a *mountain*, and there is nothing that fits this description existing in the world of phenomena to hand. The Golden Mountain is indeed golden, and it is indeed a mountain, but it does not exist. It merely subsists. Now consider the 5cm round square? Why does it not exist? Is it because it is round and 5cm and a square? 5cm squares are (logically), not round—they're square. So the 5cm round square is both *round* and it is *not round*. If the Golded Mountain failed to exist *because* it was a golden mountain—and there are no such things in the world—does the same go for the 5cm round square? Does this merely subsist and not exist because there are no round squares in reality? If this is so, then it must be because the round square is *round* (in virtue of being circular), and it is *not round* (in virtue of being square, and 5cm in size). But if the round square is both round and not round, then we have *contradicted* ourselves. We have said that Ra (object a is round) and $\sim Ra$ (object a is not round), and this is a contradiction, whether the object in question merely *subsists* or *exists*.

It was for reasons of puzzles like these—as well as the overwhelming extravagance of a domain containing *all* of those non-existent things—that Bertrand Russell attempted to show how we could give an account of existence and nonexistence *without* dividing the world between existing and subsisting objects. We could describe a world in which the only things with any sort of being are those that *exist*, and we give another account of what it is to say that something doesn't exist. It's to this account that we'll now turn, not immediately, but after a first stop at the kind of *syntax* available to us when we make claims to existence or non-existence.

7.1.1 | WHAT IS THE FORM OF THESE STATEMENTS?

Meinong was working in a period where the development of first order predicate logic was well and truly in its infancy. The dominant tradition in logic—Aristotle’s logic—took the following statements to have very similar logical form:

- Barack Obama plays basketball.
- The US President plays basketball.

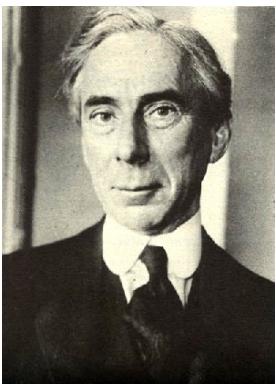
In each case, for an Aristotelean approach, the form of the two statements is “S is P”—where the *subject* is given (in these cases, either Obama, or the US President) and then has some property is ascribed of that subject (in these cases, *playing basketball*). In both cases, we are presented with an subject to describe (and here, it is the same object—Barack Obama, the president of the United States), and we describe it in the same way (as playing basketball). It is easy to see why these two claims are understood as having the same logical structure. Consider, however, the interaction between this simple Subject/Predicate sentence and other logical connectives, such as negation.

- Plato is not an empiricist.
- The teacher of Aristotle is not an empiricist.
- It’s not true that the teacher of Aristotle is an empiricist.

These *three* statements seem to me have different logical structures, even though they share a great deal in common. Plato is the teacher of Aristotle, and so, the first two claims are very closely related. One picks out Plato by name, the other picks him out by the description: the teacher of Aristotle. The third claim differs from the second merely in the position of the negation operator. The second claim takes the description to pick out the teacher of Aristotle, and then makes the claim that the bearer of this description is not an empiricist. The last claim does something similar, but now the negation has a wider scope: this claim says that it’s not true that the teacher of Aristotle is an empiricist.

Is there any difference between these last two claims? Perhaps you might find it difficult to distinguish them. However, there are ways you can see these sentences as making very different claims, corresponding to the different positions of the negation in the claim. We can see a hint of this when we consider how to formalise each claim in the language of first order predicate logic. The claim that Plato is not an empiricist has the form $\sim E a$, where ‘a’ is a name, and ‘E’ is a predicate. The claim that the teacher of Aristotle is not an empiricist have a different structure, as ‘the teacher of Aristotle’ is not itself a name. It involves the name ‘Aristotle’ and the binary relation ‘...is a teacher of ...’ but it combines these in such a way as to get something that acts somewhat like a name acts in the sentence. And when it does this, it must do this in such a way as to make open the possibility that the *negation* in the English sentence could appear in different places in the

underlying logical structure. This idea goes back at least to the English philosopher, Bertrand Russell, in his analysis of *definite descriptions*. To take a look at Russell's analysis, we will use another example, from his own writing on definite descriptions [9].



Bertrand Russell (1872-1970)

DEFINITION: A *description* is a term used to *pick out* an object or objects, using predicates. Descriptions can be “indefinite” when they do not purport to pick out an object uniquely (example: “*a* frog” or “*some* philosopher”) or *definite* when they do purport to pick out a single item (example: “*the* teacher of Aristotle”).



Louis XVI of France
Reign: May 18, 1804–April 11, 1814

7.1.2 | THE PRESENT KING OF FRANCE

When Russell wrote on the logical behaviour of descriptive terms like “the teacher of Aristotle,” it was the beginning of the 20th Century. France was a republic, which meant that it had no king. France's last king was Louis XVI, who reigned to 1814. While France had no king, the description “the present King of France” failed to pick someone out. So, if someone were to say

- The present King of France is bald.

They'd be failing to say something true. It would be untrue, because France has no King, and so, there's no-one to be picked out and truly described as bald. As a result, it would be correct to say this:

- It's not true that the present King of France is bald.

because indeed, it is not true to make the claim that the present King of France is bald. This is fair enough, but it's a very different thing to say to make the claim that

- The present King of France is *not* bald.

This is to say that there is a present king of France, and that he's bald. Under the present circumstances—where France has no King, this is *false*. In fact, it would be correct to say this:

- It's not true that the present King of France is not bald.

This seems very very strange, if all the logical structure we can see is in the shape of Aristotle's logic with subjects and predicates. To sum up our findings so far, we want to say that

1. The present King of France is bald. —**FALSE**
2. The present King of France is *not* bald. —**FALSE**
3. It's not true that the present King of France is bald. —**TRUE**
4. It's not true that the present King of France is not bald. —**TRUE**

If all we can see in these statements is the form K is B and its negation K is not B , then it seems that we want to make the following assignments of truth and falsity:

1. K is B —**FALSE**
2. K is not B —**FALSE**
3. K is not B —**TRUE**
4. K is B —**TRUE**

But this *cannot* be right—these assignments of truth values are inconsistent. So, what are we to do? The first step will be to reject the analysis of the structure of these sentences in Aristotle's logic. This cannot be the form of the sentences—formalising them in something like the language of first order predicate logic will give us more scope to position the negations in different places, in order to expose the difference between “The present King of France is *not* bald” (**FALSE**) and “It's not true that the present King of France is bald” (**TRUE**).

7.1.3 | RUSSELL'S REJECTION OF SUBJECT–PREDICATE FORM

For Bertrand Russell, claims like “the F is G ” *don't* have a subject-predicate form [9]. Constructions like “the $F\dots$ ” act more like a *quantifier* than a *name*. Given two one-place predicates F and G , then, we have a number of different ways to combine them with quantifiers:

Some F is a G	$(\exists x)(Fx \ \& \ Gx)$
All Fs are Gs	$(\forall x)(Fx \supset Gx)$
No Fs are Gs	$\sim(\exists x)(Fx \ \& \ Gx)$
The F is a G	???

A *definite description* combines two predicates to make a proposition, just as the other binary quantifiers do. We don't yet have a way to represent “the F is a G ” in terms of the language of first order predicate logic, but we could at least combine the two predicates Fx and Gx and tag the formula with the variable x , which is *bound* in the expression, in the same sort of way as in $(\exists x)(Fx \ \& \ Gx)$, or $(\forall x)(Fx \supset Gx)$. So, we will write

$$(Ix)(Fx, Gx)$$

$(\forall x)$, $(\exists y)$, etc. are *unary* quantifiers. They take a sentence and modify it by *generalising* it. The natural language “all” and “some” are *binary* quantifiers. They take two predicates and join them to make a sentence—*all frogs are green* $(\forall x)(Fx \supset Gx)$ or *some frogs are poisonous* $(\exists y)(Fy \ \& \ Py)$. The definite description is like this: *the frog is hopping*.

to stand for “the F ” where the “I” indicates that the definite description “the $F\dots$ ” picks out an *individual*. With *just* this much decided—without even deciding what it takes for $(Ix)(Fx, Gx)$ to be *true* in a model—we can already draw some distinctions which will be able to shed light on the interaction between descriptions and negation.

7.1.4 | INNER AND OUTER NEGATION

Given this syntax, we have the means to clarify the two different ways negation can interact with a description. If Kx formalises “ x is a King

of France” and Bx formalises “ x is bald,” then “the King of France is not bald” comes out with the following form:

- The King of France is not bald.
- $(Ix)(Kx, \sim Bx)$

We can call this the *inner negation* of $(Ix)(Kx, Bx)$ because the negation is inserted *inside* the description construction, to attach itself to the inner predication: the “is bald” is transformed into “is not bald,” and the description itself is kept unchanged. We go from “the King of France is bald” to “the King of France is not bald”—the inner predication is changed, but the outer structure remains the same.

On the other hand, the other negation, which simply denies the claim “the King of France” places the negation in another location:

- It’s not true that the King of France is bald.
- $\sim(Ix)(Kx, Bx)$

In this case, we have the *outer negation* of $(Ix)(Kx, Bx)$, which is negated at the outermost point of the formula. In this case, the entire formula $(Ix)(Kx, Bx)$ is denied, without this being necessarily understood as the predication of $\sim Bx$ to some object, the King of France. Perhaps the claim $(Ix)(Kx, Bx)$ is false not because the King of France is not bald, but because of some other reason. All this will be allowed for $\sim(Ix)(Kx, Bx)$ to be true. In the next section, we will look at Bertrand Russell’s *account* of when $(Ix)(Fx, Gx)$ should hold. Before that, however, we can look more deeply into what we can do with this syntax for definite descriptions.

7.1.5 | SOME TRANSLATIONS

Simple binary description statements with single predicates are straightforward to translate into our language. A claim such as

The philosopher is happy.

comes out as straightforward:

$$(Ix)(Px, Hx)$$

A slightly more complex description, in which the descriptive phrase is compound, like

The happy philosopher is female.

is also straightforwardly treated. If Hx is “ x is happy” and Fx is “ x is female,” and in addition Px is “ x is a philosopher,” then the formalisation proceeds as follows:

$$(Ix)(Hx \ \& \ Px, Fx)$$

In this case, it wouldn’t be enough to use the single predicate letters without the variables. Marking the variables explicitly indicates that

we'd like the same bearer to satisfy both properties. The need for explicitly marking the variables becomes even clearer when you see a yet more complex statement, with nested description terms. We might say, for example:

The philosopher is shorter than the engineer.

In this case, we have two descriptions, and it is best to proceed with the translation in a step-by-step fashion. First, we can say this:

$(Ix)(Px, x \text{ is shorter than the engineer})$

In which we formalise “The philosopher (x) is such that ... ” and we fill the blank in with “ x is shorter than the engineer”—and this is the next thing to translate. A statement like this, which has the description in a subordinate position, needs to be restructured in order for it to be formalised in our language. Instead of saying ‘ x is shorter than the engineer’ we can say ‘the engineer is such that x is shorter than them.’ This has the same effect, but is much more straightforward to formalise.

‘ x is shorter than the engineer’ contains the description “the engineer” in second position in the predication.

$(Ix)(Px, \text{the engineer is such that } x \text{ is shorter than them})$

For now we can use a new variable for ‘the engineer,’ in $(Iy)(Ey, \dots)$ and then in the remaining space we say ‘ x is shorter than y .’ So the result is straightforward:

$(Ix)(Px, (Iy)(Ey, Sxy))$

where Px is “ x is a philosopher,” Ey is “ y is an engineer” and Sxy is “ x is shorter than y .”

FOR YOU: Which of these formulas is a good formalisation for the sentence “the philosopher and the engineer are happier than the mathematician”, where Px is “ x is a philosopher”, Ex is “ x is an engineer,” Mx is “ x is a mathematician” and Hxy is “ x is happier than y .”

- (a)** $(Ix)(Px \& Ex, (Iy)(My, Hxy))$
- (b)** $(\forall x)((Px \vee Ex) \supset (Iy)(My, Hxy))$
- (c)** $(Ix)(Px, (Iy)(My, Hxy)) \& (Ix)(Ex, (Iy)(My, Hxy))$
- (d)** $(Ix)(Mx, (Iy)(Py, Hxy) \& (Iy)(Ey, Hxy))$

are happier than the mathematician. So, (c) is the only correct answer says, which is the reverse—that the philosopher and engineer and happier than the engineer. This is not what the original statement says that the mathematician is both happier than the philosopher something with similar structure—but with the opposite outcome. (d) is a way to say the engineer is happier than the mathematician. (d) is one way to answer the question, by saying that the philosopher is happier than the mathematician and that isn't correct either. (c) is one way to answer the question, by philosopher or an engineer is happier than the mathematician—isn't what we want to say. (b) says that anything that is either a philosopher or an engineer is such that it's happier than the mathematician—that an engineer is such that it's happier than the mathematician—that

Answer: (a) says that the thing which is both a philosopher and

This should be enough background for you to gain a grasp of the *syntax* of definite descriptions, and to recognise the possibilities for distinguishing scopes for negation. The previously inconsistent attempt to assign truth values to statements of the form ‘K is B’ and ‘K is not B’ can now be understood in the following way:

- The present King of France is bald.
 - ✗ K is B
 - ✓ (Ix)(Kx, Bx) —**FALSE**
- The present King of France is *not* bald.
 - ✗ K is not B
 - ✓ (Ix)(Kx, ~Bx) —**FALSE**
- It's not true that the present King of France is bald.
 - ✗ K is not B
 - ✓ ~(Ix)(Kx, Bx) —**TRUE**
- It's not true that the present King of France is not bald.
 - ✗ K is B
 - ✓ ~(Ix)(Kx, ~Bx) —**TRUE**

Now we have four different forms, where the two that purport to predicate a property of the King of France—(Ix)(Kx, Bx), (Ix)(Kx, ~Bx)—can both be false, and the remaining two, which deny that the property can be truly predicated—~(Ix)(Kx, Bx), ~(Ix)(Kx, ~Bx)—can be false.

Or so we hope.

All of this presumes that we have some coherent way of assigning truth values to these definite description expressions. At the very least we have ways of distinguishing the different scopes of negation, and we have four different formulas to which to assign truth values—but is there a way to understand the *meanings* of those expressions, so that two

come out as true and the other two come out as false? This requires an understanding of what it takes for such a sentence to be *true*, and it's to this topic that we will now turn.

7.2 | RUSSELL'S ANALYSIS OF DEFINITE DESCRIPTIONS

Bertrand Russell provided not only an analysis of the structure of definite description claims, he also provided a theory of the conditions under which such a claim is *true*. According to what we now understand as RUSSELL'S ANALYSIS OF DEFINITE DESCRIPTIONS, a definite description $(Ix)(Fx, Gx)$ makes *three* claims:

- There's one thing with property F.
- That thing the *only* thing with property F.
- That thing also has property G.

This means that three distinct criteria need to be satisfied for a definite description claim to come out as *true*, and, correspondingly, there are three different ways for a definite description claim to *fail* to be true—to come out as false. For $(Ix)(Fx, Gx)$ to be false, one of three things could happen: either there is nothing with property F, or there is more than one thing with that property, or (if there is a unique thing with property F) that thing could fail to have property G. This will become very important when it comes to understanding the *logic* of inner and outer negation of definite description claims.

7.2.1 | ENCODING DEFINITE DESCRIPTIONS IN FIRST ORDER LOGIC

Before we get to studying the logic of definite descriptions, we can notice that we have all of the tools at hand to use this analysis definite descriptions *within* the language of first order predicate logic with identity as we already have it. We already know how to say that something has property F, it's the only thing with property F and it has property G. It is already a sentence in the language of first order predicate logic with identity.

Notice that this expression is not literally a conjunction with three conjuncts. It is an existentially quantified conjunction. In this chapter we will not care about the bracketing of the conjunction. If you felt the need to be precise and avoid three-way conjunctions, bracket the first two conjuncts together like so: $(\exists x)([Fx \ \& \ (\forall y)(Fy \supset y = x)]) \ \& \ Gx$ —here indicated with square brackets for clarity.

RUSSELL'S ANALYSIS OF DEFINITE DESCRIPTIONS: $(Ix)(Fx, Gx)$ can be defined as

$$(\exists x)(Fx \ \& \ (\forall y)(Fy \supset y = x) \ \& \ Gx)$$

in the language of first order predicate logic with identity.

This means that we can write out our example definite description sentences from the previous section in the language of first order predicate logic.

EXAMPLE: Rendering definite description formulas in the language of first order predicate logic.

- *The philosopher is happy.*
 - $(Ix)(Px, Hx)$
 - becomes $(\exists x)(Px \ \& \ (\forall y)(Py \supset y = x) \ \& \ Hx)$
- *The happy philosopher is female.*
 - $(Iy)(Hy \ \& \ Py, Fy)$
 - becomes $(\exists y)((Hy \ \& \ Py) \ \& \ (\forall z)((Hy \ \& \ Py) \supset z = y) \ \& \ Fy)$
- *The philosopher is shorter than the engineer.*
 - $(Ix)(Px, (Iy)(Ey, Sxy))$
 - becomes $(\exists x)(Px \ \& \ (\forall z)(Pz \supset x = z) \ \& \ (\exists y)(Ey \ \& \ (\forall z)(Ez \supset y = z) \ \& \ Sxy))$

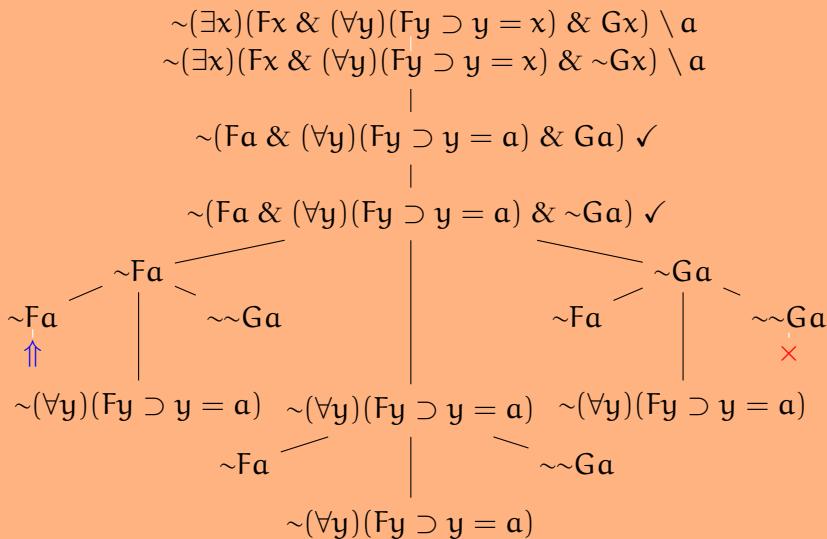
Notice that we may use different variables in the specification of the definite description. In our formalisation of “the happy philosopher is female” we used y instead of x . Just be careful to use a *distinct* variable in the inner quantification—here the ‘ $(\forall z)$ ’—used to specify the uniqueness condition.)

Notice too, that in the last example we have two definite descriptions, one inside the scope of the other. It was important to use different variables for the existential quantifiers in each description, as one is inside the scope of the other, and we want to distinguish the philosopher from the engineer when we come to Sxy . But there was no need to use different variables for the uniqueness universal quantifiers. I used the same variable (z) for the both uniqueness quantifiers, as the scopes for these do not overlap, and they are doing the same ‘job’ in each case.

7.2.2 | THE LOGIC OF DEFINITE DESCRIPTIONS

Now that we have the sentences in the language of first order predicate logic, we can use the tools of first order predicate logic to analyse these sentences and the logical relationships between them. For example, we can test the argument from $\sim(Ix)(Fx, Gx)$ to $(Ix)(Fx, \sim Gx)$, and show that it is invalid.

EXAMPLE: $\sim(Ix)(Fx, Gx) \not\vdash (Ix)(Fx, \sim Gx)$ — here is a tree with a complete open branch:



In this tree, the starting formulas $\sim(\exists x)(Fx \& (\forall y)(Fy \supset y = x) \& Gx)$ and $\sim(\exists x)(Fx \& (\forall y)(Fy \supset y = x) \& \sim Gx)$ are both *general*, so we introduced one name for the tree at this point, and substituted it into both starting formulas. The result was a negated three-way conjunction in each case, and the leftmost branch is the quickest to become complete. (The rightmost branch is the only closed branch among the nine.) The left branch generates a simple model, with domain $D = \{a\}$, and $I(F)(a) = 0$, and $I(G)(a)$ allowed to take the value either 1 or 0 as desired.

A model that results from the leftmost branch, with domain $D = \{a\}$, and $I(F)(a) = 0$, is enough to show how $\sim(Ix)(Fx, Gx)$ could be true without $(Ix)(Fx, \sim Gx)$ being true too. In this model, $\sim(Ix)(Fx, Gx)$ is true because $(Ix)(Fx, Gx)$ is *false*. There is no object with property F in this model at all. So, $\sim(Ix)(Fx, Gx)$ is true. On the other hand, $(Ix)(Fx, \sim Gx)$ is *false* since (again) there is no object with property F at all. The argument is invalid.

This is *one way* to make the premise true and the conclusion false—if there is no object with property F. Can you think of any other way the premise could turn out true and the conclusion false?

FOR YOU: Is the converse of this argument valid? Does this hold?

$$(Ix)(Fx, \sim Gx) \vdash \sim(Ix)(Fx, Gx)$$

- (a)** Yes, this is valid. **(b)** No, it isn't.

ANSWER: **(a)** The argument is valid. A tree for $(\exists x)(Fx \& (\forall y)(Fy \subset y = x) \& \sim Gx) \vdash (\exists x)(Fx, \sim Gx)$ will close.

It follows from these two results that the inner negation $(Ix)(Fx, \sim Gx)$ is a strictly *stronger* statement, in general, than the corresponding outer negation, $\sim(Ix)(Fx, Gx)$. The inner negation entails the outer negation, but not vice versa. It is an instructive exercise to show that from the premise that there is one and only one object with property F , it follows that the inner and outer negations are equivalent:

$$(\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y)) \vdash (Ix)(Fx, \sim Gx) \equiv \sim(Ix)(Fx, Gx)$$

This, perhaps, goes some way toward explaining why people regularly slide between the inner and outer negation of a definite description, without taking care to distinguish between them. There is no need to distinguish them *if* a background assumption is in place—if there is, indeed, one and only one object with the property used in that description. In the presence of that assumption, the inner negation and outer negation can be exchanged freely.

On the other hand, in the *absence* of that assumption—if it is *false*—then the inner negation and the outer negation *cannot* have the same truth value, for we have

$$\sim(\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y)) \vdash \sim(Ix)(Fx, Gx)$$

That is, if the uniqueness condition is false then the outer negation must be *true* (as it's not the case that there's a unique F), and we also have

$$\sim(\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y)) \vdash \sim(Ix)(Fx, \sim Gx)$$

That is, if the uniqueness condition is false then the inner negation must be *false*. It follows from *that* that if the inner negation is equivalent to the outer negation, then the uniqueness condition *must* hold. We have the converse validity:

$$(Ix)(Fx, \sim Gx) \equiv \sim(Ix)(Fx, Gx) \vdash (\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y))$$

Combining all of these results we have the following equivalence:

$$\vdash (\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y)) \equiv ((Ix)(Fx, \sim Gx) \equiv \sim(Ix)(Fx, Gx))$$

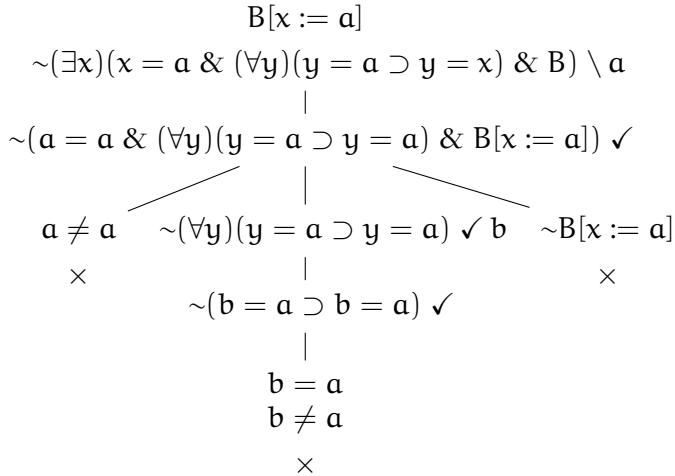
FOR YOU: Complete a tree, showing that $(\exists x)(Fx \ \& \ (\forall y)(Fy \supset x = y)) \equiv ((Ix)(Fx, \sim Gx) \equiv \sim(Ix)(Fx, Gx))$ is, indeed, a tautology in the language of predicate logic.

We will end this section with another result in the logic of definite descriptions. I will show that whatever you can do with a *name* you can do with a definite description. If something holds of the object a , it holds of the thing which is identical to a . That is, we have, for

any complex predicate B , if $B[x := a]$ (that is, the sentence found by replacing the free variable x in B by the name a), then it follows that $(Ix)(x = a, B)$. In other words, if a has property B , it follows that the thing which is identical to a has property B .

$$B[x := a] \vdash (Ix)(x = a, B)$$

Here is a tree, showing that this argument form is valid.



Notice that this tree contains the placeholder B for an arbitrary complex predicate. In the initial step of the tree proof, where we substitute a into the negated existentially quantified formula, the B inside this formula transforms into $B[x := a]$, since all free variables x in B , which were bound by the outer quantifier (Ex) , are replaced by the name a . This results in the sentence $B[x := a]$. This sentence *could* be processed using some rule or other, depending on the structure of the sentence $B[x := a]$. (Is it a conjunction, a negation, a quantified sentence, etc? It could be any of these, and different rules would apply in each case.) But regardless, we do not *have* to peek into the structure of $B[x := a]$ any more, because once the tree unfolds, the branches close without touching $B[x := a]$ (in the left and the middle branch), or in the right-most branch, where we end up with $\sim B[x := a]$, this closes with the formula $B[x := a]$ at the start of the tree.

FOR YOU: Confirm the converse: that $(Ix)(x = a, B) \vdash B[x := a]$.

argument.

which closes with the $\sim B[x = a]$, the negated conclusion from the $(\exists x)$ quantifier at the top of the tree), and $b = a$ to $B[x = a]$, Here, the crucial step is from $B[x = b]$ (found by substituting b in

$$\begin{array}{c} \times \\ [B[x = a] \\ | \\ [q = x] \\ (q = h \subset v = h)(h_A) \\ v = q \\ | \\ [q = h \subset v = h](h_A) \wedge v = q \\ | \\ [v = x] \\ [x = h \subset v = h](h_A) \wedge v = x(x\exists) \wedge b \end{array}$$

ANSWER: A tree for this is more straightforward:

7.2.3 | THE UPSHOT

This analysis of the structure and meaning of definite description statements seems to do a good job of explaining the difference between inner and outer negation—and the circumstances where inner and outer negations *agree*. It also provides us with a way to avoid difficult questions around the nature of so-called ‘non-existent objects.’ For Meinong, remember, if I deny the claim that the present King of France is bald, then the present King of France is the target of my thought, and I am denying that he is bald. If, as happens to actually be the case, I think that the present King of France fails to be bald because I take there to *be* no present King of France, this means—at least for Meinong—that I take the present King of France to merely *subsist* as the target of my thought, and I don’t take him to *exist*. This is a difficult pill for many to swallow—it leaves open very many questions about the nature of such merely subsistent, non-existent objects. But without an account of how *else* such non-existence claims are to be understood, these questions demanded answers.

With Russell’s analysis of definite descriptions in hand, we have a potential way to avoid such questions, by rejecting the assumption behind those questions. For Russell, if I deny that the present King of France is bald, this denial does not take the form of a negative ascription of a property to an object of thought (the merely subsistent present King of France), instead, it is simply a negated existentially quantified statement. If KF is the complex predicate ‘present king of France’ the claim that the present King of France is bald has the structure

$$(\exists x)(KFx \wedge (\forall y)(KFy \supset y = x) \wedge Bx)$$

and its negation has the form

$$\sim(\exists x)(KFx \ \& \ (\forall y)(KFy \supset y = x) \ \& \ Bx)$$

In *neither* of these sentences do we find a part which does the job of a *referring expression* referring to the present King of France (whether existing or subsisting). No, even in the positive statement, the part of the expression corresponding to the words “the present King of France” is the component underlined here:

$$\underline{(\exists x)(KFx \ \& \ (\forall y)(KFy \supset y = x))} \ \& \ Bx)$$

and this is *not* a referring expression—it is a part of an existentially quantified expression, with a hole remaining in it, to be filled with some other statement concerning x , the variable bound by the outer existential quantifier. This sentence fragment does not *refer* to anything: it contains a quantifier which ranges over the domain, and if the expression is true, then there is, indeed, some object in the domain of which KF is true, and indeed, it is the only object with that property (since $(\forall y)(KFy \supset y = x)$ is also true). However, the sentence also makes sense if it is *false*, and in this case there is *no* candidate for our expression to refer to, for there is either *no* object with property KF (so no choice for x to make KFx true), or there is more than one (and so, there is no choice for x to make $(\forall y)(KFy \supset y = x)$ true), so in either case, there is no target to be taken as the object of our thought. If I deny that the present King of France is bald, this is not to be understood as a judgement about a given thing—the present king of France. Instead, it is the denial of the threefold claim that (i) there is a King of France, (ii) that he’s the only King of France, and (iii) that he is bald. This can be denied by—for example—denying part (i), without *thereby* holding that there is some object (namely the King of France) whose existence I deny. I can deny the existence of an object satisfying some property without there *being* a particular object with that property whose existence I deny. Meinong’s distinction between existence and subsistence can be seen to rest on an assumption that Russell rejects in his analysis of definite descriptions.

7.2.4 | FROM HERE

Russell’s analysis of definite descriptions is not the end of the story for how definite descriptions are to be understood. In the wake of Russell’s original proposal at the beginning of the 20th Century, our understanding of the behaviour of definite descriptions has developed in a number of different ways.

UNIQUENESS? Most people—whether they agree with Russell on the details of his analysis of descriptions—agree that there is some kind of uniqueness condition in definite descriptions. If we see bowl of ten apples and I talk about *the* apple in the bowl, where I mention no distinguishing feature to help you pick out a particular apple, you will

This happens *everywhere* with definite descriptions. If I talk about the Prime Minister or the President, for example, I somehow restrict my attention to a particular country, depending on the topic of conversation.

Let alone all of the fruit in the *whole of space and time*.

wonder what I could mean. I am violating the uniqueness constraint, since there is no single apple there—the bowl contains many. However, if I have a bowl of fruit in front of me, and that bowl contains one and only one apple, it seems that when I ask you to pass me *the* apple, you know what to do. Or if I say “the apple is green” you can check if what I say is true. However, according to Russell’s analysis, if I say “the apple is green,” it seems that literally speaking, what I say is false. For my claim has the form (i) there is an apple, (ii) it’s the only apple, and (iii) that apple is green. There is no problem with claim (i). It is claim (ii) where things fall down—understood literally, it is *not* true. There are millions of apples in the world. This is not the only apple—but it is the only apple *in the bowl*. It seems that in the definite descriptions you and I use every day, we have some way to *restrict* the domain of quantification for the inner uniqueness quantifier, even though such a restriction is not mentioned in the formula itself. How is this done? If Russell’s analysis is at all correct, the mechanism for this should go in parallel with the mechanism for managing the domain of other quantifier expressions, like “some” or “all.” And this seems appropriate: if, confronted with the fruit bowl, I said “*all* of the fruit is ripe” then you would understand me to be making a claim about all of the fruit *in the bowl*, not all of the fruit *in the world*. The quantifiers in other expressions like “some” and “all” seem to act in the same sort of way as definite descriptions. This complication is not a mark against Russell’s analysis—it seems to be a mark in favour of it.

ARE EXISTENCE AND UNIQUENESS STATED OR PRESUPPOSED? Another question about the interpretation of definite descriptions is how the existence and uniqueness conditions in the definite description claim are to be understood. According to Russell’s analysis, these conditions are a part of what is *said*, but an alternate analysis of definite descriptions takes the existence and uniqueness claims to be something that is not a part of the statement, but a *presupposition* of that statement. An analogy will help explain the difference. If I ask a student “have you stopped worrying about the test?” and he says *yes*, then I can infer that he was worrying but is worrying no longer. If he says *no*, I can infer that he was worrying and is *still* worrying. The claim that he was worrying is *presupposed* by both answers to the question. To deny this presupposition, he would have to say something like—“I was *never* worrying. The question of whether I’ve stopped or not doesn’t arise.” Perhaps the same goes with definite descriptions. I could say “the King of France is bald”—you could say “yes, he is” or “no, he isn’t” (and both of these responses *presuppose* the existence and the uniqueness of the King of France) while a further response “that issue doesn’t arise—since there’s no King of France to be bald or to fail to be bald” would reject the presupposition, and neither assert the statement or the negation of the statement.

This analysis of the structure of definite descriptions is quite different. On this view, both the claim that the F is G and its denial (the F

isn't G) *presuppose* the existence and the uniqueness of an F. This claim and its denial are negations of one another, but the *disavowal* of the presupposition is possible, and this gives us a different way to reject the claim, without asserting its negation.

WHAT ABOUT NAMES? A final response worth mentioning is it seems that the *names* we use in our natural languages might actually act more like definite descriptions than the names in the language of first order predicate logic. While many names definitely *do* refer to objects, I might learn of a name which dates back to the mists of time and legend (Homer, Methuselah, Arjuna, etc.) where some might think that there is a bearer of the name, and others might deny it. Perhaps there was someone to whom that name belongs, and perhaps there isn't. It seems that names like this are at least *possible*, and that names without bearers are at least grammatically correct, even if they are semantically defective if there is nothing to which they refer. This is impossible in first order predicate logic as we have studied it. Definite descriptions can be meaningful despite there not being some object they pick out. Why can't this be the case for names? One way to go is to treat names as quantifier expressions in just the same way as other definite descriptions—so instead of Ha as the form of “Adam is happy” (which *requires* that there be an object that the name ‘a’ denotes) it has the form $(\text{Adam } x)Hx$, which need not make the assumption that there is any object to which the name refers.

7.3 | FREE LOGIC

This response to the issue of existence and non-existence—considering the behaviour of definite descriptions as quantifiers, and generalising it, to treat *names* on a par with descriptions—is just *one* possible path to take. No long-standing philosophical problem has just *one* answer.

One reason that the response we have given so far is unsatisfactory to some is that not all claims ‘about non-existent things’ seem to be on the same footing. Take Pegasus, for example, the mythical (non-existent) flying horse. Pegasus is a flying horse, but no flying horses exist. Similarly, Godzilla is a (fictional) giant monster (*kaiju*) that *also* does not exist. People familiar with the concepts *Pegasus* and *Godzilla* will happily say

Pegasus is a flying horse. Godzilla is *not* a flying horse.

Godzilla has atomic breath. Pegasus does *not* have atomic breath.

While at the very same time, agreeing that

Pegasus does not exist. Godzilla does not exist.

Non-existent things can, it seems, have different properties. Godzilla is a non-existent kaiju that can unleash a radioactive heat ray from its



Pegasus

jaws. Pegasus is a horse that can fly with its majestic wings. There is no way to treat claims like these at face value, using anything like the analysis of definite descriptions and existence we have already seen. (Why not? Think of G as a property only had by Godzilla, and P as a property had only by Pegasus. The argument $\sim(\exists x)Gx, \sim(\exists x)Px \vdash (\exists x)(Gx, Px) \equiv (\exists x)(Px, Gx)$ is valid according to Russell's treatment of definite descriptions, so if Godzilla and Pegasus both don't exist then any feature had by Pegasus is had by Godzilla, and *vice versa*.)

If we wish to understand how claims like these could be jointly true—without taking the objects so described to *exist*—we need to have a different account of existence and names and predicates than the standard models for first order predicate logic.

★ ★ ★

Why? Because $\sim(\exists x)Gx \vdash \sim(\exists x)(Gx, Px)$, and similarly for Px in place of Gx.

Whatever it means to say that they *subsist* without existing, Different philosophers will have different ideas about what subsistence might amount to. This is one of the virtues of doing the *logic*. We can start building models and looking at how they work, even if we don't have a settled view on the nature of subsistence. After all, we have used models of first order predicate logic, while not settling on some fixed answer of what it means for something to *exist*, either.

The OFFICIAL DEFINITION will require that the set D of objects is non-empty, but the set E of existing objects is allowed to be empty. For those familiar with set-theoretical notation, we require $\emptyset \subseteq E \subseteq D \neq \emptyset$.

To begin to give an account that has some hope of explaining how it is that different non-existent objects can have different features, it seems appropriate to go back to Meinong's understanding of the difference between *existence* and *subsistence*. If we are to not stray too far from standard models of first order predicate logic, we could allow names (such as Pegasus, Godzilla, Frodo Baggins or Superman) to pick out objects which merely *subsist* rather than requiring that they *exist*. Of those things which subsist, some, in addition, *exist*. So, our models of the formal language will have a domain D as always, of objects which at least subsist. Of those objects, *some* of them *exist*, so there is a subset E of D, of those things which also exist.

The result of this picture of an outer domain containing an inner domain is what is called *free logic*—a logic which is free of ‘existential import.’ That is *nothing* in the language requires that something *exist* in order to interpret it. In standard predicate logic, the *predicates* have no existential import. There is no requirement, for any predicate, that any particular object *exist* in order to interpret that predicate. Predicates do not need to have some objects to satisfy them. Names, on the other hand, *do* have existential import. For a name in the language of predicate logic to be interpreted there must be an object in the domain to be the value or *referent* of that name. This object *exists*, according to the language of predicate logic, since it is ranged over by the existential quantifier. For any name a , it is a tautology of the language of first order predicate logic that $(\exists x)x = a$ —that is, there *exists* something that is identical to a . Names have existential import in first order predicate logic in a way that predicates do not.

DEFINITION: A *free logic* is a logic in which no non-logical item of vocabulary—not even names—has existential import.

Why the rider concerning ‘*non-logical items*’? Because the existential quantifier and the existence predicate will have existential import—because that’s what they’re designed to *say*. But they are logical concepts—logical constants.

In a free logic, therefore, names and predicates are *equally* ontologically neutral. There are no concepts or words which *just in virtue of meaning*

fully using them require other objects to come into existence or to stay in existence.

In the ‘outer domain’ ‘positive’ free logic we will be exploring, is achieved without major changes to the models of first order predicate logic by taking the original domain D (from which all names are interpreted) to be the *outer* domain of objects which at least *subsist*. In this way, names are still interpreted as referring to objects, but we have jetisoned the assumption that these objects need to *exist* to play their role in interpretation.

7.3.1 | MODELS FOR FREE LOGIC

Now we can define models for the language of predicate logic. The language is just the same as we have already seen, except we add a single logical predicate, $E!$, the *existence predicate*, which is used to mark the difference between existence and subsistence.

A model for INNER-DOMAIN POSITIVE FREE LOGIC consists of

- A *regular* model $\langle D, I \rangle$ of first-order predicate logic, in which all of the non-logical items (names and predicates) are interpreted in the usual way (the domain D is the set of things which at least *subsist*, according to the model), and
- a special subset set $E \subseteq D$ of the domain, the *inner* domain. This consists of all of the items that are taken to *exist*.

Given the model, it is possible to interpret sentences in the language of predicate logic in the usual way:

Sentences in the language of free logic are interpreted in a model in the following way:

- $Fa_1 \dots a_n$ is true in the model iff $I(F)$ assigns 1 to the n -tuple $\langle i(a_1), \dots, i(a_n) \rangle$.
- $E!a$ is true in the model iff $I(a)$ is in E .
- $A \& B$ is true in the model iff A is true in the model and B is true in the model; $\sim A$ is true in the model iff A is not true in the model,...etc
- $(\forall x)A$ is true in the model iff $A[x := e]$ for every e in E .
- $(\exists x)A$ is true in the model iff $A[x := e]$ for some e in E .

The only difference between this and the standard clauses interpreting the language in standard non-free models is the addition of a clause for $E!$, and the restriction to the inner domain E in the interpretation

of the *quantifiers*. This restriction is due to the interpretation of the existential quantifier as that quantifier that ranges over *existent* objects. Rather than talking about these clauses in the abstract, it will be clearer to illustrate the definition with an example. Consider this model, with domain $D = \{a, b, c, d\}$.

	E	$I(H)$	$I(F)$	$I(S)$	a	b	c	d
a	0	1	1	a	0	0	0	0
b	1	1	0	b	1	0	0	0
c	1	0	0	c	1	1	0	1
d	0	0	1	d	0	1	0	0

The inner domain here is represented in the column for E: in other words, $E = \{b, c\}$.

- $Fa \ \& \ Ha$. This is true in the model because both H and F are interpreted in such a way as to assign the value 1 in the a row.
- $(\forall x)(Hx \supset \neg Fx)$. This is *true*, because the instances $Hb \supset \neg Fb$ and $Hc \supset \neg Fc$ are both true, and b and c are the only objects in the inner domain. For every *thing* (every *existing* thing), if it's H then it isn't F. (If it's a horse, it doesn't fly, for example.) This does not contradict the claim that Ha and Fa are both true, because a does not exist—that is, a is only in the outer domain, not the inner domain.
- Scd . This is true. There is a 1 at the intersection of the c row and d column in the $I(S)$ table.
- $(\exists x)Sbx$. This is *false*. The only instance of Sbx that's true is Sba , and a is only in the outer domain, not the inner domain.
- $Ha \ \& \ \neg Hd \ \& \ \neg E!a \ \& \ \neg E!d$. This is *true*. This statement illustrates how two different non-existent (merely subsistent) items can differ in their properties. Although a and d both don't exist (they merely subsist), we have Ha and $\neg Hd$.

FOR YOU: Which of *these* formulas are true in the model?

1. $\neg E!a \ \& \ (Fa \ \& \ Ha)$
2. $(\exists x)(\neg E!x \ \& \ Hx \ \& \ Fx)$
3. $Scd \supset Hd$

4. $(\forall x)(Sx \supset Hx)$

1. TRUE; 2. FALSE — no formula of the form $(\exists x)(\sim Elx \And \dots)$ is ever true in models of outer domain free logic, since $(\exists x)$ ranges over only objects in E , while $\sim Elx$ is only true of objects *outside* E . 3. Scd is true, and Hd is false, so this conditional is false. Finally, $(\forall x)(Sx \supset Hx)$, since for any of the existent objects (b and c) if Scx relates c to that object x then indeed, x must have property H (it must be b).

Models for free logic are straightforward, and they are not much more difficult than models for standard first order predicate logic. We can use them to define tautologies, contradictions, logical equivalence and logical validity in the usual way. Not a great deal is different from first order predicate logic—but some things are different.

EXAMPLE: $(\exists x)(Fx \vee \sim Fx)$ is *not* a tautology in free logic. (It is in standard predicate logic.) This formula can be made false in any model in which the inner domain is *empty*. In such a model there is no instance of $(\exists x)(Fx \vee \sim Fx)$ that is true, since there is no name to substitute into it which picks out an object in the inner domain. There is *no* object in the inner domain, so *no* existentially quantified sentence is true in any model with an empty inner domain.

EXAMPLE: The argument from $(\forall x)Fx \And (\forall x)Gx$ to conclusion $(\forall x)(Fx \And Gx)$ is valid in free logic. Here is why: suppose we have a model in which $(\forall x)Fx \And (\forall x)Gx$ is true. It follows that $(\forall x)Fx$ and $(\forall x)Gx$ are both true in the model, and hence that for each object e in the inner domain, Fe and Ge are true, and hence $Fe \And Ge$ is true for each such object, and hence $(\forall x)(Fx \And Gx)$ is true too.

There are many more things we could say about logical consequence in free logic. For now, let's look at how we can modify the tree rules for classical first order predicate logic to apply to free logic instead. This turns out to be surprisingly straightforward.

7.3.2 | TREE RULES FOR FREE LOGIC

We modify the tree rules for free logic in only one way: with changes to the rules for quantifiers. The particular rules—those that introduce new names, from $(\exists x)A$ or $\sim(\forall x)A$ —introduce not only the instance of the relevant formula, but *also* the claim that the introduced object gen-

uinely *exists* and doesn't merely *subsist*. As for the general rules, these are slightly more complicated than the particular rules. For example, in the previous example we discussed, the generalisation $(\forall x)(Hx \supset \sim Fx)$ is true, even though $Ha \supset \sim Fa$ is false. In this case, the substitution of a for x in $Hx \supset \sim Fx$ would not be appropriate in a tree—at least not without further work in explaining how the rules are to be applied in this new world. Here are the rules for the existential quantifier—*thee* introduce a new name, and along with it, the claim that the object referred to exists and doesn't merely subsist.

$$\begin{array}{c} (\exists x)A \\ | \\ E!n \\ A[x := n] \quad n \text{ new} \end{array}$$

On the other hand, the rule for the negated existential quantifier allows you to substitute any name you like, but instead of substituting it directly, the tree branches: either that object doesn't exist, or (if it does) it is fair to substitute into the original formula.

$$\begin{array}{c} \sim(\exists x)A \\ \swarrow \quad \searrow \\ \sim E!a \quad \sim A(x := a) \quad \text{any } a \end{array}$$

The same goes for the rules for the universal quantifier. These allow for any substitution you please, but the tree branches into two options, as with the negated existentially quantified formula.

$$\begin{array}{c} (\forall x)A \\ \swarrow \quad \searrow \\ \sim E!a \quad A(x := a) \quad \text{any } a \end{array}$$

Finally, the negated universal quantifier rule is just like the existential quantifier rule, with its provision for existence claims.

$$\begin{array}{c} \sim(\forall x)A \\ | \\ E!n \\ \sim A[x := n] \quad n \text{ new} \end{array}$$

With these tree rules at hand, we can use trees to test arguments. Here is a simple tree, testing the argument from $(\forall x)(Fx \supset Gx)$ and $(\forall x)(Gx \supset Hx)$ to the conclusion $(\forall x)(Fx \supset Hx)$. It turns out to be just as valid as in the classical predicate calculus, though the tree showing that it is valid has rather more branching in the classical case:

$$\begin{array}{c}
 (\forall x)(Fx \supset Gx) \setminus a \\
 (\forall x)(Gx \supset Hx) \setminus a \\
 \sim(\forall x)(Fx \supset Hx) \checkmark a \\
 | \\
 E!a \\
 \sim(Fa \supset Ha) \checkmark \\
 | \\
 Fa \\
 | \\
 \sim Ha \\
 | \\
 \sim E!a \quad Fa \supset Ga \checkmark \\
 | \quad | \\
 \times \quad \sim Fa \quad Ga \\
 | \\
 \sim E!a \quad Ga \supset Ha \checkmark \\
 | \quad | \\
 \times \quad \sim Ga \quad Ha \\
 | \quad | \\
 \times \quad \times
 \end{array}$$

Outer domain free logic is a straightforward way we can model the language while finding room for different non-existent objects with different properties. Unfortunately, it does not answer every question we might have about non-existent objects. After all, we have different merely subsistent, non-existent objects with different properties, but these objects are picked out by way of *names*. Sometimes, it seems, we wish to pick out non-existent objects by *description* rather than by name, but outer domain positive free logic, as it stands, is useless for this. If you consider the Goldent Mountain, you could attempt to model talk about the Golden Mountain in terms of a definite description like this

$$(\exists x)((Gx \& Mx) \& (\forall y)((Gy \& My) \supset y = x) \& \dots)$$

which says that there is a Golden Mountain, it's the only one and ... etc. This is *useless* in the language of free logic, because the existential quantifier in this expression ranges only over the *inner* domain, not the outer domain. There exists no Golden Mountain, so this attempted description does not help to pick it out. But it's clear that something else might—if we could have a quantifier that ranges over the *outer* domain as well as the inner domain (say Σ for the ‘some’ and Π for the ‘all’ quantifier), then it seems that a description like this

$$(\Sigma x)((Gx \& Mx) \& (\Pi y)((Gy \& My) \supset y = x) \& \dots)$$

might do the trick, for there is *some* gold mountain in the outer domain, etc. However, even with this change, we have remaining cause for concern. Although we have a quantifier that ranges over the outer domain, questions remain. What reason do we have to think that there is one and only one *Golden Mountain* in the outer domain? Aren't there many distinct golden mountans? One shaped like the Matterhorn, and one as tall as Everest? Why not? But wouldn't this render traditional



Remember the Golden Mountain?

definite description structure useless, for we would expect the uniqueness clause to fail to hold in almost all cases. (This sort of outer domain free logic in which we have a quantifier that ranges over the outer domain is called a *Meinongian* free logic, after Alexius Meinong.)

Instead of answering all of those questions—and those questions are a matter of active research and exploration among philosophical logicians to this day—I will bring the chapter to an end with a little reflection on the nature of the relationship between logic and philosophy.

7.3.3 | LOGIC AND PHILOSOPHY

- As we have seen, we can use logic to *inform* philosophy, by giving us new ways to represent information about the world and relationships between our claims. We have seen in this chapter how different ways to chart the boundary between existence and nonexistence are motivated and facilitated by different techniques in the syntax and models of predicate logic—the syntax of definite descriptions and their analysis in the language of predicate logic with identity, and the development of outer domain models of free logic. These techniques help make precise some of the less clear and untrained intuitions involved in arguing about these issues.
- But this is not the only relationship between logic and philosophy. As we saw last class with Logic: Language and Information 1, there are different philosophical positions involved in the use of logical techniques—the assumption that every proposition is either true or false is a substantial position—at least to the extent that this position is *opposed* by some who think that a more accurate picture is found by allowing truth values to range between 0 and 1, as in free logic.

The relationship between logic and philosophical questions is a live and active one—tools from logic are applied to shed light on different issues in philosophy, and the critical eyes of the philosopher can have their uses when we want to critically reflect on what it is that our techniques assume, to see if that's the only way to do things, and to explore other options. In all of this, we are doing philosophy of logic.

QUANTIFIERS IN LINGUISTICS

8

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- Contextually determined ranges of quantifier expressions in natural language.
- The difference between the truth conditions and the conversational implicatures of natural language quantifier expressions.
- The meaning of a quantifier expression as determining a particular property of a pair of sets from the domain.
- *Corefrential* and *bound variable* interpretations of pronouns in natural language.
- Scope distinctions in natural language syntax and semantics.
- *Donkey sentences* and the use of pronouns seemingly outside the scope of a corresponding quantifier.
- *Lexical, structural* and *scope* ambiguities in sentences.

As in Part 1, we gratefully acknowledge the generous assistance of our colleague, [Lesley Stirling](#), who developed lectures in [UNIB10002](#) from which these notes are derived.

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.

SKILLS

These are the skills you will learn to demonstrate.

- Explain the way that the range of a quantifier expression in natural language might be shifted as the conversation shifts.
- Determine the difference between the truth conditions and the conversational implicatures of natural language quantifier expressions.
- Explain the difference in meaning between unary and binary quantifiers, and judge between better and worse candidates for the meaning of natural language quantifier expressions.
- Distinguish different uses of pronouns in natural language, and translate difficult quantificational constructions such as donkey sentences.
- Distinguish lexical, structural and scope ambiguities in sentences.
- Distinguish different interpretations of natural language sentences corresponding to different grammatical structures or scopes for the quantifiers in the formalisation of that sentence.

WHAT DO LINGUISTS DO? As we saw in *Logic: Language and Information* 1, linguistics is the study of *language* in *all* its aspects. Languages have many different aspects and features, so linguistics has many different branches:

- *Phonetics* and *phonology* — sounds in language production.
- *Morphology* — morphemes, the individual units in languages.
- *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
- ...

As a result, the study of language is rich, complex and varied. Logic connects with language in a number of different ways, especially in syntax, semantics, and also in pragmatics and discourse analysis. In this part of the course, we'll be looking at three different aspects of the meanings of *quantifier* expressions in natural language—first, we will look at the general framework for quantifier meaning provided by first order predicate logic, second, we will look at the interactions between quantifiers and *pronouns* in natural languages, and finally, we will look at the kinds of *ambiguity* that natural language quantifiers make possible.

8.1 | QUANTIFIER MEANING

Logic proves useful in helping us explain *meanings*—I can clarify the meaning of an expression by distinguishing circumstances where it applies from circumstances where it doesn't. I learn the meaning of the word “elm” better if I gain a better understanding of what counts as an elm and what doesn't. We clarify the meaning of the word “or” if we make more precise whether we are understanding it inclusively, or exclusively—of whether it would be correct to say $p \text{ or } q$ in a circumstance where p and q are both true, or whether that would be incorrect. The tools of formal logic are often very useful in this enterprise of clarifying and explaining meanings, because we have a very precise account of the truth conditions of sentences in a restricted language: in our case, the language of first order predicate logic with identity. For these sentences, there is no ambiguity about whether something counts as a sentence or not and there is no ambiguity about whether or not a sentence is true in a model. Relating other sentences to sentences in this kind of language is one way to force you to clarify meaning—it forces you to confront choices which you may not have previously considered. In this section, we'll take a look at the meanings of *quantifier expressions*, like “all” and “some” but also other quantifier expressions from natural language.

Let's start with a simple example. Let's suppose we are having a party, and we have asked a few people to bring some beer. After a few of the people have arrived, and brought beer, I say "All of the beer is here." What did this *mean*? Could it have been *true*?



"All of the beer is here."

No matter how much beer people bring to the party, there's no doubt that it isn't *all* of the beer in the world—it isn't even all of the beer in the world that existed at the time of the party. But *no-one* when I say "all of the beer is here" would think I mean *that*. Everyone would understand, instead, that this means that all of the beer *intended for the party* is here. There is no beer (that anyone else was planning to bring) that is *not* here. This is genuinely a *universal* quantifier that is being used when I say "all the beer", but there is some kind of *restriction* in place in our interpretation of that quantifier. We only select items from some restricted domain—not a domain of all things, or even a domain of all things that we've been talking about in that very conversation. So, this is one aspect of the meaning of quantifiers we should remember:

We could, for example, have just been talking our favourite beers from previous parties.

Natural language quantifiers exploit restrictions on the domain of quantification. These restrictions can vary from one conversation to another.

Determining the relevant domain of a quantifier expression is not a simple task, and it seems that quite young children engage in quite complex acts of interpretation involving quantifiers.

Consider this image—in which there are four elephants, three with riders, and one without. If you ask young children to agree or disagree with the statement "every man is riding an elephant" you can often get a surprising reply.



“Every man is riding an elephant.”

You might expect to receive the answer *yes*, every man *is* riding an elephant, on the assumption that the domain of quantification is implicitly restricted to the people *in the picture*. Of the men in the picture, every one is riding an elephant. Nonetheless, a significant proportion of children answer *no*—not every man is riding an elephant. When probed as to the meaning of this reply, they respond like this—these three elephants have their rider with them, while this elephant here is missing its rider—its rider must be off somewhere else, *not* on his elephant. This is a coherent explanation, and one which makes sense of the reply. It also hints at the complex and subtle features involved in shifting the domain of quantification. The children have used the background assumption that every elephant has some rider to expand the domain of quantification out to include that other rider, wherever he or she is. There was never any thought to expanding the domain out *unrestrictedly*—no child answers to the effect that their father is not riding an elephant, so no, not every man is riding an elephant—but they were prone to expand the focus of the answer, if background assumptions about the scene allowed it.

For the next case, consider the following claim about *Graz*, an Austrian band, depicted here. Suppose I say “some of the musicians are wearing black”, describing them as they appear in this photo:



“Some of the musicians are wearing black.”

Is what I said *true*, when it is clear that in fact *all* of the musicians are wearing black? Many hearers would think that when I said that *some* of the musicians are wearing black, I was saying that some *but not all* of the musicians are wearing black. Is this a part of the meaning of the claim, or is it an *implicature*? You might reasonably think that if I merely

said that *some* are wearing black, and I didn't say that all were, that I don't know that all of them are. So, at the very least, if I am being as informative as I might reasonably be, all things considered, if I say that *some* are wearing black, I don't know that all of them are. But suppose I look at the musician one by one, and after seeing a couple, I say "some of them (at least) are wearing black"—in this case I clearly can't know that not all of them are wearing black, but if "some" in literally *meant* "some but not all", I shouldn't be so confident to assert that some are wearing black if I don't have any information that not all of them are. For this reason, many of us think that the "some" in English might well *implicate* the conclusion "not all" but this isn't literally a part of the meaning of what is said. At the very least, "all" is understood to be more informative than "some," so when it comes to the maxim of quantity in cooperative dialogue, we can reasonably take it if someone says that *some* Fs are Gs, then they do not know that *all* Fs are Gs.

What's an *implicature*? It's something that you can reasonably infer from the fact of what I've said, even though it isn't something which follows literally from the content of what I've said. See *Logic: Language and Information 1* for more on implicatures.

"Some" and "all" have *implicatures* in natural language expressions. It's reasonable to conclude in many circumstances that if someone says "Some Fs are Gs" they do not also not also know that "All Fs are Gs." However, there is no evidence that "some Fs are Gs" is *inconsistent* with "all Fs are Gs."

What about at the lower end of the number required for "some"? Suppose *one* of the musicians were wearing black? In that case would it be correct to say that some of the musicians are wearing black? Or does some require two or more? Or even more? The existential quantifier of first order predicate logic asks for one or more. Does "some" in English mean "one or more" or is the lower threshold higher than just one?

This phenomenon of vagueness or imprecision at the threshold is more prominent in *other* quantifiers, besides *some* and *all*. Consider "most".



"Most of the girls are wearing head coverings."

In this scene, there are 9 girls and 3 boys. Of the 9 girls, 6 are wearing headcoverings, and 3 are not. Would it be correct to say that *most* are wearing headcoverings? When we asked our students on campus at

Ha! Most! What do we mean by most?

We're not telling. But in *this* case it was over 80% of those who responded.

the University of Melbourne, most of them said that yes, most of the girls were wearing headcoverings. A few said *no*, they either thought that it was false (because they wanted a threshold of $\frac{3}{4}$ or $\frac{4}{5}$ or so for ‘most,’ not the $\frac{2}{3}$ we have here), while others demurred and said that they didn’t want to commit themselves with a ‘yes’ or a ‘no.’ What do you think? Are *most* of the girls in this scene wearing head coverings?

It seems to us that nothing in the way we use ‘most’ settles this definitively to everyone’s satisfaction. But if we’re having a discussion where something of substance *turns on* the question of whether most Fs are Gs, we can always make things *more precise* by positing a particular way to interpret the quantifier. One precise way to do this is the 50% + 1 interpretation, according to which “most Fs are Gs” is true when the number of F that are Gs is at least 50% + 1 of the number of Fs as a whole. Then we would have a precise account of the borderline between when a “most” statement holds, and when it doesn’t. In this case, the claim that most of the girls are wearing headcoverings is true, since 6 is more than 50% of 9. But the claim that most of the *children* are wearing head coverings is false—since 6 of the 12 children aren’t wearing head coverings.

This process, of more precisely specifying the meaning of a quantifier expression can be made general. A *binary quantifier expression*, like “most” will occur in a sentence with this sort of structure

Q Fs are Gs

where the “Q” could be any sort of quantifier like “most” or “few” or “many” or “all” or “some” or “no” ...In the case of “most” we said that *most Fs are Gs* should count as true when the number of Fs that are Gs is 50% + 1 or more of the number of Fs. We could write this in the following way:

most Fs are Gs if and only if $\#(\llbracket F \rrbracket \cap \llbracket G \rrbracket) > \# \llbracket F \rrbracket / 2$

NOTATION: ‘ $\llbracket F \rrbracket$ ’ is the EXTENSION of the predicate F—it is the collection of things in the domain which have the property F.

So, $\llbracket F \rrbracket \cap \llbracket G \rrbracket$ is the intersection of two extensions: it is the set of things that have property F (those that are in $\llbracket F \rrbracket$) and the things that have property G (those that are in $\llbracket G \rrbracket$).

Finally, $\# \llbracket F \rrbracket$ is the NUMBER of things in the extension of F—the number of things which have that property.

So, in the specification of the meaning of ‘most’, when we write $\#(\llbracket F \rrbracket \cap \llbracket G \rrbracket) > \# \llbracket F \rrbracket / 2$, this says that the number of things that are both F and

G is greater than a half of the number of things that are F —which is another way of specifying the 50% + 1 clause

FOR YOU: If we read the ‘most’ quantifier with the 50% + 1 interpretation in finite domains, is the argument from $(\text{most } x \ Fx) Gx$ and $(\text{most } x \ Fx) Hx$ to the conclusion $(\exists x)(Gx \ \& \ Hx)$ valid or invalid? What do *you* think? What reasons can you give for your answer?

If you like, post your answer in the Coursera forums and check what other students think.

This notation can be useful for specifying the meanings of *other* natural language quantifiers. Check for yourself that these make sense.

all Fs are Gs if and only if $\#([\![F]\!] \setminus [\![G]\!]) = 0$

some Fs are Gs if and only if $\#([\![F]\!] \cap [\![G]\!]) > 0$

no Fs are Gs if and only if $\#([\![F]\!] \cap [\![G]\!]) = 0$

What about *few* Fs are Gs? Could we say this?

few Fs are Gs if and only if $\#([\![F]\!] \cap [\![G]\!])$ is *small*

Perhaps we could, but the question remains: what counts as being *small* in this definition? Is it a single number that is fixed in this definition (like being less than 10?) Or is it a proportion of the number of Fs in total? A close examination of the ways we use “few” will help decide this issue.

The account of meaning of quantifiers we are sketching here is a part of what is now known as *Generalised Quantifier Theory*, and it’s an active research program in logic and linguistics to this day [7].

The *binary* structure of quantifiers provides us a lot of scope for different quantifiers. It is surprising, then, that so much can be done with *one-place* quantifiers $(\forall x)$ and $(\exists x)$. The binary *some* and *all* quantifiers can be defined in terms of one-place quantifiers and propositional connectives:

$$\begin{aligned} (\text{some } x \ Fx) Gx &\text{ is } (\exists x)(Fx \ \& \ Gx) \\ (\text{all } x \ Fx) Gx &\text{ is } (\forall x)(Fx \supset Gx) \end{aligned}$$

We can do more for other quantifiers, too. In Chapter 4 we showed how finite numerical quantifiers (*at least n*, *at most n* and *exactly n*) can be defined in terms of the propositional connectives and unary quantifiers. Could we do the same with other quantifiers? It turns out that we can’t always do this. For example, it would be useful if there were some way we could define ‘*most*’ by having some unary quantifier Q and some propositional complex ?? such that

$$(\text{most } x \ Fx) Gx \text{ is } (Q x)(Fx ?? Gx)$$

But we have no such luck.

Here, $A \setminus B$ is the set of all things that are in A but not in B , so $\#([\![F]\!] \setminus [\![G]\!]) = 0$ if and only if there is nothing that is an F that isn’t also a G .

Notice that in each of these definitions, what is important is the *number* of objects in the domain satisfying a property, not *which* objects do that. This goes some way of explaining the connection between quantifiers and *quantities*.

It’s also enlightening to explore the difference between the quantifiers “few” and “a few”. Despite the superficial similarity, they are very different. It seems that if *few* Fs are Gs and all Hs are Fs, then *few* Hs are Gs, too. But this inference is not valid for ‘a few’. The converse seems to be valid. If *a few* Fs are Gs and all Fs are Hs, then *a few* Hs are Gs, too (namely, those Fs, at least). Also it is worth examining the connection between ‘few’ and ‘many’—is ‘few’ synonymous with ‘not many’?

Jon Barwise and Robin Cooper proved [2] that *nothing* we can express in the language of first order predicate logic could work in the place of the Q and ?? for ‘most’ In some sense, ‘most’ is an *essentially binary quantifier*.

For our next topic on the linguistics of quantifiers, we’ll look at the interaction between quantifiers and pronouns.

8.2 | QUANTIFIERS AND PRONOUNS

Pronouns, like *she*, *he* and *it*, play an important role in language. They help us say things more efficiently and with less repetition that would be possible in a language without pronouns. In the two sentences:



Tenzing Norgay saw Everest. He climbed it.

the ‘he’ and the ‘it’ act as *shorthands*. If I were to say

- *Tenzing Norgay saw Everest. Tenzing Norgay climbed Everest.*

This would be a slightly more lengthy way of saying

- *Tenzing Norgay saw Everest. He climbed it.*

The ‘he’ and the ‘it’ can be simply treated as shorthands or abbreviations for ‘Tenzing Norgay’ and ‘Everest.’ There are subtle rules and conventions governing how to pick *which* pronoun is a shorthand for *what* name in the dialogue. But the intuitive understanding of pronouns as shorthands is incomplete, when it’s combined with quantifier expressions. Consider *this* use of the pronouns:

Tenzing Norgay saw a mountain. He climbed it.

Could the ‘it’ in the second sentence act as shorthand for ‘*a mountain*’?



Tenzing Norgay saw a mountain. Tenzing Norgay climbed a mountain.

These statements are consistent with Tenzing Norgay seeing one mountain (Mount Everest, say, at 8,848m elevation above sea level) and climbing another (Mount Kosciuszko, Australia's highest mountain, which is a relatively easy stroll at 2,228m elevation above sea level). Whatever the pronoun ‘it’ is doing, it isn’t simply a shorthand the expression ‘a mountain.’ It is, in some sense, pointing back to the earlier expression, but not in a way that simply repeats it. To see what is going on, it is worthwhile to attempt to render each string of sentences as a formula in the language of predicate logic. For the first, we could get something like this:

- *TN saw Everest. He climbed it.*
- Sne, Cne

Where n is Tenzing Norgay, e is Mt. Everest, Sxy is ‘ x saw y ’ and Cxy is ‘ x climbed y .’ This is relatively straightforward. For the second, though, we need to use a quantifier for the first sentence.

- *TN saw a mountain. He climbed it.*
- $(\exists x)(Mx \ \& \ Snx), \dots$

But we couldn’t just say $(\exists x)(Mx \ \& \ Cnx)$ for the second sentence, because the mountain he climbed need not be connected in any way to the mountain he saw, because the scope of the second quantifier is different from the scope of the first. We couldn’t just say Cnx alone (he climbed x), because this second sentence is not in the scope of the first quantifier. The only way to encode this in the language of predicate logic is to encode it as *one* sentence, and to bring the Cnx (which is surely correct), *inside* the scope of the existential quantifier, like this:

- *TN saw a mountain. He climbed it.*
- $(\exists x)((Mx \ \& \ Snx) \ \& \ Cnx)$

Where the formula says, there’s a mountain, Tenzing Norgay saw it, and he climbed it. The pronoun ‘it’ in the second sentence is doing the job of a variable. It’s a variable that’s bound by the quantifier *in the first sentence*, and as a result, we need to have that variable inside that quantifier’s scope. There is no way to encode this in a pair of sentences from predicate logic—it needs to be a single sentence. We can see, then, that the tacit assumption that a sentence of a natural language can be translated into a formula of the language of predicate logic, fails. The sentence “He climbed it” does not occur as a single formula, but rather as a part of a formula.

DEFINITION: An occurrence of pronoun in a natural language sentence has a **COREFERENTIAL** interpretation if the expression in which it features it can be translated into the language of predicate logic,

in which the occurrence of the pronoun is translated by the second occurrence of a name.

An occurrence of a pronoun in a natural language sentence has a **BOUND VARIABLE** interpretation if the expression in which it features can be translated into the language of predicate logic, in which the occurrence of the prounoun is translated by a variable bound by a quantifier.

These definitions mark the different behaviours of pronouns in English and other natural languages. Some do act in ways that are essentially repeating names or other parts of speech—these are called *coreferential*. Some like the “it” our second example, must be translated with a bound variable—these are called *bound variable* occurrences. Being aware that there are bound variable occurrences of pronouns is an important part of discerning the logical or semantic structure in our forms of words. Sometimes the logical depth of such expressions is significant. A string of sentences in natural language could contain a number of pronouns which demand treatment as variables bound by quantifiers occurring in previous sentences. Here is an example:

Some students of logic are really quite smart. *They* can understand quite a few ideas that many other people find hard to understand. *These ideas* contain concepts that have have complex logical structures. Logical structures like *those* take quite some hours of practice to come to comprehend. *That kind of practice* is an investment that not everyone is prepared to make. After all, *such an investment* requires enough time free from the other tasks of life...

In this string of four sentences, each word or words italics will be interpreted as a variable bound by quantifier occuring in the previous sentence.

FOR YOU: Select which of the following pronouns (underlined) *must* be given a bound variable interpretation, not a coreferential interpretation.

- (a) Greg is outside. He's happy.
- (b) Several students are outside. They're happy.
- (c) Every boy thinks he's smarter than average.
- (d) Most students have read *Macbeth*. Most students like it.

But as it stands, in no wider context, it must give a bound variable rather than to *every boy*, could have a coreferential interpretation. Spoken in a way to make it clear that the *he* points back to Bob (Replicating if it occurs after another sentence, “Jim is popular at his school. Every boy thinks he’s smarter than average” — when enital reading if it occurs before another sentence, “Jim is popular at average”). You could use these same form of words with a coreferent interpretation.

ANSWER: (a) and (d) can be given coreferential interpretations.

Having seen the behaviour of pronouns as bound variables in these cases, we can now tackle some of the more interesting and puzzling semantic constructions involving quantifiers and pronouns—which have come to be known as *donkey sentences*.



Every farmer who owns a donkey beats it

How are we to display the logical structure of this sentence? Thinking about what the sentence *says*, it seems clear that it says that *for any* farmer, and *for any donkey* owned by that farmer, the farmer beats the donkey:

$$(\forall x)(\forall y)((Fx \wedge (Dy \wedge Oxy)) \supset Bxy)$$

After all, if we ever have a farmer *f* and a donkey *d* owned by that farmer, if the farmer *doesn’t* beat that donkey, we have a counterexample to the rule that every farmer who owns a donkey beats it. In other words, we’d have a counterexample to this conditional:

$$(Ff \wedge (Dd \wedge Ofd)) \supset Bfd$$

and this would be an *instance* of (and instance of) the doubly quantified formula we have above. So much seems clear enough. The interesting issue is that the English quantificational structure seems quite different than the double universal quantifier. When I say every farmer who owns a donkey... it seems that the logical structure is this:

$$(\forall x)((Fx \wedge (\exists y)(Dy \wedge Oxy)) \supset \dots)$$

Some think that not all ‘donkey’ sentences have this reading. Some people think that “Every person who has a credit card pays for dinner with it when they go to an expensive restaurant” could be true even if there’s someone with *two* credit cards, who pays for dinner with one of them and not the other. That wouldn’t have this doubly universally quantified logical structure.

where we put the rest of the formula where those ‘...’ appear. The antecedent of the conditional seems to be “ x is a farmer and there’s some donkey that x owns.” And this is fair enough, *except* that the consequent of the conditional contains the pronoun ‘it’, which points back to the ‘a donkey’ quantifier—which is, in this case, an existential quantifier in the antecedent of the conditional.

I’ll end this section with an explanation of *why* this sort of doubly universal structure is to be expected despite the fact that the natural language expression does have an existential quantifier (‘a donkey’) rather than a universal in that position. The crucial feature is that the existential quantifier in the antecedent of the conditional

$$(\forall x)((Fx \ \& \ (\exists y)(Dy \ \& \ Oxy)) \supset Bxy)$$

This syntax, with the variable y outside of the narrow scope of the quantifier, is suggestive. Groenendijk and Stokhof’s “Dynamic Predicate Logic” [4] provides a semantics in which expressions like this not only have a meaning, they have the meaning we have sketched here. It takes us too far afield to introduce dynamic logic here.

You don’t believe this when you see it? Do a tree for the biconditional, and you’ll see that it closes.

Why not? Consider a model in which $D = \{a, b\}$ and $I(Ga) = 1$ but $I(Gb) = 0$, and $I(p) = 0$. Then $(\exists x)Gx$ is true and p is false, so $(\exists x)Gx \supset p$ is false in this model. But $Gb \supset p$ is true in this model, so $(\exists x)(Gx \supset p)$ is also true. It follows that $(\exists x)Gx \supset p$ and $(\exists x)(Gx \supset p)$ are not logically equivalent.

does not bind the variable y in the consequent of the conditional. The scope of the $(\exists y)$ is just the underlined part of the formula, no more. If we want to place the y in Bxy in the scope of the $(\exists y)$ quantifier, we must move the $(\exists y)$ *out* (or higher) in the structure of the formula. The first thing to note is that it’s under a conjunction. But this is straightforward:

$$\models Fx \ \& \ (\exists y)(Dy \ \& \ Oxy) \equiv (\exists y)(Fx \ \& \ (Dy \ \& \ Oxy))$$

because in general, we can distribute an existential quantifier over a conjunction where the conjunct does not contain the variable bound by that quantifier. (In general, $p \ \& \ (\exists x)Gx$ is logically equivalent to $(\exists x)(p \ \& \ Gx)$.) So, we can transform our formula to

$$(\forall x)((\exists y)(Fx \ \& \ (Dy \ \& \ Oxy)) \supset Bxy)$$

which expands out the scope of $(\exists y)$ a little, but not far enough to reach the Bxy . To do that, we need to go out from under the conditional. But now the logical equivalence does not work in the same way. A conditional $(\exists x)Gx \supset p$ is not logically equivalent to $(\exists x)(Gx \supset p)$. However, $(\exists x)Gx \supset p$ is logically equivalent to $(\forall x)(Gx \supset p)$. With that in mind, if we were to move the existential quantifier out from the antecedent of the conditional to *outside* the whole conditional, we’d expect it to change from an existential to a universal. The result would be

$$(\forall x)(\forall y)((Fx \ \& \ (Dy \ \& \ Oxy)) \supset Bxy)$$

and now, the $(\forall y)$ quantifier indeed does bind the y in Bxy . And we have a kind of explanation as to why the quantifier has become a universal. We have had to move the quantifier from the existential that is in the antecedent of a conditional into a universal that is now properly outside that conditional, binding *all* of the instances of y inside.

In this section we’ve looked at the interaction between pronouns and quantifiers. The gap between natural languages and the formal language of predicate logic is large enough to make translations between one and the other, not straightforward, and translating from natural language to the predicate logic gives us a way to clarify the kinds of dependence relations between different parts of speech—especially between

pronouns and quantifiers. In the next section, we'll explore the kinds of ambiguity that can appear in natural language sentences, especially those ambiguities involving quantifiers.

8.3 | QUANTIFIER AMBIGUITIES

Something is *ambiguous* if it could be taken in more than one way. A word or a phrase or a sentence is ambiguous if it can have more than one meaning. The compressed writing found in newspaper headlines leaves sentences which are prone to ambiguous readings.

British Left Waffles on Falkland Islands.

Kicking Baby Considered to be Healthy.

Sisters Reunited After 18 Years in Checkout Line.

Squad Helps Dog Bite Victim.

Many of the ambiguities in these headlines are *lexical* ambiguities—they are due to the way that individual words can be given different meanings, often from different semantic categories. In one reading of ‘*British Left Waffles*’, ‘Waffles’ could be a *noun*, so then the statement would say that the British have left Waffles (the dessert) on the Falkland Islands. On the other hand, it could be the *verb* for prevaricating, and then the sentence says that the Left of the British political spectrum prevaricates on the issue of the Falkland Islands.

The same sort of explanation can be given for the other sentences in this collection. Lexical ambiguities—different meanings for individual words—have the consequence that sentences can be parsed with very different structures.

DEFINITION: A word or phrase or sentence **AMBIGUOUS** if it can be given different meanings. A **LEXICAL AMBIGUITY** in a phrase or sentence is an ambiguity that arises out of an ambiguity in the meaning of an individual word.

You might think that the *only* ambiguities that are possible in a sentence would be lexical—perhaps because any difference in the meaning of a sentence must come from a difference in the meaning of individual words. But this isn't true. There are sentences which are ambiguous not because of any differences in the meanings of the individual words, but because of the differences in how those meanings are *combined*. Consider these sentences:

- 1' Boys think highly of themselves. Every boy thinks he is smarter than average.

1" Jim is popular at his school. Every boy thinks he is smarter than average.

In this case, the sentence "Every boy thinks he is smarter than average" is ambiguous in meaning, because it can be given different interpretations. In the first, the pronoun 'he' is a bound variable, bound to the quantifier 'Every boy.' In the second, the pronoun 'he' is coreferential with the name Jim, occurring in the previous sentence. In both cases, 'he' has the standard and usual meaning—as the male singular personal pronoun—but the way that pronoun is linked to the preceding material changes from one sentence to the other. This is one form of *structural ambiguity*.

DEFINITION: A STRUCTURAL AMBIGUITY in a sentence is an ambiguity that arises out of different ways the sentence can be parsed, even though the lexical meanings of each of its constituents are the same in each case.

Structural ambiguities arise not just because of the ways pronouns are interpreted. They can also arise out of the ways that *quantifiers* are interpreted. Consider the following two sentences:

- Two students read every book.
- Two arrows hit every target.

For most people who hear these sentences, they are parsed (interpreted) in very different ways, despite the surface structure being identical in both cases. If I say "two students read every book" you would probably hear me saying that there were two students who both read all of the books in some collection. It would have a form rather like this:

$$(\exists x_1)(\exists x_2)((Sx_1 \ \& \ Sx_2 \ \& \ x_1 \neq x_2) \ \& \ (\forall y)(By \supset (Rx_1y \ \& \ Rx_2y)))$$

There are two students (x_1 and x_2) such that for every book (each y where By) the students read those books (Rx_1y and Rx_2y). Here, the structure is relatively close to the structure of the English sentence. "Two students" comes first, and this is paralleled in the formula with the dual existential quantifiers at the head of the formula. Then *within that scope* we have the quantification over every book—both in the natural language sentence, and in the formula.

On the other hand, when I say that two arrows hit every target, most listeners understand this differently. I have a collection of targets, and what happened to each of them? Two arrows hit them. Which two arrows? It need not be the same in each case. Different arrows could have hit each target.



The result is a very different form:

$$(\forall y)(Ty \supset (\exists x_1)(\exists x_2)((Ax_1 \ \& \ Ax_2 \ \& \ x_1 \neq x_2) \ \& \ (Hx_1y \ \& \ Hx_2y)))$$

Now, the quantifiers are swapped with respect to each other's scope. The *each target* is given the most prominent position, and the choice of the arrows (the two existential quantifiers) is placed *under* that scope, in parallel with the way that the interpretation allows for *different* arrows for each target.

These different meanings for the different sentences are not a case of ambiguity. What would make this a case of ambiguity is if the *one* sentence could be interpreted in at least two different ways. In fact, I think that the arrow/target question *is* ambiguous. I think that this could be given the other interpretation—the interpretation according to which it was the particular two arrows which hit every target. This is a *deviant* (or at least, a less likely) reading than the standard one, with the universal quantifier dominant.

DEFINITION: A QUANTIFIER SCOPE AMBIGUITY in a sentence is a kind of structural ambiguity in a sentence which arises out of the different possibilities for nesting the quantifiers in the sentence.



A restaurant advertisement

The difference in interpretation of quantifiers can be very sensitive to the particular meanings of the terms involved. As another example, take a look at the two sentences here.

- John *visited* a restaurant on every street.
- John *advertised* a restaurant on every street.

It's understood that if you visit a restaurant on different streets, it's unlikely to be the same restaurant—as the same restaurant won't be in more than one place. On the other hand, you *can* advertise the one restaurant in more than one street—because you can advertise a restaurant all over town, if you like. So the natural parsings for each sentence is different.

- John visited a restaurant on every street.

$$- (\forall x)(Sx \supset (\exists y)(Ry \And Oxy \And Vjy))$$

Here, the ‘every street’ is dominant in logical scope, and for each street, there is a restaurant on that street that John visits.

On the other hand, for the case of advertising, we have a different structure.

- John advertised a restaurant on every street.
- $(\exists x)(Rx \And (\forall y)(Sy \supset Ajxy))$

Here, there is some restaurant, and for each street, John advertised the restaurant on that street.

FOR YOU: Explain the different ways you could interpret the sentence

- At most two students passed each exam.

Do you think that one is more likely as the meaning of the sentence? Explain your answer.

I think that the first is *slightly* more natural, though I can easily hear the second interpretation as a possible meaning of this sentence, too. What do you think? Can you find another possible meaning of this sentence? Discuss your answer in the forums in Courseera.

- ANSWER: Two possible meanings for this sentence are
1. $(\forall x_1)(\forall x_2)(\forall y)((\forall z)(P_{xz} \And P_{x_1z} \And P_{x_2z}) \subset (x_1 = x_2 \And x_1 = x_3 \And x_2 = x_3))$ (There are at most two students such that these students passed each of the exams.)
 2. $(\forall y)(\exists j)((\forall x_1)(\forall x_2)(\forall z)(P_{x_1z} \And P_{x_2z} \And P_{x_3z}) \subset (x_1 = x_2 \And x_1 = x_3 \And x_2 = x_3))$ (For each exam, there are at most two students who passed that exam.)

- Every incarcerated prisoner in California is housed by some penal institution.

with the following sentence—just using the passive-to-active transformation we have seen:

- Some penal institution houses every incarcerated prisoner in California.

The result is a sentence which means something very different. We have gone from the first formula here to the second:

- $(\forall x)((Ix \ \& \ Cx) \supset (\exists y)(Py \ \& \ Hxy))$
- $(\exists y)(Py \ \& \ (\forall x)((Ix \ \& \ Cx) \supset Hxy))$

In the first, for each Incarcerated prisoner in California, there is some Penal institution that Houses them. For the second, there is some Penal institution, where every Incarcerated prisoner in California is Housed. The result is something quite different—there is one prison holding all of California's prisoners.

UPSHOT: ‘ $\alpha \Phi$ s β ’ does not *always* mean the same thing as ‘ β is Φ d by α ’ — especially when there are quantifiers in α and in β and the relative scopes change from one to the other.

In each of these examples, we've tried to give you an account of some of the different ways logic has been applied to phenomena of meaning in language—especially featuring quantifiers. Scope ambiguities show that the intermediate step of translating sentences from a natural language like English into a logical representation is useful if we want to give a clear interpretation. A non-ambiguous representation system is a useful fixed point when isolating ambiguities. These connections can help bridge some disparate fields across linguistics:

[LOGIC] How do quantifiers contribute to argument forms? When are they true?

[FORMAL LINGUISTICS] How do we represent and distinguish their meanings? What readings are preferred and why?

[PSYCHO-LINGUISTICS] How do people use and interpret quantifiers and reason with them? How do we learn them?

[COMPUTATIONAL LINGUISTICS] How do we implement computer systems to correctly interpret quantifiers?

SEQUENTIAL DIGITAL SYSTEMS

9

WHAT YOU WILL LEARN IN THIS CHAPTER

CONCEPTS

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

- Combinational digital systems are built from AND, OR and NOT gates, and are those systems describable in propositional logic.
- Sequential digital systems extend combinational systems with the addition of a digital clock, memory components, internal state signals and feedback loops within circuits.
- Elementary 1-bit memory component is called the *D flip-flop* which delays by one clock tick.
- State-transition/output tables for sequential digital systems.
- Linear Temporal Logic (LTL) with *Next*, *Previous*, *Always* and *Eventually*: syntax, and semantics based on valuations mapping atoms to digital signals.
- Specification of desired behaviour and description of internal workings of sequential digital systems using LTL.
- Formal verification that a system satisfies its specification in LTL.

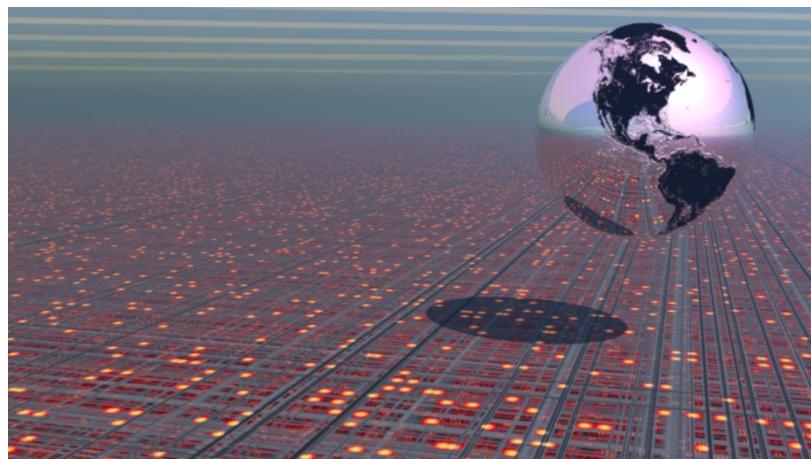
SKILLS

- Complete a simulation table of values for the state signal(s) and output signal(s) of a sequential digital system in response to a given input signal initial segment.
- Translate a functional description of input-output behaviour into a specification formula of LTL.
- Read a circuit diagram for a sequential digital system, and write LTL formulas expressing each next-state signal and output signal in terms of current-states and current inputs.
- Given a state-transition/output table for a sequential digital system, write LTL formulas expressing each next-state signal and output signal in terms of current-states and current inputs.
- Given an initial segment of a run of a sequential digital system together with a time point in that run, identify LTL logic formulas true at that time point under the given valuation.
- Apply the semantics of LTL to determine whether a formula is a *validity* of the logic.

9.1 | DIGITAL SIGNALS AND SYSTEMS

9.1.1 | INTRODUCTION

Through more than 50 years of the digital revolution, we have all witnessed sweeping changes to most aspects of our social and personal lives in virtue of digital computing and communication technology. An educated person in the 21st century needs to have not only the skills needed to use digital technology, but also a basic understanding of how it works. Building on a knowledge of core logic, this sequence of lessons aims to deliver such an understanding.



In this first section, we will begin by reviewing the relationship between propositional logic and *combinational* digital systems, which is the basic class of systems built using AND, OR and NOT gates as the elementary components. For further review, you can go back to the video lessons and course notes from **LLI 1 Chapter 3 Combinational Digital Systems**. We will refer to some topics from that chapter, but also summarise what is needed, so that *mostly*, the present chapter is self-contained, and assumes only the core content from **LLI 2**.

Our key task in this first section is to identify the *additional* elements needed to construct the larger class of *sequential* digital systems. Virtually all computer hardware, from the tiniest embedded microcontroller through to the multiple CPU cores of a top-shelf supercomputer, are a composition of sequential and combinational components. Beyond those within combinational systems, the additional elements in sequential systems are *memory*, *feedback*, *state signals* and *time*, in the form of a digital clock. We will spend several lessons deepening our understanding of each of these elements, and we will be using logic at each step of the way.

The logic we will use in both describing digital systems and specifying their desired behaviour is called *temporal logic*. In terms of logical strength and expressivity, it can be thought of as intermediate between propositional logic and predicate logic. Temporal logic is customised for reasoning about change over time. It includes operators that behave like the predicate logic quantifiers *for all* and *there exists*, but in

temporal logic they refer to time, with the meanings *for all time in the future* and *at some time in the future*. Crucial for the purpose of describing and specifying sequential systems, temporal logics also include one-step operators *Next* and *Previous*, which give a means to refer to one time step ahead and one time step behind. These two logic operators correspond to some elementary operations on signals: the *shift operators* that move a signal one time step back or one time step forward. Somewhat counter-intuitively, the shift one time step *back* corresponds to the *Next* temporal operator, while the shift one time step *forward* corresponds to the *Previous* temporal operator. This will make more sense with examples. Developing step-wise through this series of five lessons, we will build up to the point of being able to formally verify using temporal logic that (an admittedly small) sequential system satisfies its specification on all possible inputs.

9.1.2 | DIGITAL SIGNALS AND SYSTEMS

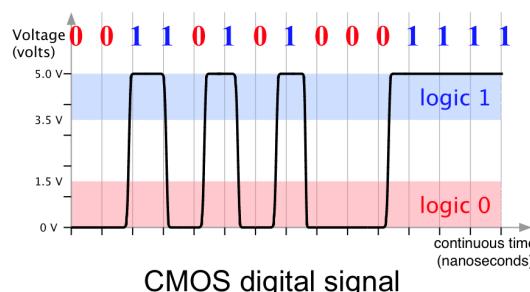
First, digital signals. The *Binary* or base-2 numbers are just 0 and 1. The word “bit” is a contraction of “binary digit”.

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

Here is an initial segment of a digital signal:

n	0	1	2	3	4	5	6	7	8	9	10	11	12	13	...
p	0	0	1	1	1	1	0	0	1	1	0	0	1	1	---

This is an initial segment of a digital signal, as it can keep on going forever. In practice, all digital signals are *finite* in length, but we use the mathematical abstraction of an *infinitely long* sequence to capture the idea of having *arbitrarily large finite length*: whatever finite length you pick, the sequence is longer than that. The positions *n* in the sequence mark discrete ticks of a digital clock.



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 real time voltage signal that continuously 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 example,

in the CMOS family of integrated circuits, the maximum voltage is 5 volts, as shown in the graph above. Transitions between the low and high bands occur very rapidly, in much less time than the real time between discrete ticks of the digital clock. It is only at these discrete *clock tick* moments that the low/high band level or 0/1 content of a signal really matters. We will further discuss timing aspects of digital signals and systems later in the next section.

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 MOS-FET transistors if you are interested.

Videos on construction and operation of integrated circuits:

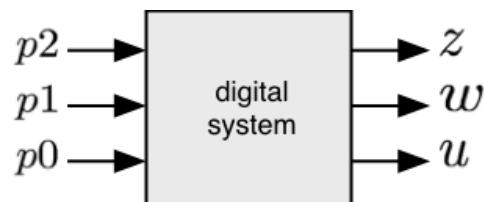
http://www.youtube.com/watch?v=v7J_snw0Eng

<http://www.youtube.com/watch?v=Q05FgM7MLGg>

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

Abstractly, a *system* is an abstract machine with inputs and outputs.

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

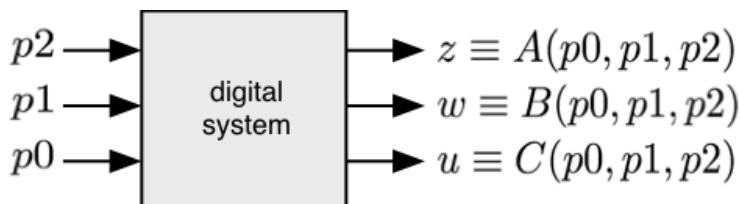


The conventions for representing the input/output *block structure* of a system, and for representing internal components within circuit diagrams, is that *inputs* go in on the *left* and outputs come out on the *right*. Arrow heads are also useful to indicate direction.

9.1.3 | COMBINATIONAL DIGITAL SYSTEMS

Definition: A *combinational digital system* is a memoryless digital system such that each output can be expressed as a propositional

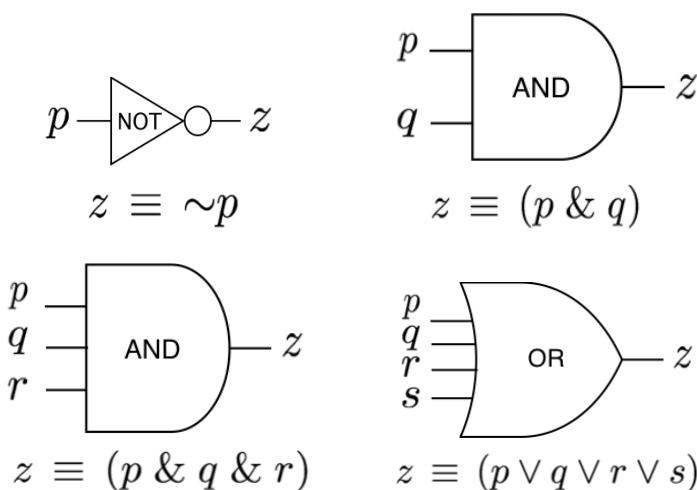
logic formula in terms of the inputs (using only AND, OR and NOT).



Corresponding with *propositional logic*, the sub-class of *combinational* digital systems can be delineated as being those systems such that each output signal can be characterized by a propositional logic formula in terms of the inputs, in the sense that an output has value 1 if and only if its characterising formula is true. In our sample, output z is characterised by a formula A built from atoms p_0 , p_1 and p_2 ; and likewise formula B for output w and formula C for output u . For combinational digital systems, we can restrict to characterising logic formulas A , B , C containing only the connectives AND, OR and NOT. The *memoryless* aspect of combinational systems is that their current output depends only on their current input, and not on any remembered, previous values of their input signals.

The propositional connectives AND, OR and NOT correspond precisely to the basic building blocks within combinational systems.

Starting with the NOT gate, it is a basic 1-input, 1-output system that inverts the value in the input signal p to produce output z such that $z \equiv \sim p$.



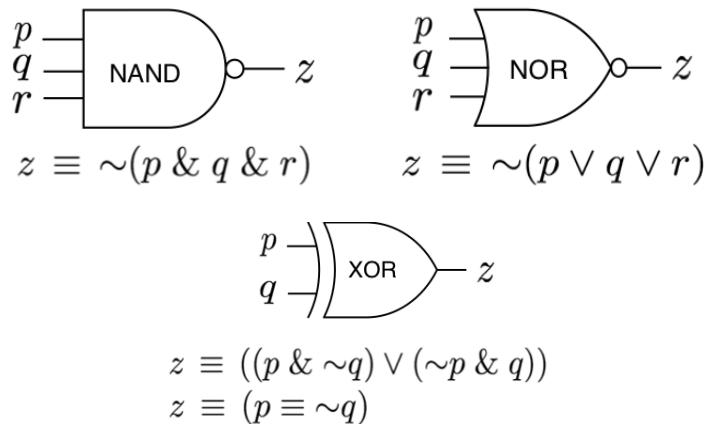
An AND gate is a basic system with 2 or more inputs and 1 output such that the output is 1 if and only if all of the inputs are 1.

An OR gate is a basic system with 2 or more inputs and 1 output such that the output is 1 if and only if at least one of the inputs is 1. Every combinational system can be built from AND, OR and NOT gates, with the output of one gate fed into the input of another.

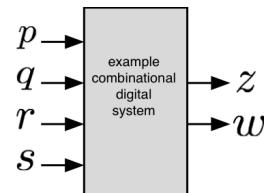
Further combinational components include:

- **NAND** gates with 2 or more inputs and 1 output (the 2-input connective originally known as the *Sheffer stroke*; Sheffer: 1913);
- **NOR** gates with 2 or more inputs and 1 output (originally the *Peirce arrow*; Peirce: 1881); and
- the 2-input *exclusive OR* or **XOR** gate, whose output z is 1 if and only if the two inputs p and q have opposite bit values.

NAND gates, and separately NOR gates, have the *universal* property of being able to implement all other logic gates. Practically, this means combinational digital circuits can be built using only one type of component. The XOR gate often appears in combinational circuits implementing arithmetic operations, because XOR behaves the same as 1-bit addition base 2.

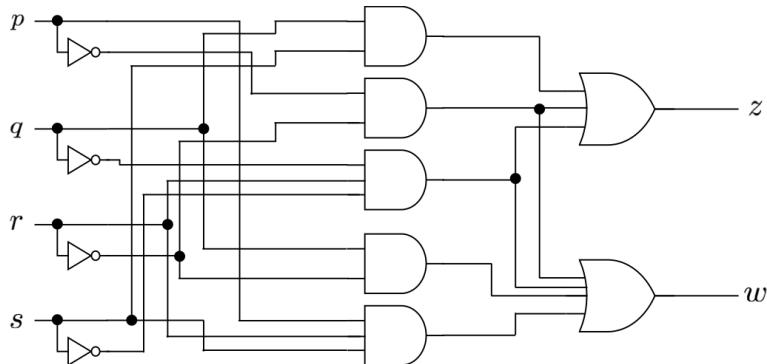


Example: Consider a combinational digital system with four 1-bit input signals p , q , r and s , and two 1-bit output signals z and w such that:

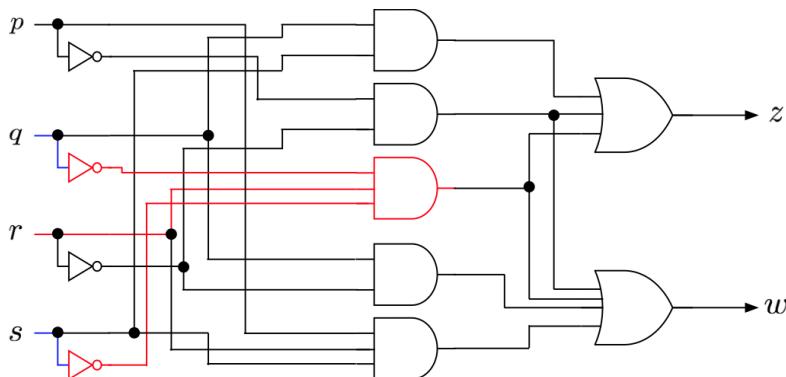


- The output z is active (has value 1) if and only if either both p and r are inactive, or both q and s are active, or else both q and s are inactive at the same time r is active.
- The output w is active if and only if either both p and r are inactive, or q is active while r is inactive, or both q and s are inactive at the same time r is active, or else p , r and s are all active together.

This system is implemented in the circuit diagram as shown below.



This circuit is in a standard form known as *disjunctive normal form* or *DNF*; it is also known as *sum-of-products* form in the Boolean algebra-based digital systems literature (see **LLI 1** §3.3.3).



Exercise: The AND gate highlighted in red is described by which propositional logic formula:

- (a) $(\sim q \ \& \ r \ \& \ s)$ (b) $(\sim q \ \& \ r \ \& \ \sim s)$ (c) $(\sim q \ \& \ \sim r \ \& \ s)$

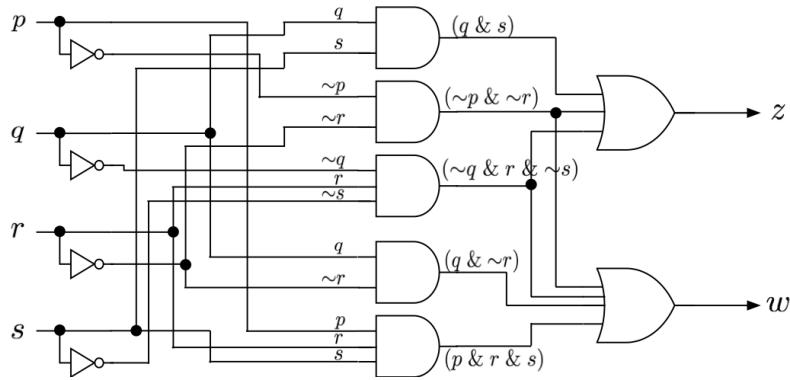
$(s \sim \otimes r \otimes b \sim) (q)$

disjunctive logic formula:

The AND gate highlighted in red is described by which proposi-

Since the AND gate red is shared by both outputs z and w , it being active will activate both of them. This branching or splitting is called a *feed-forward* connection of signals, and stands in contrast with *feedback* connections of signals, which allow signal paths that start from the output of a component and lead around in a loop and back into the input of the *same* component. Combinational systems do not allow feedback, but sequential systems do. We will see feedback in action very soon.

By labelling all the intermediate signal wires, it is easy to *read off* from the circuit characteristic formulas for each of the two outputs, where those formulas are in DNF.

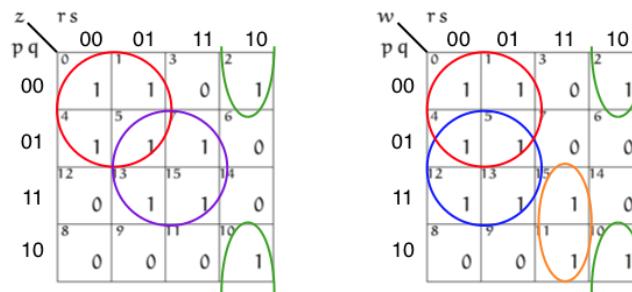


If we go back to the natural language functional specification for the system from a few slides ago, we can see that the three conditions for z being active in the functional specification correspond exactly to the three AND gate outputs leading into the OR gate for z . Likewise, the four conditions for z being active in the functional specification correspond exactly to the four AND gate outputs leading into the OR gate for w , with two out of the total of five AND gates being shared by the two outputs.

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

$$w \equiv ((\sim p \& \sim r) \vee (\sim q \& r \& \sim s) \vee (q \& \sim r) \vee (p \& r \& s))$$

For this combinational system - and the same is true for many combinational systems - with its system description in the form of the circuit diagram and characteristic logic formulas, it is easy to see that the system description is in fact *logically equivalent* to the functional specification. This means that it is relatively easy to *formally verify* in logic that this system satisfies its specification. As we shall see throughout this series of lessons, formal verification in logic for sequential systems takes quite a lot more effort.



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

$(\sim q \& r \& \sim s)$

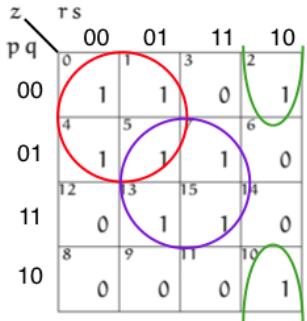
$(q \& \sim r)$

$(q \& \sim r)$

$(p \& r \& s)$

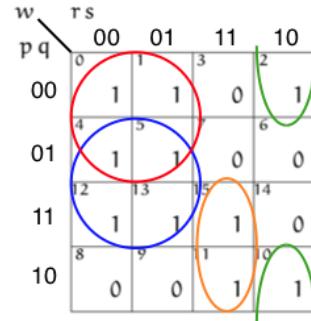
Back with our example combinational system, each of the AND gates in the circuit correspond to a “loop” of 1s in the K-maps (Karnaugh maps) for the system, as shown.

By analysing the loops in the K-maps, we can prove that the circuit as shown is a *minimal* DNF circuit for the system: it has the smallest number of AND gates (5) and the smallest total input-count across those AND gates (12). We will continue to need to find minimal DNF formulas for combinational sub-systems within sequential systems, so this extent, we are assuming familiarity with K-maps. However, the combinational minimization will be a side issue, and can also be understood as reasoning through a sequence of logical equivalences, without reference to K-maps.



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

$$w \equiv ((\neg p \& \neg r) \vee (\neg q \& r \& \neg s) \vee (q \& \neg r) \vee (p \& r \& s))$$



Combinational digital systems can be described in a multitude of ways. In stepping up to sequential systems, we assume familiarity with the ones in **blue** text:

- functional descriptions of specified input/output behaviour;
- propositional logic formulas;
- Boolean algebra equations;
- circuit diagrams with AND, OR and NOT gates;
- circuit diagrams with NAND gates (or NOR gates) only;
- input/output truth tables;
- Karnaugh maps (K-maps);
- binary decision diagrams (BDDs);
- hardware description language code (e.g. VHDL or VERILOG).

p	q	r	s	z	w
0	0	0	0	1	1
0	0	0	1	1	1
0	0	1	0	1	1
0	0	1	1	0	0
0	1	0	0	1	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	1	1	0
1	0	0	0	0	0
1	0	0	1	0	0
1	0	1	0	1	1
1	0	1	1	0	1
1	1	0	0	0	1
1	1	0	1	1	1
1	1	1	0	0	0
1	1	1	1	1	1

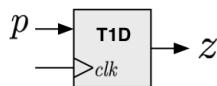
If you have not covered the material in Chapter 3 **Combinational Digital Systems** from LLI 1, then at this stage we recommend you just continue on ahead and see how you go. If it turns out you have difficulty with some combinational aspect of circuit diagrams or logic

formulas describing outputs in terms of inputs, then go back to the earlier material to review from it what you need.

9.1.4 | EXAMPLE SEQUENTIAL SYSTEM: TRIPLE 1 DETECTOR

Without further ado, we consider an example of a sequential system. Starting with its functional input/output specification, the *Triple 1 Detector* (TID) system has one 1-bit input signal p and one 1-bit output signal z such that for all times n :

The output z has value 1 at time n if and only if
input p was 1 at time $(n-1)$ and p was also 1 at time $(n-2)$
and p was also 1 at time $(n-3)$.



That is, z must be active immediately after the input p has been 1 for the *previous three clock ticks*. To indicate that this is a *synchronous* sequential system, we need to include a *clock* input as well as the input signal p and output signal z in the block diagram.

Clearly, *memory* of past input values is involved in this system. Further paraphrased, the output z of the TID system must be active immediately after its input p has been 1 for *three consecutive clock ticks*.

With sequential systems, a good way to get a feel for a system is to take it for a “run” by simulating it on a sample input signal. We can describe a simulation or run in a table indexed by time n .

Here is one possible run of the TID system, according to specification:

n	0	1	2	3	4	5	6	7	8	9	10	...
p	1	1	1	0	1	1	1	1	0	1	0	---
z	0	0	0	1	0	0	0	1	1	0	0	---

In this simulation, the input signal p that starts with three 1s in a row, at discrete time points $n = 0, 1, 2$, then flips to 0 at $n = 3$. It then continues with another four 1s in a row, as shown. Now, if the output z is produced according to the specification, we expect that z will be 0 for the first three time points $n = 0, 1, 2$. Then at time $n = 3$, we get z is 1 because looking back we see p is 1 at time $(n-1) = 2$, at time $(n-2) = 1$, and at time $(n-3) = 0$. At the next time $n = 4$, z goes back to being 0 and stays 0 until time $n = 7$, when z is 1 because looking back, the previous three input values for p were 1, at times $n = 6, 5, 4$. Output z stays 1 at the next time $n = 8$ because p is also 1 at times $n = 5, 6, 7$. Then after that, at time $n = 9$, z goes back to 0.

Another possible run of the TID system according to specification:

n	0	1	2	3	4	5	6	7	8	9	10	...
p	0	1	1	1	1	1	0	0	1	1	0	---
z	0	0	0	0	1	1	1	0	0	0	0	---

You can step through this second run yourself, seeing why output z becomes 1 at time $n = 4$, and stays 1 until $n = 7$, when it drops back to 0.

The functional specification of input/output behaviour is required to be satisfied at *all* time points n . This universal quantification over n allows us to do a substitution, replacing n with $(n + 3)$, to reformulate the specification as follows:

The output z has value 1 at time $n + 3$ if and only if
 p is 1 now at time n and p is also 1 next time at time $(n + 1)$
 and p is also 1 at the next-next time $(n + 2)$.

Since the output z cannot be 1 until at least time $n = 3$, this shift by 3 time steps still captures the intended meaning that z is 1 immediately after the input p has been 1 for three consecutive clock ticks. This *future-focused* rather than *past-focused* way of stating the specification is the most common approach within *temporal logic*.

Looking ahead to temporal logic, we introduce the *Next* operator on signals, that *looks forward* one time step. We use the \oplus symbol for *Next*. At time n , the value of the signal $\oplus p$ is equal to the value of signal p at time $(n + 1)$, one time step *ahead*.

$$\square (\oplus \oplus \oplus z \equiv (p \& \oplus p \& \oplus \oplus p))$$

This future-focused version of the specification for system TID says: **for all time points n , $\oplus \oplus \oplus z$ is 1 if and only if p is 1 now, $\oplus p$ is 1, and $\oplus \oplus p$ is also 1**. The temporal operator \square means “*always*” or “*at all times*”.

The past-focused version of the specification requires the *Previous* operator on signals, that *looks backward* one time step. We use the \ominus symbol for *Previous*. At time n , the value of the signal $\ominus p$ is equal to the value of signal p at time $(n - 1)$, one time step *behind*, with $\ominus p$ *undefined* at time $n = 0$.

$$\square (z \equiv (\ominus p \& \ominus \ominus p \& \ominus \ominus \ominus p))$$

The past-focused version of the specification for system TID says: **for all time points n , z is 1 now at time n if and only if $\ominus p$ is 1 and $\ominus \ominus p$ is 1 and $\ominus \ominus \ominus p$ is also 1**. In §9.4, when we formalise the syntax and semantics of temporal logic, we will see how to prove these two versions of the specification for system TID are logically equivalent.

9.1.5 | SIGNALS AND THEIR SHIFTS

The *Next* and *Previous* temporal operators correspond to two elementary operations on signals: the *shift operators* that move a signal one time step back or one time step forward. As we warned in the introduction, the correspondence is somewhat counter-intuitive in that, as *operations on signals*, they do the *opposite*.

The *Next* operator \oplus on signals *predicts the future* by shifting the whole signal *backward* by one time step; $\oplus p$ is the result of *backward-shifting* signal p by one time step:

n	0	1	2	3	4	5	6	7	8	9	10	...
p	1	0	0	1	0	1	0	1	1	1	0	---
$\oplus p$	0	0	1	0	1	0	1	1	1	0	---	

Check this works: at each time n , $\oplus p$ has the same value as p at time $n+1$. As an operation on signals, it is a *backward* time translation of the whole signal.

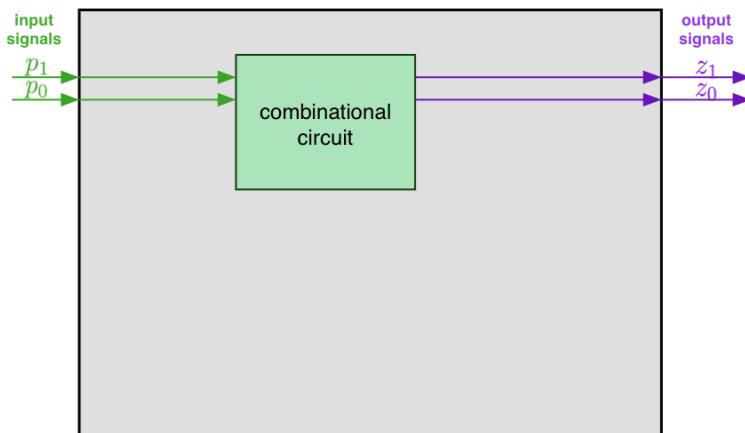
In contrast, the *Previous* operator \ominus on signals *remembers the past* by shifting the whole signal *forward* by one time step, leaving it undefined at $n = 0$; $\ominus p$ is the result of *forward-shifting* signal p by one time step:

n	0	1	2	3	4	5	6	7	8	9	10	...
p	1	0	0	1	0	1	0	1	1	1	0	---
$\ominus p$	_	1	0	0	1	0	1	0	1	1	1	0---

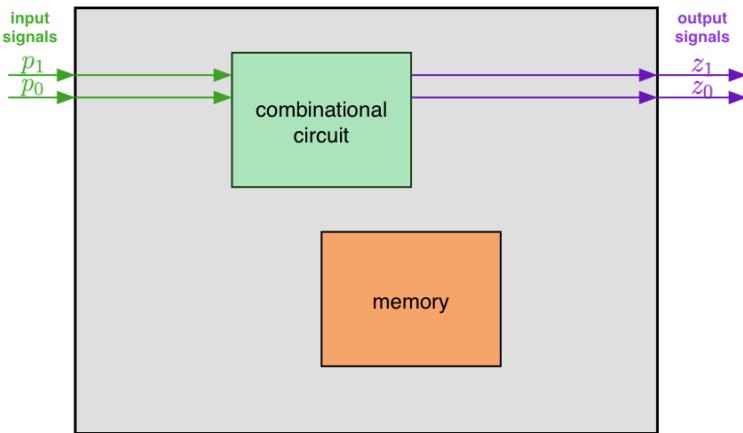
where $_$ means “has no value”. Again, check this works: at each time n , $\ominus p$ has the same value as p at time $n-1$, except for when $n = 0$, when $\ominus p$ has no value. Under the semantics of temporal logic, the formula $\sim \ominus p$ will be true at time $n = 0$ and $\ominus p$ will be false at time $n = 0$, for all signals p . In this sense, we can think of the signal $\ominus p$ as taking a *default* value of 0 at time $n = 0$. This also serves as advance warning that in temporal logic, we will be working with a semantics that has formulas true or false *at each time point* n in a model that assigns a *digital signal* to each atomic proposition p .

9.1.6 | THE STRUCTURE OF SEQUENTIAL SYSTEMS

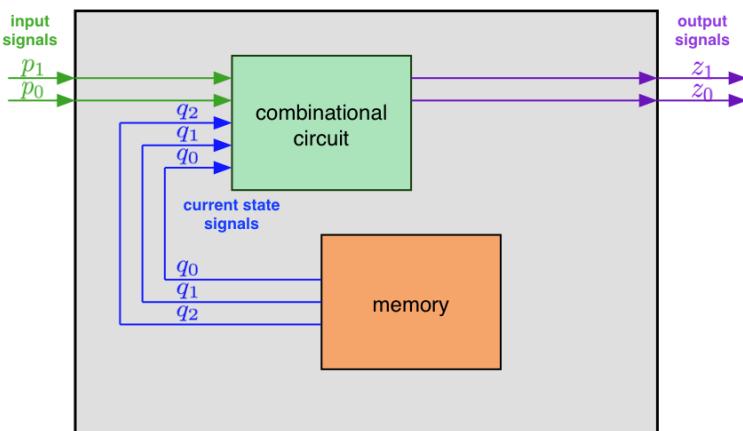
In the build-up to sequential digital systems, we start with a combinational sub-circuit:



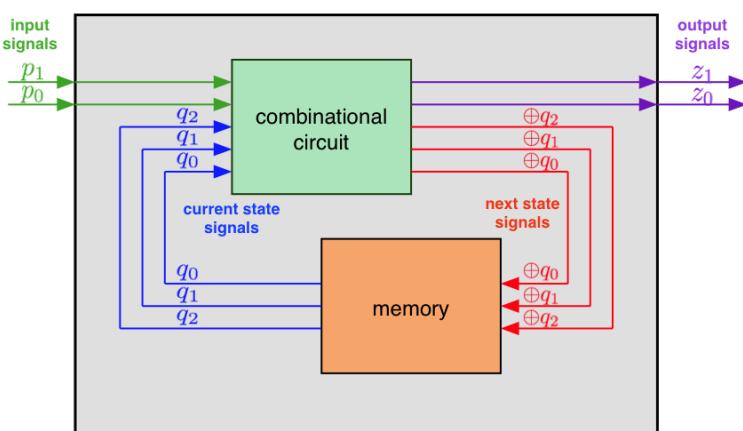
We then add a *memory* component. We shall learn about memory components in the next section, §9.2, starting with the elementary 1-bit memory components called *flip-flops*.



The memory has to be *read-from* or output-accessed, so there are signal wires coming *out of* the memory component and into the combinational sub-system. The signals q_i coming out of the memory component are called *state signals*, and they are signals internal to the system and in addition to the input and output signals.

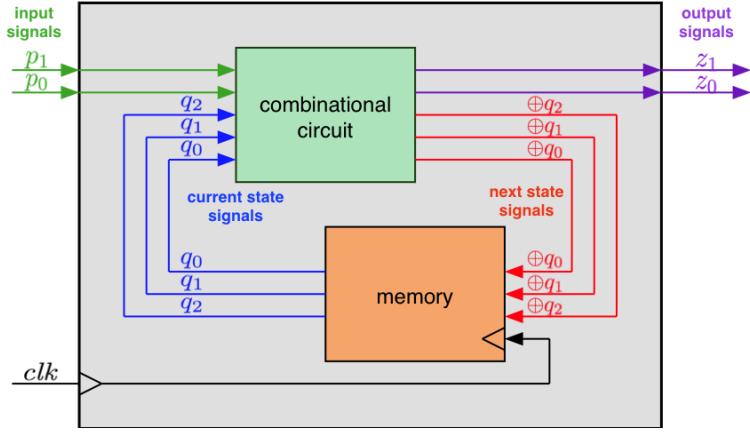


The memory also has to be *written-to* or input-accessed, so there are signal wires leading *into* the memory component and out of the combinational sub-system.



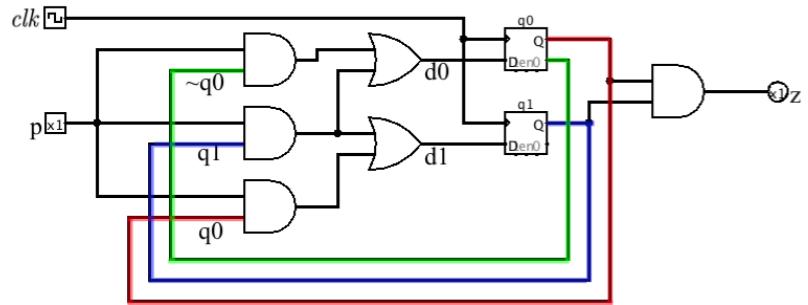
The result is a *feedback loop* inter-connecting the memory component and the combinational sub-system. The signals leading *into* the memory component are the *Next-state* signals $\oplus q_i$. This is because after one

tick of the digital clock, *Next-state* becomes *current-state*. The *Next-state* signals $\oplus q_i$ are a combinational function of the *current-state* signals q_i and the current input signals p_j . The memory component *remembers* the content of each of the state signals for one tick of the digital clock. So we need to make the clock explicit: the digital clock is an additional input to the *memory* component since this is the part of the system where time and synchronicity really matter.



We have now assembled all the pieces needed to set out the general structure of circuit diagrams for *sequential digital system*.

Going back to our running example, the Triple-1 detector system TID, here is a circuit diagram for that system.



The next-state and output equivalences for TID described in this circuit:

$$\begin{aligned}\oplus q_1 &\equiv d_1 \equiv ((p \& q_1) \vee (p \& \sim q_0)) \\ \oplus q_0 &\equiv d_0 \equiv ((p \& q_1) \vee (p \& q_0)) \\ z &\equiv (q_1 \& q_0).\end{aligned}$$

By the end of §9.3, we will understand how such a circuit diagram is designed by deriving logic formulas characterising the next-state and output signals in terms of the current state and input signals.

Sequential digital systems can be described in many ways.

- functional descriptions of specified input/output behaviour;
- circuit diagrams with combinational components, memory components, digital clock signal and feedback;

- linear temporal logic formulas characterising next-state and output behaviour;
- linear temporal logic formulas specifying required input/output behaviour;
- state-transition/output table;
- finite state machine (FSM) transition diagrams;
- hardware description language code (e.g. VHDL, VERILOG).

From this list, we will see and use those in blue and red text. We have seen a functional description of specified input/output behaviour already, for the TID system, and we will learn how to read and design sequential circuit diagrams, with combinational components, memory components, digital clock signal and feedback.

We will use temporal logic formulas for *system description*, to express the internal workings by characterising the next-state and output behaviour in terms of the current state and current input, like the ones we have given for the TID system.

We will also use temporal logic formulas for *system specification*, to formally express the required external input/output behaviour. We have had a first look at system specification formulas for the TID system, in both future-focused and past-focused versions. The fragment of temporal logic we need will be discussed and formalised a little later, in §9-4. In contrast with combinational systems, even quite small sequential systems in general require some effort to formally verify in logic that the specification formula is a logical consequence of the system description formula.

A further way we will describe sequential systems is via their *state-transition/output table*, which sets out the truth-functional dependence of the next-state signals and the output signals on the current state and current input – in much the same way that a truth table sets out the truth-functional dependence of the output of a combinational system on the inputs of that system. As we shall see in §9.3, a state-transition/output table is an essential tool in the process of designing a sequential system in a manner that guarantees enforcement of the desired specification.

Among the multitude of *other* ways to describe sequential digital systems, we mention two that are prominent in the digital system literature but we do not have time to discuss them. These are *finite state machine* (FSM) transition diagrams; and *hardware description language* code, using languages such as VHDL or Verilog.

9.2 | TIME, MEMORY AND FLIP-FLOPS

9.2.1 | DIGITAL TIME AND DIGITAL CLOCKS

In a digital signal $(p_n)_{n \in \mathbb{N}}$, the sequence index numbers $0, 1, 2, 3, \dots$ in \mathbb{N} are thought of as *discrete time points*, marking discrete “ticks” of a

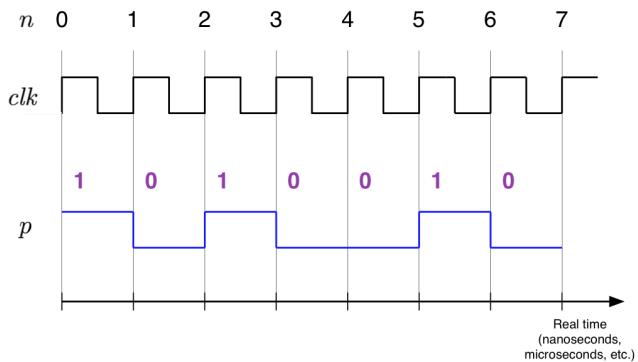
digital clock.

The *frequency* of a digital system is measured in the units *Hertz*, which means *cycles per second*. For example, a frequency of 1 GHz means 1 billion digital clock ticks per second of real time.

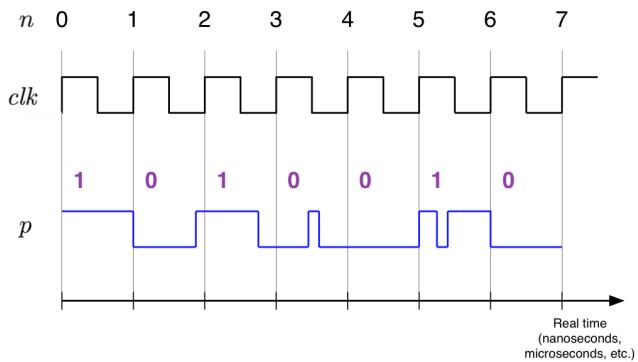
The real-time *period* of a digital clock cycle is the reciprocal of the frequency, so a 1 GHz system has a clock period of 1-billionth ($\frac{1}{10^9}$) of a second, which is 1 nanosecond.

A clock signal is visually represented as a square-edged sine-wave. We will be working with systems where the clock *tick* point occurs at the *positive-going* transition from 0 to 1. (Dually, one can also set up systems in which the clock tick point occurs at the negative-going transition from 1 to 0.) An actual clock signal in a physical implementation cannot be a true square-wave because a physical signal cannot transition from 0 to 1 in zero time. A physical clock signal is a smoothed version of the square-edged ideal, in which the time taken for the transition event is *much shorter* than the clock period.

Within synchronous sequential systems, a data signal like the *p* signal shown is only *read* or *sampled* at the clock tick times: it is only the value of *p* at the clock tick times that matters. This is straight-forward when the signal is synchronous with the clock, meaning it only changes value only at clock tick points.



Now consider a data signal *p* that is *asynchronously* generated, such as when it is sourced externally from human input or from some physical sensor.

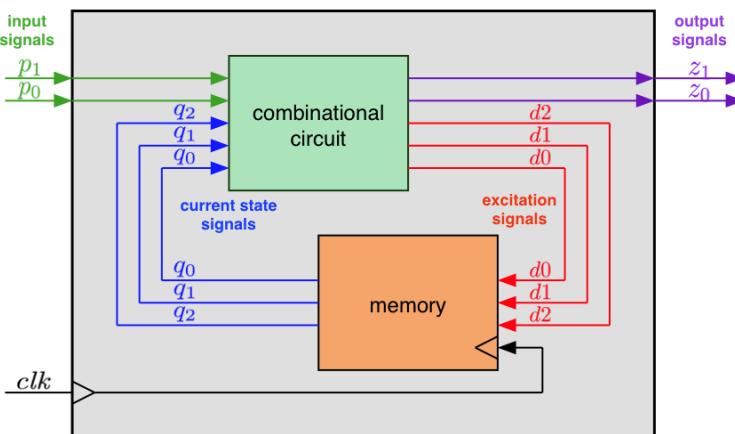


No matter the signals origin, once it is inside a *synchronous* sequential system, it is read or sampled only at clock tick times. So any signal fluctuations between clock ticks are ignored.

For the correct behaviour of memory components, there are some real-time constraints: the *setup time* is the minimum amount of time the data signal should be held steady *before* the clock tick event so that the data are reliably sampled, while the *hold time* is the minimum amount of time the data signal should be held steady *after* the clock tick event so that the data are reliably sampled. These times are specified in the data sheet for the device, and for modern devices are typically between a nanosecond and a hundred picoseconds (between 10^{-9} and 10^{-10} of a second). These times are always *much smaller* than (in the order of $\frac{1}{100}$ of) the real-time clock *period*.

9.2.2 | FLIP-FLOPS AS MEMORY DEVICES

Coming back to the general structure of synchronous sequential digital systems, it is now time to look inside the memory component and see what is there. As a first step, we re-label the input signals to the memory component to call them *excitation* signals d_0, d_1, d_2 , etc. After we have worked through an analysis of how these memory components work, we will arrive back at the conclusion that each excitation signal d_i is equivalent to $\oplus q_i$, the *Next* of the corresponding state signal q_i .



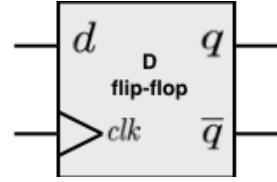
Most people think of flip-flops as cheap rubber footwear . But flip-flops are also *memory devices*.



The simplest 1-bit memory component is called a *flip-flop* because it is a device that stabilises in one of two states, one representing 0 and the other representing 1, and it *flips* and *flops* between them. (A *three-valued* or *tri-state* device has been called a *flip-flap-flop*!) A flip-flop is also called a *bistable multivibrator*.



We introduce the simplest 1-bit synchronous memory component: a *D-type type flip-flop* which is *positive edge-triggered*, as we are working with *positive-going* clock tick events. The D stands for *data* and also for *delay*.



The system has a 1-bit input d for data, and a clock signal input for synchronisation; on the output side, it has two 1-bit outputs called q and \bar{q} , known as *state signals*.

d	clk	$\oplus q$	$\oplus \bar{q}$
0	↑	0	1
1	↑	1	0

The device works as follows. At each clock tick event (positive transition \uparrow), the value in the d signal is passed to the $\oplus q$ signal, with the result that the state signal q always holds a 1-step *delayed* copy of the input data signal d . The system is designed so that the second output \bar{q} is equivalent to $\sim q$, the *negation* of q .

Three formulas of temporal logic characterise the behaviour of the D flip-flop:

$$D_1 : \square(\oplus q \equiv d) \quad D_2 : \square(\bar{q} \equiv \sim q) \quad D_3 : \square(q \supset \ominus T)$$

Reading the outer \square as *always*, the first formula D_1 says the next-state signal $\oplus q$ is always equivalent to, or has all the same values as, the data input d . In Boolean algebra-based treatments of digital systems, this is known as *the characteristic equation* for D flip-flops.

The second logic formula D_2 says \bar{q} is always equivalent to the negation of q .

The third formula D_3 expresses that at time $n = 0$, the state signal q has the default value of 0. The symbol T or *Top* is a logical constant which is read as *True* and which denotes the signal that is constantly 1. The formula $\ominus T$ is true exactly when time n is greater than 0. So the conditional $(q \supset \ominus T)$ says that signal q has value 1 *only if* time n is greater than 0. The formula's equivalent contrapositive, $(\sim \ominus T \supset \sim q)$ says that if time $n = 0$ (because $\sim \ominus T$ is true), then $\sim q$ has value 1, so q has value 0.

In §9.4, when we formalise the fragment of temporal logic needed for our purposes, we shall discover that the conjunction of D_1 and D_3 is logically equivalent to the following formula D_0 :

$$D_0 : \square(q \equiv \ominus d) \quad D_{20} : \square(\bar{q} \equiv \sim \ominus d)$$

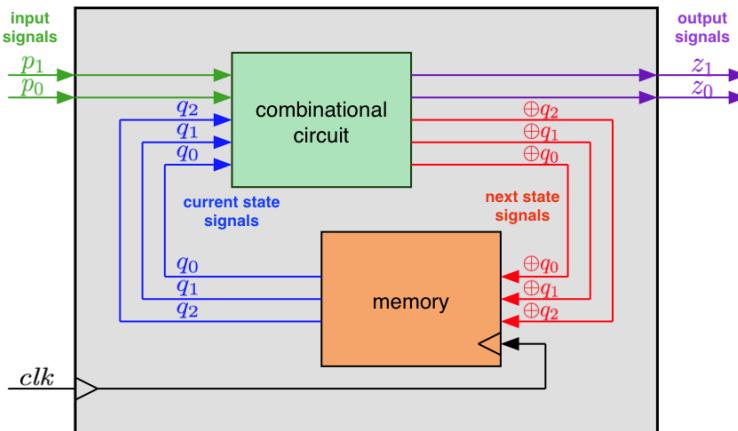
The formula D_0 is more in parallel with the combinational gates and their logic descriptions, in that the output q is expressed as the result of *doing something* to the input d . The *doing something* of a D flip-flop is to *delay* by one clock tick. So the output, the state signal q , is always equivalent to $\ominus d$.

The second formula D_{20} is a logical consequence of D_2 in conjunction with D_0 ; it says: \bar{q} is always equivalent to the negation of $\ominus d$. This means \bar{q} is 1 exactly when either $\ominus d$ is 0 or time $n = 0$.

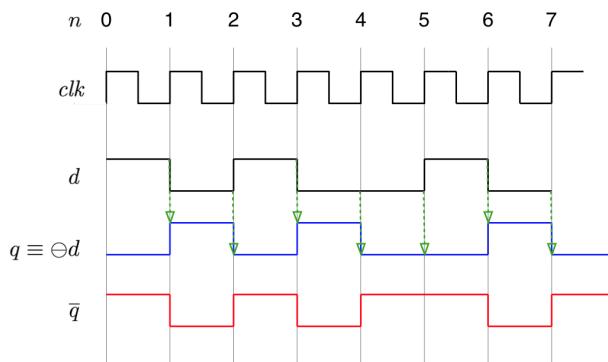
In the light of this analysis of D flip-flops, in particular the characteristic formula:

$$D_1 : \square (\oplus q \equiv d)$$

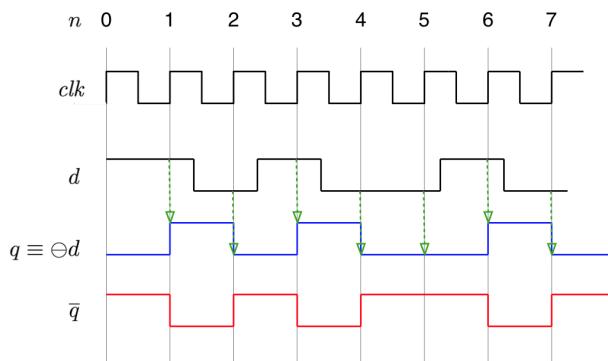
we can now arrive at the conclusion that, within the general structure of sequential systems, each excitation signal d_i is equivalent to $\oplus q_i$, the *Next* of the corresponding state signal q_i .



Timing diagrams show a D flip-flop in action. When the input to a D flip-flop is a *synchronous* signal (changing only with the clock), then the state signal q is simply the 1-step delay or forwards shift of d .

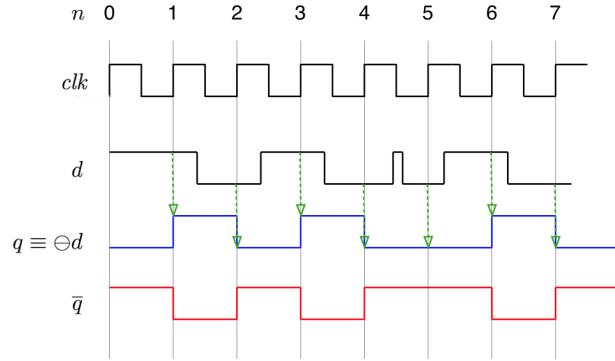


When the input to a D flip-flop is not synchronous with the digital clock, it is its value at the clock tick times that is read and processed.



So even if there are signal fluctuations between clock ticks, these are ignored – provided signal fluctuations occur away from the clock tick

event times, so the *setup time* and *hold time* conditions are met, as these ensure the correct sampling of the data signal d at clock tick times.



Now, an exercise for you, to check your understanding of the D flip-flop. Study these three different d input signal initial segments and responses from output signals q and \bar{q} .

Which of these are possible runs of a D flip-flop, satisfying:

$$\begin{array}{ll} A_0 : (q \equiv \oplus d) & A_1 : (\oplus q \equiv d) \\ A_2 : (\bar{q} \equiv \sim q) & A_3 : (q \supset \ominus T) \end{array}$$

at all time points n (here, for $n = 0, 1, 2, \dots, 10$)?

n	0	1	2	3	4	5	6	7	8	9	10	...
(a)	d	1	0	0	1	1	0	0	1	0	0	1
	q	0	1	0	0	1	1	0	0	1	0	0
	\bar{q}	1	0	1	1	0	0	1	1	0	1	1

n	0	1	2	3	4	5	6	7	8	9	10	...
(b)	d	1	1	0	1	0	0	0	1	1	0	1
	q	1	1	1	0	1	0	0	0	1	1	0
	\bar{q}	0	0	0	1	0	1	1	1	0	0	0

n	0	1	2	3	4	5	6	7	8	9	10	...
(c)	d	0	0	1	1	1	0	1	0	0	1	1
	q	0	0	0	1	1	1	0	1	0	0	1
	\bar{q}	1	1	1	0	0	0	1	0	1	1	0

Simulation table option (a) is correct.
Simulation table option (b) is not correct: (i) should start with $q_0 = 0$ and $\bar{q}_0 = 1$; (ii) also has $q_{10} = 0$ when $d_{10} = d_0 = 0$, so should be $q_{10} = 1$.

Simulation table option (c) is correct.

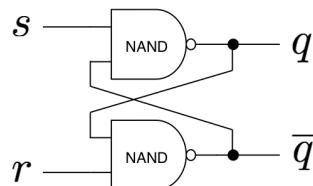
Errors in (b) shown:

9.2.3 | INSIDE A D-TYPE FLIP-FLOP

Next, we take a look *inside* a D-type flip-flop. Our D-type flip-flop can be built from three inter-connected SR NAND Latches. The **SR NAND Latch** is an asynchronous feedback circuit which under *normal* operation simultaneously satisfies:

$$q \equiv \sim(s \& \bar{q}) \quad \text{and} \quad \bar{q} \equiv \sim(r \& q) \quad \text{and} \quad \bar{q} \equiv \sim q$$

as set out in its circuit diagram:



These three formulas cannot be simultaneously satisfied or stabilised when $s = 0$ and $r = 0$, so that configuration is not allowed. In this circuit, the inputs s for SET and r for RESET are *active*, or doing their job, when they have value 0, so they are said to be *active low*.

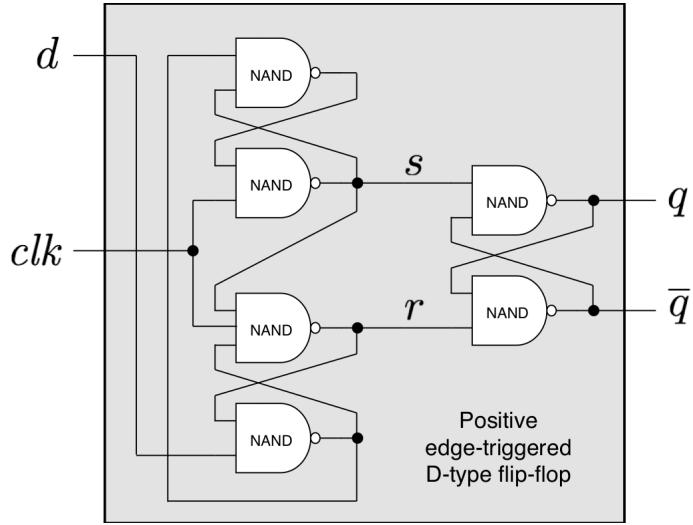
s	r	q	\bar{q}	action
0	1	1	0	set q to 1 (s active)
1	0	0	1	reset q to 0 (r active)
1	1	$\sim \bar{q}$	$\sim q$	no change if $\bar{q} \equiv \sim q$
0	0	1	1	unstable: not allowed

When s is active and r inactive ($sr = 01$), the state q is set to 1, while when r is active and s inactive ($sr = 10$), the state q is reset to 0. When both s and r are inactive ($sr = 11$), there is no change in the value of q provided that \bar{q} is equivalent to $\sim q$. The “no change” case provides the means to “remember” or “hold on to” the values of q and \bar{q} , since:

$$q \equiv \sim \bar{q} \quad \text{if} \quad \bar{q} \equiv \sim q.$$

The disallowed configuration is when s and r are both active low ($sr = 00$), since this results in a violation of $\bar{q} \equiv \sim q$.

The interconnection of three SR NAND latches inside a D-type flip-flop is as follows:

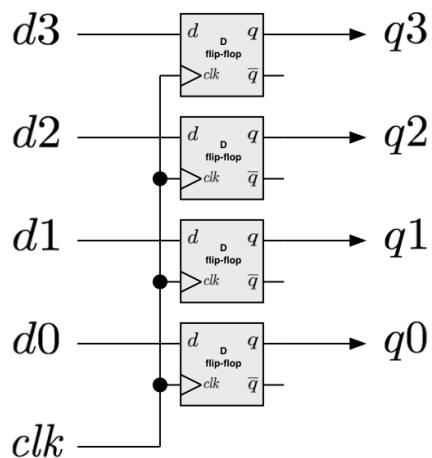


The crucial feature is that the *set* signal *s* and the *reset* signal *r* are *never* both 0; this is what ensures $\bar{q} \equiv \sim q$.

The state signal starts at time $n = 0$ with $q = 0$ and $\bar{q} = 1$, then the clock signal *clk* positive transitions from 0 to 1, causing the value of the excitation input *d* to be transferred to the state signal *q*, so $\oplus q \equiv d$ and $q \equiv \ominus d$. At all other times, *q* and \bar{q} remain unchanged.

9.2.4 | COMBINING FLIP-FLOPS

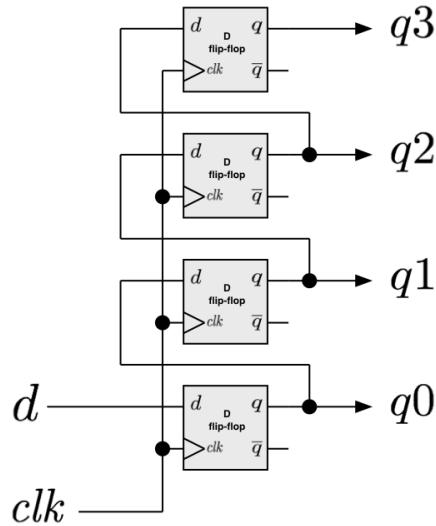
A single 1-bit memory or delay component might not seem like much, but as we shall see later in the rest of this chapter and in the quiz questions for this section of the course, putting just two D flip-flops together allows us to create quite a variety of sequential systems. Within more complex digital systems like the central processing unit of a computer, or the arithmetic-logic unit inside a simple embedded microprocessor, one finds blocks of D flip-flops called *registers*. A 4-bit memory register is configured as shown.



At each clock tick, the current value of the 4-bit signal (*d3* *d2* *d1* *d0*) is loaded into the register, and transferred to the *Next* value of the 4-bit state signal (*q3* *q2* *q1* *q0*). A 64-bit memory register scales up the same,

with a 64-bit *signal bus* (layout of 64 parallel signal wires) on the input side and another leading from the output side.

Flip-flops can also be combined to *convert* between *serial* and *parallel* data streams.



For example, a 4-bit serial-to-parallel shift register as shown takes 4 clock ticks to process a 4-bit word from the serial data input d , at which point the 4-bit word is ready to be accessed by another sub-circuit.

9.2.5 | HISTORY OF FLIP-FLOPS

- The term “**flip-flop**” is used for any electronic circuit with two stable states that can be used to store 1 bit of information.
- The first electronic flip-flop invented 1918 by British physicists [William Eccles and Frank Wilfred Jordan](#); known as *Eccles-Jordan trigger circuit* (British patent docs) and implemented using two vacuum tubes.
- Modern flip-flop types (D, T, JK) were first discussed in 1954 UCLA course on computer design taught by Prof. Montgomery Phister, and then appeared in his 1958 book *Logical Design of Digital Computers* (John Wiley & Sons, New York).

Memory components based on flip-flops provide the memory requirements for registers inside the computational core of a system, such as the arithmetic-logic unit within a computer CPU or an embedded micro-controller.

Modern computers contain additional memory components. We have all heard of RAM and ROM in computers: RAM is *Random Access Memory* and ROM is *Read-only Memory*. These components are *external* to the computational core of the system, and are beyond the scope of this brief introduction.

9.3 | FEEDBACK AND STATE SIGNALS

The four elements of sequential systems beyond those within combinational systems are:

- time (with a digital clock);
- memory;
- feedback;
- state signals.

We have looked at time and memory; it remains to examine feedback and state signals.

9.3.1 | FEEDBACK OF STATE SIGNALS

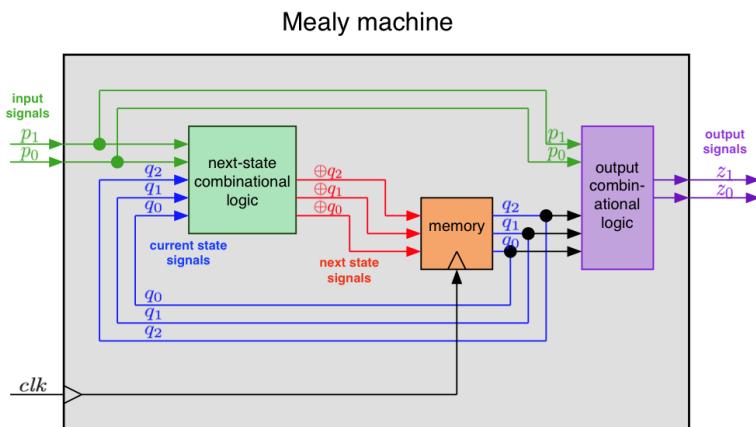
Sequential digital systems are distinguished from combinational digital systems in allowing feedback. *Feedback* of signals is:

- the means by which memory elements are created: inside our D flip-flops are three inter-connected SR NAND latches, and each SR NAND latch is an asynchronous feedback circuit; and
- the means by which the internal state signals are updated: the *next state* $\oplus q$ is equivalent to the excitation signal d , which is in turn a function of the *current state* q and the *current input/s* p .

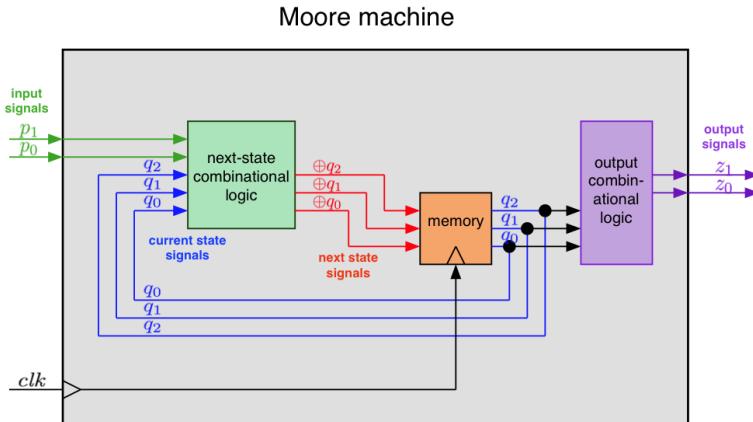
The state signal value in $\oplus q$ is “remembered” for one clock tick inside the flip-flop, and then passed on to the current state q , to be available for use in the next clock cycle.

Within the generic structure of synchronous sequential digital systems, we can separate the combinational sub-system into two smaller parts:

1. the next-state combinational logic sub-circuit (shown in green) is in feedback with the memory component; while
2. the output combinational logic sub-circuit (shown in purple) has the job of computing the output signals in terms of current state and current input.



The larger, more general class of sequential digital systems are known as *Mealy machines*. These allow the output to depend on the current input as well as the current state.



We will stay within the smaller class of *Moore machines* which require that the output depends only on the current state signals, but does not depend on the current input signals.

9.3.2 | STATE SIGNALS: WHAT AND HOW MUCH TO REMEMBER?

What we need to address now is the role and purpose of state signals. *What is it* that needs to be remembered or *kept track of*? Well, the answer is: **whatever configurations are required to be remembered in order to satisfy the specification**. That varies from system to system.

For example, if there are **4** different configurations to be tracked, then **two** state signals **$q_1 q_0$** will be required, as they take 4 combinations of values: 00, 01, 10 and 11. Under an assignment of state value codes to the different configurations to be tracked, we can distinguish between 4 different configurations with two state signals.

Or if there are **more than 4 and at most 8** different configurations to be tracked, then **three** state signals **$q_2 q_1 q_0$** will be required, as they take up to 8 combinations of values: 000, 001, 010, 011, 100, 101, 110 and 111.

There will be one D flip-flop for each state signal q_i , so two D flip-flops when there are four configurations to be tracked, and three D flip-flops there are more than 4 and at most 8 different configurations. We will not see examples with more states than that, but the general

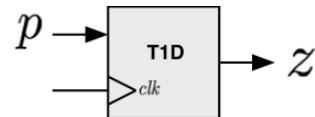
pattern is clear: the memory sub-system will consist of **n-many D flip-flops** if there are at most 2^n -many, but more than 2^{n-1} -many, different configurations to be tracked.

So far, we have guidance on *how many* state signals, but not on their *content or meaning*. For that, we need to work through an example, so we go back to the specification for the TID system, the triple-1 detector.

9.3.3 | SPECIFICATION ANALYSIS AND DESIGN SYNTHESIS FOR SYSTEM T1D

For our system TID, recall the past-focused version of the functional specification:

The output z has value 1 at time n if and only if input p was 1 at time $(n-1)$ and p was also 1 at time $(n-2)$ and p was also 1 at time $(n-3)$.



$$\square(z \equiv (\ominus p \& \ominus\ominus p \& \ominus\ominus\ominus p))$$

Direct from the meaning of the TID specification, we can see that the different *configurations* that need to be tracked include these four:

- $\langle 0 \rangle$ the previous value of input p was 0;
- $\langle 1 \rangle$ the previous value of input p was 1;
- $\langle 2 \rangle$ the previous two values of input p were 11;
- $\langle 3 \rangle$ the previous three values of input p were 111.

But perhaps we need all *eight* combinations of 3-bit values of:

$$(\ominus p \ \ominus\ominus p \ \ominus\ominus\ominus p)$$

covering the last three values of input p ? In order to see exactly which configurations need to be tracked, we have to work through a more detailed analysis of how a system would go about *meeting* the specification by *transitioning* from one configuration to another, and how those configurations correspond to combinations of three previous p input values, $(\ominus p \ \ominus\ominus p \ \ominus\ominus\ominus p)$.

- $\langle 0 \rangle$ First, there is the “*Got nothing*” configuration, which the system is in if $\ominus p$ has value 0, which means $\ominus p$ is false, and this is the same as $\ominus\sim p$ being true. The system is also in this configuration if **time $n = 0$** because we have just started, which means $\sim\ominus T$ is true. Note that among the 3-bit values of $(\ominus p \ \ominus\ominus p \ \ominus\ominus\ominus p)$, this configuration covers four cases: (000), (001), (010) and (011), because the values of $\ominus\ominus p$ and $\ominus\ominus\ominus p$ are not constrained.
- If the system is in configuration $\langle 0 \rangle$, and the current input p is 0,

then the *next* configuration will be $\langle 0 \rangle$ again: still *Got nothing*. If the system is in configuration $\langle 0 \rangle$ and the current input p is 1, then the *next* configuration will be $\langle 1 \rangle$, the *Got one 1* state, as described below.

- $\langle 1 \rangle$ The second configuration is “*Got one 1*”, which means $\ominus p$ is true, and either $\ominus\ominus\sim p$ is true or **time $n = 1$** . Among the 3-bit values of the previous p signals, ($\ominus p$ $\ominus\ominus p$ $\ominus\ominus\ominus p$), this configuration covers two cases: (100) and (101), because the value of $\ominus\ominus\ominus p$ is not constrained, so can be either 0 or 1.
If the system is in configuration $\langle 1 \rangle$, and the current input p is 0, then the *next* configuration will be $\langle 0 \rangle$, the *Got nothing* state.
If the system is in configuration $\langle 1 \rangle$ and the current input p is 1, then the *next* configuration will be $\langle 2 \rangle$, the *Got two 1s* state, as described below.
- $\langle 2 \rangle$ The third configuration is “*Got two 1s*”, which means $\ominus p$ and $\ominus\ominus p$ are both true, and either $\ominus\ominus\ominus\sim p$ is true or **time $n = 2$** . This covers the single case of the previous p signals ($\ominus p$ $\ominus\ominus p$ $\ominus\ominus\ominus p$) having the 3-bit value (110).
If the system is in configuration $\langle 2 \rangle$, and the current input p is 0, then the *next* configuration will be $\langle 0 \rangle$, the *Got nothing* state.
If the system is in configuration $\langle 2 \rangle$ and the current input p is 1, then the *next* configuration will be $\langle 3 \rangle$, the *Got three 1s* or *Success* state, as described below.
None of these first three configurations are yet *successful* in the sense of resulting in a z output of 1, but they are three stages of progress *towards* such success, with the progression of configurations $\langle 0 \rangle \mapsto \langle 1 \rangle \mapsto \langle 2 \rangle \mapsto \langle 3 \rangle$ as the input p receives three 1s in succession, after a 0 or time $n = 0$.
- $\langle 3 \rangle$ The final configuration we need to track is the “*Success*” configuration in which $\ominus p$, $\ominus\ominus p$, and $\ominus\ominus\ominus p$ are all true. This covers the single case of (111) for the 3-bit value of ($\ominus p$ $\ominus\ominus p$ $\ominus\ominus\ominus p$).
If the system is in configuration $\langle 3 \rangle$, and the current input p is 0, then the *next* configuration will be $\langle 0 \rangle$, the *Got nothing* state.
If the system is in configuration $\langle 3 \rangle$ and the current input p is 1, then the *next* configuration will be $\langle 3 \rangle$ again, the *Got three 1s* state.

Output To meet the specification for the triple-1 detector, the output signal z must have value 1 if and only if the system is in the *Got three 1s Success* configuration $\langle 3 \rangle$.

In this way, we reduce the *eight* possible combinations of 3-bit values of previous p signals ($\ominus p$ $\ominus\ominus p$ $\ominus\ominus\ominus p$) to *four* different configurations relevant for a system to satisfy the triple-1 detector specification.

Having four distinct configurations, we can use a 2-bit state signal $q_1 q_0$, and assign the four state configurations $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$ and $\langle 3 \rangle$, to

the four 2-bit values: (00), (01), (10) and (11), respectively.

On the basis of this assignment of state values, the *success configuration* is assigned the state combination of q_1 is 1 and q_0 is 1. So to compute the output signal z we can just use an AND gate and take z to be the conjunction ($q_1 \& q_0$). This makes z a *Moore machine* output because z depends only on state signals q_1 and q_0 , but not directly on the current value of the input p .

The system design arising from our analysis of the TID specification can be summarised in the following table:

$q_1 q_0$	configuration to be remembered, and transitions
00	$\langle 0 \rangle$: Got nothing: $\ominus p$ or $n = 0$ from $\langle 0 \rangle$ goto $\langle 0 \rangle$ if p is 0; from $\langle 0 \rangle$ goto $\langle 1 \rangle$ if p is 1.
01	$\langle 1 \rangle$: Got one 1: $\ominus p$ and $\ominus \ominus p$ or $n = 1$ from $\langle 1 \rangle$ goto $\langle 0 \rangle$ if p is 0; from $\langle 1 \rangle$ goto $\langle 2 \rangle$ if p is 1.
10	$\langle 2 \rangle$: Got 11: $(\ominus p \& \ominus \ominus p)$ and $\ominus \ominus \ominus p$ or $n = 2$ from $\langle 2 \rangle$ goto $\langle 0 \rangle$ if p is 0; from $\langle 2 \rangle$ goto $\langle 3 \rangle$ if p is 1.
11	$\langle 3 \rangle$: Got 111: $(\ominus p \& \ominus \ominus p \& \ominus \ominus \ominus p)$ from $\langle 3 \rangle$ goto $\langle 0 \rangle$ if p is 0; from $\langle 3 \rangle$ goto $\langle 3 \rangle$ if p is 1.
output z :	$z \equiv (q_1 \& q_0)$ Moore output

Notice the *arbitrariness* of the state assignment process. We have chosen *one* way of associating 2-bit q values ($q_1 q_0$) to state configurations, here labelled $\langle 0 \rangle$, $\langle 1 \rangle$, $\langle 2 \rangle$ and $\langle 3 \rangle$. Admittedly, the assignment here is the most *obvious* one, since it gives a base 2 encoding of the numbers 0, 1, 2 and 3. Had we done things differently, for example, and assigned configuration $\langle 3 \rangle$ the 2-bit value (10), we would end up with a different output logic, namely $z \equiv (q_1 \& \sim q_0)$.

9.3.4 | STATE-TRANSITION/OUTPUT TABLES

A *state-transition/output table* sets out the truth-functional dependence of the next-state signals $\oplus q_1$ and $\oplus q_0$ and the output signals z on the current input p and the current state q_0 and q_1 .

For the system TID, the state-transition/output table is as follows:

p	q_1	q_0	$\oplus q_1$	$\oplus q_0$	z
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	1

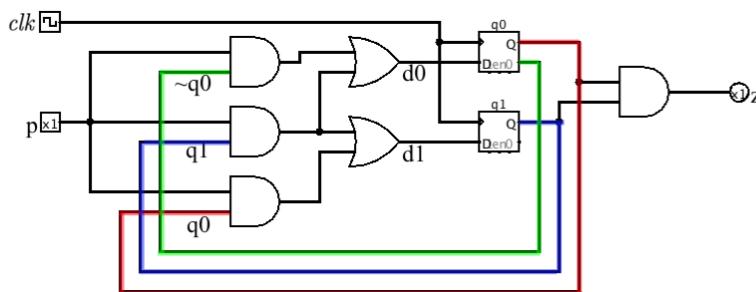
As a truth-table, it has 8 rows because we are looking at the truth-functional dependence on three signals, the current input p and the 2-bits of the current state, q_1 and q_0 . The table expresses the dependence of the 2-bits of the next-state signal, $\oplus q_1$ and $\oplus q_0$, and the dependence of the output signal z , on the three signals p , q_1 and q_0 that are inputs to the next-state combinational sub-circuit.

- The state transitions “from $\langle i \rangle$ goto $\langle 0 \rangle$ if p is 0”, for $i = 0, 1, 2, 3$, are encoded in the first four rows of the table, in which the next state ($\oplus q_1 \oplus q_0$) is (00) when ($p \ q_1 \ q_0$) is (000), (001), (010), or (011).
- The state transition “from $\langle 0 \rangle$ goto $\langle 1 \rangle$ if p is 1” is encoded in row (100) of the table, in which ($p \ q_1 \ q_0$) is (100) and the next state ($\oplus q_1 \oplus q_0$) is (01).
- The state transition “from $\langle 1 \rangle$ goto $\langle 2 \rangle$ if p is 1” is encoded in row (101) of the table, in which ($p \ q_1 \ q_0$) is (101) and the next state ($\oplus q_1 \oplus q_0$) is (10).
- The state transition “from $\langle 2 \rangle$ goto $\langle 3 \rangle$ if p is 1” is encoded in row (110) of the table, in which ($p \ q_1 \ q_0$) is (110) and the next state ($\oplus q_1 \oplus q_0$) is (11).
- The state transition “from $\langle 3 \rangle$ goto $\langle 3 \rangle$ if p is 1” is encoded in row (111) of the table, in which ($p \ q_1 \ q_0$) is (111) and the next state ($\oplus q_1 \oplus q_0$) is (11).
- The output z is 1 if and only if the current state ($q_1 q_0$) is (11); hence:

$$z \equiv (q_1 \ \& \ q_0)$$

In particular, z is *independent* of the current input p , and is thus a *Moore-type* output.

In §9.1, we first presented this circuit, and these next-state and output logic formulas, for the system TID.



The next-state and output equivalences for TID in the circuit:

$$\begin{aligned} \oplus q_1 &\equiv d_1 \equiv ((p \ \& \ q_1) \vee (p \ \& \ \sim q_0)) \\ \oplus q_0 &\equiv d_0 \equiv ((p \ \& \ q_1) \vee (p \ \& \ q_0)) \vee (\sim p \ \& \ q_1) \\ z &\equiv (q_1 \ \& \ q_0) . \end{aligned}$$

By analysing the state-transition/output table for the TID system, we can understand how such a circuit is *designed*.

9.3.5 | DESIGNING NEXT-STATE CIRCUIT FROM STATE-TRANSITION TABLE

We give a general recipe for extracting the next-state logic circuit from a state-transition table for a system. We continue with the triple-1 detector system TID, and will follow on with a second example.

To derive a logic formula for $\oplus q_1$ in terms of p, q_1 and q_0 , we go to the state-transition table and focus on the 1s in the column for $\oplus q_1$.

p	q_1	q_0	$\oplus q_1$	$\oplus q_0$	z
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	1

There are three 1s, and they pick out three combinations of p, q_1 and q_0 , corresponding to the last three rows of the table, in which $(p \ q_1 \ q_0)$ is (101), (110) and (111). We can express these as a DNF (Disjunctive Normal Form) formula. Using K-maps or a sequence of propositional logic equivalences, we can cancel complementary literals and end up with a simplified DNF formula for $\oplus q_1$ as a size-2 disjunction of size-2 conjunctions, as follows:

$$\begin{aligned}\oplus q_1 &\equiv ((p \ \& \ \sim q_1 \ \& \ q_0) \vee (p \ \& \ q_1 \ \& \ \sim q_0) \vee (p \ \& \ q_1 \ \& \ q_0)) \\ \oplus q_1 &\equiv ((p \ \& \ q_1) \vee (p \ \& \ q_0))\end{aligned}$$

Likewise, to derive a logic formula for $\oplus q_0$ in terms of p, q_1 and q_0 , we go to the state-transition table and focus on the 1s in the column for $\oplus q_0$.

p	q_1	q_0	$\oplus q_1$	$\oplus q_0$	z
0	0	0	0	0	0
0	0	1	0	0	0
0	1	0	0	0	0
0	1	1	0	0	1
1	0	0	0	1	0
1	0	1	1	0	0
1	1	0	1	1	0
1	1	1	1	1	1

There are three 1s, and they pick out three combinations of p, q_1 and q_0 , corresponding to the fourth-last and the last two rows of the table, in which $(p \ q_1 \ q_0)$ is (100), (110) and (111). We can express these as a DNF (Disjunctive Normal Form) formula. Using K-maps or a sequence of propositional logic equivalences, we can cancel complementary literals and end up with a simplified DNF formula for $\oplus q_0$ as

a size-2 disjunction of size-2 conjunctions, as follows:

$$\oplus q_0 \equiv ((p \& \sim q_1 \& \sim q_0) \vee (p \& q_1 \& \sim q_0) \vee (p \& q_1 \& q_0))$$

$$\oplus q_0 \equiv ((p \& q_1) \vee (p \& \sim q_0))$$

In this way, we end up with the next-state and output logic for the TID system, which is what is built in to the circuit above:

$$\begin{aligned}\oplus q_1 &\equiv d_1 \equiv ((p \& q_1) \vee (p \& q_0)) \\ \oplus q_0 &\equiv d_0 \equiv ((p \& q_1) \vee (p \& \sim q_0)) \\ z &\equiv (q_1 \& q_0).\end{aligned}$$

9.3.6 | SPECIFICATION ANALYSIS AND DESIGN SYNTHESIS FOR MOD-4 COUNTER SYSTEM CNT4

To get some more practice with state signals and feedback, we will quickly work through another example: a *mod-4 counter*. Essentially, the job of this system is to count through a cycle of four unless its reset input is activated.

Consider a clocked sequential system CNT4 with:

- a 1-bit input signal p , for *reset*;
- a 2-bit internal state signal $q = (q_1 q_0)$; and
- a 2-bit output signal $z = (z_1 z_0)$, such that z is *the same value* as q .

The natural language specification is as follows:

the 2-bit output signal z must cycle through the values $(00) \mapsto (01) \mapsto (10) \mapsto (11) \mapsto (00) \mapsto \dots$ whenever the reset input p remains *inactive* with value 0,

while output z must *next* become (00) after the reset input p is activated with value 1.

For this specification, the analysis of the configurations to be remembered is rather easier. The state signal q must range through the four values (00) , (01) , (10) and (11) , and the output z must be the same as the state q . So we can skip the stage of using names for the state configurations, and go straight to the 2-bit values themselves.

(00) The system will be in the (00) state if one of three situations hold:

- the previous value of p was 1: the *reset* input was activated and the counter state was reset to become (00) now;
- time $n = 0$ because the run has just started; or
- the previous value of p was 0 and previous state $(\oplus q_1 \oplus q_0)$ was (11) .

If the system is in configuration (00) , and current input p is 0,

then the *next* configuration will be (01) , advancing the counter;

If the system is in configuration (00) and current input p is 1,

then the *next* configuration will be (00) again.

- (01) The system will be in the (01) state if and only if the previous value of p was 0 and the previous state ($\ominus q_1 \ominus q_0$) was (00).
 If the system is in configuration (01), and current input p is 0, then the *next* configuration will be (10), advancing the counter;
 If the system is in configuration (01) and current input p is 1, then the *next* configuration will be (00).
- (10) The system will be in the (10) state if and only if the previous value of p was 0 and the previous state ($\ominus q_1 \ominus q_0$) was (01).
 If the system is in configuration (10), and current input p is 0, then the *next* configuration will be (11), advancing the counter;
 If the system is in configuration (10) and current input p is 1, then the *next* configuration will be (00).
- (11) The system will be in the (11) state if and only if the previous value of p was 0 and the previous state ($\ominus q_1 \ominus q_0$) was (01).
 If the system is in configuration (11), and current input p is 0, then the *next* configuration will be (00), advancing the counter;
 If the system is in configuration (11) and current input p is 1, then the *next* configuration will be (00).

Output To meet the specification for the mod-4 counter, we must take $z_1 \equiv q_1$ and $z_0 \equiv q_0$.

The system design arising from our analysis of the CNT4 specification can be summarised in the following table:

$(q_1 q_0)$	configuration to be remembered
00	State (00): $\ominus p$ or $n = 0$ or $(\ominus \sim p \ \& \ \ominus q_1 \ \& \ \ominus q_0)$ from (00) goto (01) if p is 0; from (00) goto (00) if p is 1.
01	State (01): $(\ominus \sim p \ \& \ \ominus \sim q_1 \ \& \ \ominus \sim q_0)$ from (01) goto (10) if p is 0; from (01) goto (00) if p is 1.
10	State (10): $(\ominus \sim p \ \& \ \ominus \sim q_1 \ \& \ \ominus q_0)$ from (10) goto (11) if p is 0; from (10) goto (00) if p is 1.
11	State (11): $(\ominus \sim p \ \& \ \ominus q_1 \ \& \ \ominus \sim q_0)$ from (11) goto (00) if p is 0; from (11) goto (00) if p is 1.
output z :	$(z_1 \equiv q_1) \ \& \ (z_0 \equiv q_0)$ Moore output

Completing the state-transition/output table for the mod-4 counter system CNT4 is also pretty clear.

Start with a truth table with 8 rows for valuations of p , q_1 and q_0 , and with four blank columns grouped in 2-bit pairs, for the next-states $\oplus q_1$ and $\oplus q_0$, and for the outputs z_1 and z_0 . The easiest bit is to fill in the columns for Moore-machine outputs z_1 and z_0 ; here, we simply

duplicate the columns for current states q_1 and q_0 , so that $z_1 \equiv q_1$ and $z_0 \equiv q_0$.

p	q1	q0	$\oplus q1$	$\oplus q0$	z1	z0
0	0	0	0	1	0	0
0	0	1	1	0	0	1
0	1	0	1	1	1	0
0	1	1	0	0	1	1
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	0	1	0
1	1	1	0	0	1	1

For the state transitions from $(q_1 q_0)$ to $(\oplus q_1 \oplus q_0)$, the last four rows of the table encode the transitions making $(\oplus q_1 \oplus q_0)$ take value (00) from any state $(q_1 q_0)$ when the current state p is 1. The first four rows of the table encode the state transitions that increment the state $(00) \mapsto (01) \mapsto (10) \mapsto (11) \mapsto (00) \mapsto \dots$ while the current input p remains inactive with value 0.

Now an exercise for you: reading from the state-transition/output table, extract the next-state logic circuit for the mod-4 counter system CNT4. You might want to highlight the 1s in the column for $\oplus q_1$ and in the column for $\oplus q_0$, to assist you in the task.

Reading from the state-transition/output table, which of the following are correct next-state formulas for the mod-4 counter system CNT4?

- (a) $\oplus q_1 \equiv (p \wedge \neg q_1 \wedge q_0) \vee (p \wedge q_1 \wedge \neg q_0)$
 $\oplus q_0 \equiv (p \wedge \neg q_0)$

(b) $\oplus q_1 \equiv (\neg p \wedge \neg q_1 \wedge q_0) \vee (\neg p \wedge q_1 \wedge \neg q_0)$
 $\oplus q_0 \equiv (\neg p \wedge \neg q_0)$

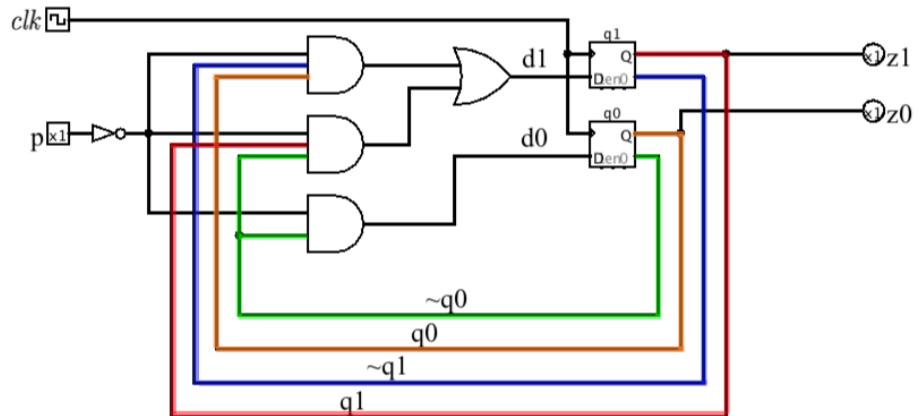
(c) $\oplus q_1 \equiv (\neg p \wedge \neg q_1 \wedge \neg q_0) \vee (\neg p \wedge q_1 \wedge \neg q_0)$
 $\oplus q_0 \equiv (\neg p \wedge \neg q_0)$

(d) $\oplus q_1 \equiv (\neg p \wedge \neg q_1 \wedge q_0) \vee (\neg p \wedge \neg q_1 \wedge \neg q_0)$
 $\oplus q_0 \equiv (\neg p \wedge \neg q_1)$

The correct answer is (b). The two 1s for \oplus_1 are for the two rows in which $(p \oplus_1 q)$ is (001) and (010). This gives the DNF $(\sim p \wedge \sim q \wedge q_0) \vee (\sim p \wedge q_1 \wedge \sim q_0)$. The two 0s for \oplus_1 are for the two rows in which $(p \oplus_1 q)$ is (000) and (010). This gives the DNF $(\sim p \wedge \sim q_1 \wedge \sim q_0) \vee (\sim p \wedge q_1 \wedge \sim q_0)$. The two 1s for \oplus_0 are for the two rows in which $(p \oplus_0 q)$ is (000) and (010). This gives the DNF $(\sim p \wedge q_0) \vee (\sim p \wedge q_1 \wedge \sim q_0)$. The two 0s for \oplus_0 are for the two rows in which $(p \oplus_0 q)$ is (000) and (010). This gives the DNF $(\sim p \wedge q_0) \wedge (\sim p \wedge q_1 \wedge \sim q_0)$, which then simplifies to $(\sim p \wedge q_0)$.

Putting these next-state formulas together with the extremely simple output formulas for z_1 and z_0 , and we get a recipe to construct a circuit for the mod-4 counter system CNT4:

$$\begin{aligned}\oplus q_1 &\equiv (\neg p \wedge \neg q_1 \wedge q_0) \vee (\neg p \wedge q_1 \wedge \neg q_0) \\ \oplus q_0 &\equiv (\neg p \wedge \neg q_0) \\ z_1 &\equiv q_1 \\ z_0 &\equiv q_0\end{aligned}$$



To explore the behaviour of sequential systems, we recommend an educational software tool for designing and simulating digital logic circuits, called *Logisim*, and developed by Professor Carl Burch from Hendrix College, Arkansas. It is free, downloadable, open-source (GPL), cross-platform software based on Java 5: <http://ozark.hendrix.edu/~burch/logisim/> The *Logisim* circuit files CNTR4.circ for the mod-4 counter, and T1D.circ for the triple-1 detector system, will be available for students to experiment with using *Logisim*.

9.4 | LINEAR TEMPORAL LOGIC

9.4.1 | LOGICS FOR SEQUENTIAL DIGITAL SYSTEMS

We have already been using a logic extending propositional logic with the *Next* and *Previous* operators \oplus and \ominus , and using such logic formulas with correct but not yet formal semantics. Before we get to the formalities, let's stop and review the work we want to do using logic. For sequential digital systems, we want to work in a logic that:

- enables us to formalise *system specifications* of input-output behaviour as formulas of the logic's language;
- enables us to formalise *system descriptions* of next-state and output behaviour as formulas of the logic's language;

- extends the framework of propositional logic as both a specification language and a system description language for combinational digital systems, within which atomic propositions denote digital signals; and
- has a semantics that enables us to *formally verify* that a specification holds of a system in response to all possible input signals.

We use a fragment of a logic called *Linear Temporal Logic* or LTL, restricted to one-place operators. Basic temporal logic uses *future-focused* operators, starting with *Next*, \oplus , and adding *Always*, \square , and *Eventually*, \diamond . As we have seen already, the addition of the past-focused *Previous*, \ominus , is rather useful in analysing the D flip-flop and the content that needs to be remembered by state signals.

So for our purposes, the optimal choice of logic is LTL_{\ominus}^1 : *Linear Temporal Logic* restricted to 1-place future operators, augmented with the past operator *Previous*. Let's start by defining the language precisely.

9.4.2 | THE LANGUAGE OF LINEAR TEMPORAL LOGIC

The language of LTL_{\ominus}^1 , *Linear Temporal Logic with Previous*, is generated from a set Σ of atomic proposition symbols together with a constant proposition \top , whose meaning is that it is *always true*.

- Atoms in Σ and the constant \top are formulas.
- If A and B are formulas, then $\sim A$, $(A \& B)$, $(A \vee B)$, $(A \supset B)$ and $(A \equiv B)$ are formulas.
- If A is a formula, then $\oplus A$, $\ominus A$, $\diamond A$ and $\square A$ are formulas.
- Nothing else is a formula.

Full **LTL** has 2-place temporal operators *until* and *release*, making formulas $(A \mathcal{U} B)$ and $(A \mathcal{R} B)$, but in this short introduction, we don't have the time and space to discuss these.

Note that other presentations of temporal logic may use different symbols for the same operators; in particular, for the *Next* operator, a plain circle symbol \bigcirc or a bold \mathbf{X} are sometimes used rather than the \oplus symbol used here. The symbol \ominus is usual for *Previous*. We have chosen \oplus and \ominus because they are easy to read, are visually a complementary pair, and are available in the limited set of math symbols that can be coded within web pages, which we need for online quizzes.

LTL_{\ominus}^1 formula	meaning in English
$\oplus A$	<i>next-time A</i> or <i>next A</i>
$\ominus A$	<i>last-time A</i> or <i>just previously A</i>
$\diamond A$	<i>eventually A</i> or <i>at some time A</i>
$\square A$	<i>always A</i> or <i>at all times A</i>

The \square and \diamond operators function like quantifiers, but applied directly to propositions, with the quantification over time. We could have also added *past-focused* versions of the \square and \diamond operators, expressing the quantifications “*at all times in the past*” and “*at some time in the past*”. We don’t include them in this introduction as they are of less immediate use for sequential digital systems.

Now, let’s have a look at some sample formulas of our language.

$$\square(\oplus\oplus\oplus z \equiv (p \& \oplus p \& \oplus\oplus p))$$

Our first example is the future-focused version of the specification for the triple-1 detector system. It says: “at all times in the future, $\oplus\oplus\oplus z$ is true if and only if p and $\oplus p$ and $\oplus\oplus p$ are all true”. Future-focused specifications are the most common in temporal logic. We will see a \square operator out the front of all formulas expressing either specifications or system descriptions for digital systems that are required to hold at all discrete time points n .

$$\square(z \equiv (\ominus p \& \ominus\ominus p \& \ominus\ominus\ominus p))$$

Our second example is the past-focused version of the specification for the triple-1 detector. Again, we need the \square out the front. The formula says: “for all times, z is 1 if and only if $\ominus p$, $\ominus\ominus p$ and $\ominus\ominus\ominus p$ are all true”. Later in this lesson, we will make precise the relationship between these two versions of the specification for the TID system.

$$\diamond\square(p \& \oplus\sim z)$$

The formula $\diamond\square(p \& \oplus\sim z)$ says that at some time in the future, $(p \& \oplus\sim z)$ is true, and then it remains true for all times after that. More succinctly: “eventually permanently $((p \& \oplus\sim z))$ ”.

$$\square\diamond(z \& \ominus\sim p)$$

Reversing the \diamond and \square operators, the formula $\square\diamond(z \& \ominus\sim p)$ says that the combination $(z \& \ominus\sim p)$ occurs *infinitely often*. Once we formalise the semantics, we will see that this formula says: “for every time point n , there exists a later time $m \geq n$ such that $(z \& \ominus\sim p)$ is true at time m ”. This guarantees that the total number of times at which $(z \& \ominus\sim p)$ is true, is infinite.

$$\square \left(\begin{array}{l} (\oplus(\sim z_1 \& \sim z_0)) \equiv ((\sim p \& z_1 \& z_0) \vee p)) \\ \& (\oplus(z_1 \& z_0)) \equiv (\sim p \& \sim z_1 \& \sim z_0)) \\ \& (\oplus(z_1 \& \sim z_0)) \equiv (\sim p \& z_1 \& \sim z_0)) \\ \& (\oplus(\sim z_1 \& z_0)) \equiv (\sim p \& \sim z_1 \& z_0)) \end{array} \right)$$

Our last example is the future-focused version of the specification for the mod-4 counter: “at all times: first, the 2-bit output z is next (00) if and only if either p is now 0 and z is (11), or else the reset input p is now 1; second, z is next (01) if and only if p is now 0 and z is (00); third, z is next (10) if and only if p is now 0 and z is (01); and fourth, z is next (11) if and only if p is now 0 and z is (10). In the case of the mod-4 counter, the specification is quite close to the description: just replace output z with state q , and add the output equivalences saying that z is the same as q in both bits.

Now, an exercise for you, identifying well-formed formulas of the language of LTL_\ominus^1 . Note that we are being relaxed about parentheses in conjunctions and disjunctions of size 3 or greater, so $(A \& B \& C)$ and $(A \vee B \vee C)$ are considered well-formed. We are interested here in the appropriate use of the *temporal* operators.

Let p, q, r, z and w be atoms in Σ .

Which of the following are well-formed formulas of the language of LTL_\ominus^1 ?

- (a) $\square(\sim p \supset \oplus\ominus\oplus(z \vee \sim q))$
- (b) $((p \& \oplus q) \vee \sim r) \supset \lozenge\lozenge(\sim z \ominus w)$
- (c) $\square\square(z \equiv (\ominus p \& \ominus\ominus\sim p \& \ominus\ominus\ominus p))$
- (d) $\lozenge\square(z \& w) \equiv (\sim(p \lozenge r) \vee \oplus\lozenge\square(q \& \sim r))$
- (e) $\square\lozenge(p \& \sim r) \supset \square\lozenge(\sim z \& w)$

$$\lozenge\square(z \& w) \equiv (\sim(\lozenge d) \vee \square\lozenge(\sim d \& \sim r))$$

correctly as a 2-place operator:

Formula (d) is also not well-formed, because \lozenge is being used incorrectly as a 2-place operator:

$$(((p \& \oplus q) \vee \sim r) \supset \lozenge\lozenge(z \sim w))$$

error:

well-formed, because \ominus is being used incorrectly as a 2-place operator:

Formulas (a) (c) and (e) are all well-formed. Formula (b) is not well-formed, because \ominus is being used incorrectly as a 2-place operator:

After we set out the formal semantics, we will be interested in seeing when various strings of temporal operators end up equivalent to shorter strings. As a first example, in formula (c) we expect the $\square\square$ combination should mean the same as one \square .

9.4.3 | SEMANTICS OF LINEAR TEMPORAL LOGIC

In propositional logic, a valuation maps each atomic proposition symbol to a bit value, 0 or 1. Here, we need a valuation v to map each atom to an infinite length *digital signal*.

Definition: A *model* or *valuation* for the language of LTL_{\ominus}^1 is a mapping v which assigns to each atomic proposition symbol p in Σ an infinite length *digital signal* $v(p)$: for each discrete time step n in $\mathbb{N} = \{0, 1, 2, 3, \dots\}$, the value $v(p)_n$ is either 0 or 1.

So that for every time step n , a valuation v assigns each *atom* a truth value. We can then propagate the truth value of atoms through to obtain a truth value for every complex formula at every time step n in the natural numbers. We use the standard symbol \mathbb{N} (blackboard-bold N) for the natural numbers.

Notice that the *number* of different possible valuations is infinite. The number of different finite-length digital signals is *countably infinite*, which means having the same cardinality as the natural numbers \mathbb{N} . However, the semantics of temporal logic require *infinite length* digital signals; the number of these is *uncountably infinite*, and is the same cardinality as the *real numbers* \mathbb{R} . (To see this, think about an infinite binary sequence as the base 2 encoding of a decimal real number.)

Notational aside: When the intended valuation v is clear, we will sometimes abbreviate $v(p)_n$ and simply write p_n to refer to the n -th bit value of a signal p – as we have done already with some concrete examples of initial segments of signals. Formally, this is a confusion between p as a syntactic object (an atomic proposition symbol) and as a semantic object (a signal interpreting an atom symbol). We will be careful with this *abuse of notation*, and make sure the intended valuation is clear.

Now, with each pair (v, n) consisting of a valuation v mapping atoms to digital signals, together with a time point n in the natural numbers, we have enough semantic information to determine the truth of all formulas A of our temporal language. The semantic definitions proceed according to the build-up of formulas in the language.

Definition: For formulas A in the language of LTL_{\ominus}^1 , for valuations v , and for discrete time points n in \mathbb{N} , we define the notion:

$$(v, n) \models A$$

read: valuation v at time point n makes formula A true, and its negation:

$$(v, n) \not\models A$$

read: valuation v at time point n does not make A true (so it makes A false), where the definition proceeds by induction on the structure of formulas.

- $(v, n) \models \top$ always holds;
- $(v, n) \models p$ if and only if $v(p)_n = 1$;
- $(v, n) \models \sim A$ if and only if $(v, n) \not\models A$;
- $(v, n) \models (A \& B)$ if and only if $(v, n) \models A$ and $(v, n) \models B$;
- $(v, n) \models (A \vee B)$ if and only if $(v, n) \models A$ or $(v, n) \models B$;
- $(v, n) \models (A \supset B)$ if and only if $(v, n) \not\models A$ or $(v, n) \models B$;
- $(v, n) \models (A \equiv B)$ if and only if
 - either $(v, n) \models A$ and $(v, n) \models B$,
 - or $(v, n) \not\models A$ and $(v, n) \not\models B$;
- $(v, n) \models \oplus A$ if and only if $(v, n+1) \models A$;
- $(v, n) \models \ominus A$ if and only if $n > 0$ and $(v, n-1) \models A$;
- $(v, n) \models \Box A$ if and only if for all time points $m \geq n$, we have $(v, m) \models A$;
- $(v, n) \models \Diamond A$ if and only if for some time point $m \geq n$, we have $(v, m) \models A$.

The first clause covers the constant symbol \top , and says \top is always true, for all valuations v and all time points n . For atomic propositions, p is true at (v, n) if and only if the signal $v(p)$ has value 1 at time point n . For a negation formula, $\sim A$ is true at (v, n) if and only if A is not true at (v, n) . In particular, for a negated atomic proposition, $\sim p$ is true at (v, n) if and only if the signal $v(p)$ have value 0 at time point n . For conjunction, $(A \& B)$ is true at (v, n) if and only if both A is true at (v, n) and B is true at (v, n) . Likewise for the other propositional connectives. The semantics proceed exactly as we would expect from classical propositional logic except that here we need a pair (v, n) of a valuation and a time point, rather than just a valuation as in propositional logic.

Now for the temporal operators: the formal meaning of \oplus and \ominus will match our less formal usage so far. A formula $\oplus A$ is true at (v, n) if and only if A is true at $(v, n+1)$. For *Next*, we look one time step to the future. A formula $\ominus A$ is true at (v, n) if and only if time n is greater than 0 and A is true at $(v, n-1)$. For *Previous*, we look one time step

to the past, if there is one, and get false if there isn't.

The *Always* operator is a temporal universal quantifier: $\Box A$ is true at (v, n) if and only if for all later time points m at or after n , formula A is true at (v, m) . The *Eventually* operator is a temporal existential quantifier: $\Diamond A$ is true at (v, n) if and only if for some later time point m at or after n , formula A is true at (v, m) .

The semantics for the constant T and the previous operator \ominus combine to yield that for all valuations v and time points n in \mathbb{N} :

$$\begin{aligned} (v, n) \models \sim \ominus T &\quad \text{if and only if time } n = 0 \\ (v, n) \models \ominus T &\quad \text{if and only if time } n > 0. \end{aligned}$$

This is another way of saying that time $n = 0$ is the only time that the \ominus operator is undefined, and this makes the formula $\ominus T$ false.

To make the semantics more concrete, here's an exercise for you straight away: you have to determine whether some formulas are true or false at a particular time under a particular valuation.

Let v be a valuation of atoms p , q and z with initial segments of signals as below:

n	0	1	2	3	4	5	6	7	8	9	10	...
p	1	0	0	1	0	0	1	1	0	1	0	...
q	0	1	0	0	1	0	0	1	1	1	0	...
z	1	0	0	1	0	0	1	1	0	0	1	...

For which of these formulas A is it the case that $(v, 6) \models A$?

- (a) $\ominus \sim p \& \ominus \ominus q \& \oplus \oplus \sim z$
- (b) $\oplus(z \equiv q) \& \oplus \oplus(z \equiv q)$
- (c) $(q \equiv \ominus p) \& \oplus(q \equiv \ominus p)$
- (d) $\Diamond((p \& q) \& \oplus(q \& z))$

TRUE: $p^6 = 1$, $b^6 = 1$ (as well as $p^7 = 1$, $b^7 = 1$, $z^6 = 1 = \sim z$)
 $\ominus \sim p \equiv \ominus 1 = 0$ and $\ominus \ominus q \equiv \ominus 0 = 1$
 $\oplus(b \oplus \ominus b) \equiv \oplus(1 \oplus \ominus 1) \equiv \oplus(1 \oplus 0) \equiv 1$ (p)

TRUE because $p^6 = 1 = \sim b$ and $z^6 = 0$
 $\oplus(p \oplus \ominus p) \equiv \oplus(1 \oplus \ominus 1) \equiv \oplus(1 \oplus 0) \equiv 1$ (c)

FALSE because $z^8 = 1 \neq 0 = \sim b$
 $\oplus(b \oplus \ominus b) \equiv \oplus(1 \oplus \ominus 1) \equiv \oplus(1 \oplus 0) \equiv 1$ (q)

TRUE because $p^5 = 0 = \sim b$ and $z^5 = 1$
 $\oplus(p \oplus \ominus p) \equiv \oplus(0 \oplus \ominus 0) \equiv \oplus(0 \oplus 0) \equiv 0$ (a)

So far, we have only defined truth under a valuation v in a *local* way, at each time point n in \mathbb{N} . We also need a *global* notion of truth under a valuation v , and that is taken as truth at time 0.

Definition: Given a valuation v for the set of atoms Σ and a formula A in the language of LTL_{\ominus}^1 , we write:

$$v \models A$$

meaning valuation v makes formula A true, if and only if:

$$(v, 0) \models A.$$

To express the global notion of truth at *all* times points along infinitely long digital signals, we use the *Always* operator \square .

$$v \models \square A \text{ if and only if } (v, n) \models A \text{ for all times } n \text{ in } \mathbb{N}.$$

A tautology of propositional logic is true under all valuations. The corresponding notion here is that of a formula being a *validity* of the logic, which is the case if and only if the formula is true in *every valuation* v .

Definition: We write:

$$LTL_{\ominus}^1 \models A$$

meaning formula A is a validity of the logic LTL_{\ominus}^1 , if and only if $v \models A$ for all possible valuations v .

For formal verification of sequential digital systems, we are particularly interested in establishing LTL_{\ominus}^1 -validity status for formulas of the form:

$$(\square D_{sys} \supset \square A_{spec})$$

where D_{sys} characterizes the next-state and output behaviour as a function of current input and current state, and A_{spec} expresses the desired input-output specification for that system. By requiring that such a conditional be a *validity*, we use the truth-transferring semantics of the conditional to ensure that for all possible valuations v , if v makes true $\square D_{sys}$ then v makes true $\square A_{spec}$. In this way, we can quantify over all of the uncountable infinity of possible input signals, and guarantee that every output signal resulting from the system as described will satisfy the specification.

In contrast with combinational systems, sequential systems in general require some effort to formally verify in logic that the specification formula is a logical consequence of the system description formula.

One can also consider stronger notions of consequence and equivalence that require $\Box(A \supset B)$ and $\Box(A \equiv B)$ to be validities, but we do not examine them here.

The notions of *logical consequence* and *logical equivalence* are as usual.

Definition: Formula B is a *logical consequence* of formula A in LTL_{\ominus}^1 if and only if:

$$LTL_{\ominus}^1 \models (A \supset B).$$

Formulas A and B are *logically equivalent* in LTL_{\ominus}^1 if and only if:

$$LTL_{\ominus}^1 \models (A \equiv B).$$

9.4.4 | VALIDITIES OF LTL WITH PREVIOUS

Our first batch of useful equivalence validities involve \oplus and \ominus . Each of these equivalences are always true, so we can replace A with *any* formula of the language, and put a \Box out the front, and the result is a validity of the logic. By *validity scheme*, we mean this sort of general recipe for generating validities.

For any formula A, the \Box of each of these is a validity of LTL_{\ominus}^1 :

$$\begin{array}{ll} \oplus\ominus A \equiv A & \ominus\oplus A \equiv (A \& \ominus T) \\ \sim\oplus A \equiv \oplus\sim A & \sim\ominus A \equiv (\ominus\sim A \vee \sim\ominus T) \\ \sim\oplus\sim A \equiv \oplus A & \sim\ominus\sim A \equiv (\ominus A \vee \sim\ominus T) \end{array}$$

The first equivalence says that the operator combination $\oplus\ominus$ cancels out, so it is always the case that $\oplus\ominus A$ is equivalent to A. The other combination, $\ominus\oplus$ is a little messier: $\ominus\oplus A$ is equivalent to the conjunction of A and the formula saying time $n > 0$. The \oplus operator commutes completely with negation, so $\sim\oplus A$ is equivalent to $\oplus\sim A$. Again, a little messier with *Previous*: $\sim\ominus A$ is equivalent to the disjunction of $\ominus\sim A$ with the formula saying time $n = 0$.

Summarizing the time-specificity formulas: for all valuations v and time points n in \mathbb{N} ,

- $(\sim\ominus T)$ is true at (v, n) if and only if time $n = 0$;
- $(\ominus T)$ is true at (v, n) if and only if time $n > 0$;
- $(\sim\ominus\ominus T)$ is true at (v, n) if and only if time $n = 1$;
- $(\sim\ominus\ominus\ominus T)$ is true at (v, n) if and only if time $n = 2$

The pattern generalises: the formula $(\sim\ominus\ominus^n T)$ is true at time n, for all natural numbers n.

Applying the definition of validity, we get the somewhat curious result for the time $n = 0$ formula:

$\sim\ominus T$ is a validity of LTL_{\ominus}^1 .

This is so because $(v, 0) \models \sim\ominus T$ for all valuations v . In fact this is not really so curious: it just reinforces that truth in a valuation is evaluated at time $n = 0$ because this means that the full length of the signals in the valuation are incorporated into the semantics. There are some subtleties to the semantics of temporal logic, and this is one of them. The alternative choice of taking truth under a valuation to mean true at all time points n also has some curious side-effects, and at any rate, we can simply express the *at all times* meaning using the \Box operator.

A few more useful equivalences involving \oplus and \ominus . Again, each of these equivalences are always true, so replace A and B with any formulas of the language, and put a \Box out the front, and the result is a validity of the logic. These equivalences tell us that both operators cleanly distribute over both conjunction and disjunction.

$$\begin{array}{ll} \sim\oplus A \equiv \oplus\sim A & \sim\ominus A \equiv (\ominus\sim A \vee \sim\ominus T) \\ \oplus\sim A \equiv \sim\oplus A & \ominus\sim A \equiv (\sim\ominus A \& \ominus T) \\ \oplus(A \& B) \equiv (\oplus A \& \oplus B) & \ominus(A \& B) \equiv (\ominus A \& \ominus B) \\ \oplus(A \vee B) \equiv (\oplus A \vee \oplus B) & \ominus(A \vee B) \equiv (\ominus A \vee \ominus B) \\ \oplus(A \supset B) \equiv (\oplus A \supset \oplus B) & \ominus(A \supset B) \equiv (\ominus A \supset \ominus B) \\ \oplus(A \equiv B) \equiv (\oplus A \equiv \oplus B) & \ominus(A \equiv B) \equiv (\ominus A \equiv \ominus B) \end{array}$$

Since \oplus also commutes with negation, we can see that \oplus also distributes over the conditional and bi-conditional connectives. For the \ominus operator, distribution over the conditional and bi-conditional connectives requires an extra side condition that time n is greater than 0, via the condition $\ominus T$.

We set out a few principles for generating validities of the temporal logic. First, if instances of propositional tautologies give validities because tautologies are always true.

If A is an instance of a propositional tautology in the language of LTL_{\ominus}^1 , then $\Box A$ is a validity of LTL_{\ominus}^1 .

For example, instantiating the material conditional equivalence, which is $(A \supset B) \equiv (\sim A \vee B)$, with A as $\Diamond\Box p$ and B as $\Box\Diamond p$, we get:

$$\Box((\Diamond\Box p \supset \Box\Diamond p) \equiv (\sim\Diamond\Box p \vee \Box\Diamond p))$$

is a validity of LTL_{\ominus}^1 .

Substitution of equivalents is a basic principle for generating validities, and it is along the same lines as substitution of equals for equals in elementary mathematics. If C is an LTL_{\ominus}^1 formula containing atom p , and A is another LTL_{\ominus}^1 formula, then $C[p := A]$ is the formula obtained by replacing all occurrences of p in C with A .

Substitutivity of Equivalents:

$$(\square(A \equiv B) \supset \square(C[p := A] \equiv C[p := B]))$$

is a validity of LTL_{\ominus}^1 .

For a simple example, suppose A is $\oplus q$, B is d and C is the formula $\ominus p$. Then $C[p := A]$ is $\ominus \oplus q$ and $C[p := B]$ is just $\ominus d$. So we have that:

$$\square(\oplus q \equiv d) \supset \square(\ominus \oplus q \equiv \ominus d)$$

is a validity of LTL_{\ominus}^1 . Recall that $\square(\oplus q \equiv d)$ is characteristic formula D_1 of the D flip-flop.

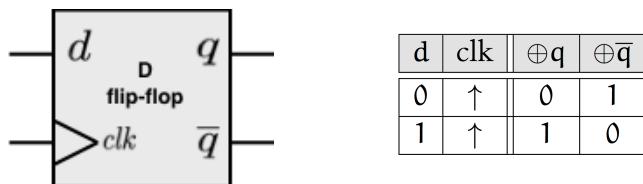
Heres a few more useful equivalences involving \square and \diamond as well as \oplus and \ominus . For any formulas A and B , the \square of each of these gives a validity of LTL_{\ominus}^1 :

$$\begin{array}{lll} \sim \square A \equiv \diamond \sim A & \sim \diamond A \equiv \square \sim A \\ \square(A \& B) \equiv (\square A \& \square B) & \diamond(A \vee B) \equiv (\diamond A \vee \diamond B) \\ \square A \equiv (A \& \oplus \square A) & \diamond A \equiv (A \vee \oplus \diamond A) \\ \oplus \square A \equiv \square \oplus A & \oplus \diamond A \equiv \diamond \oplus A \\ \square \square A \equiv \square A & \diamond \diamond A \equiv \diamond A \\ \square(A \supset B) \supset (\square A \supset \square B) & \diamond(A \& B) \supset (\diamond A \supset \diamond B) \end{array}$$

The first two say that \square and \diamond are de Morgan-like duals in the same way \forall and \exists are de Morgan-like duals: a *not always* formula $\sim \square A$ means the same as *sometimes not* $\diamond \sim A$. The \square operator distributes over conjunction, while \diamond distributes over disjunction; these ones are used often. The fifth pair of equivalences express that $\square \square$ is the same as \square , and $\diamond \diamond$ is the same as \diamond .

9.4.5 | THE TEMPORAL LOGIC OF THE D FLIP-FLOP

We make use of LTL with Previous to further analyse the characterising conditions for the D type positive edge-triggered flip-flop.



Three formulas of LTL_{\ominus}^1 characterise the behaviour of the D flip-flop memory component:

$$D_1 : \square(\oplus q \equiv d) \quad D_2 : \square(\bar{q} \equiv \sim q) \quad D_3 : \square(q \supset \ominus T)$$

As we explained using pre-formal semantics in §9.2.2, the first formula D_1 says the next-state signal $\oplus q$ always has the same value as the data input d . The second logic formula D_2 says \bar{q} is always equivalent to the negation of q . And the third formula D_3 expresses that at time $n = 0$, the state signal q has the default value of 0; D_3 's equivalent contrapositive, $(\sim \ominus T \supset \sim q)$ says that if time $n = 0$, when $\sim \ominus T$ is true, then $\sim q$ has value 1, so q has value 0.

In §9.2.2, we put forth two more formulas:

$$D_0 : \square(q \equiv \ominus d) \quad D_{20} : \square(\bar{q} \equiv \sim \ominus d)$$

and promised to show, here in §9.4, that the conjunction of D_1 and D_3 is logically equivalent to D_0 . As discussed in §9.2.2, this past-focused version of the D flip-flop equivalence gives us a better understanding of its *delay* functionality. The flip-flop output q is the result of delaying the data input d by one clock tick. It is clear that formula D_{20} is a logical consequence of the conjunction of D_0 and D_2 .

First, we prove $(D_1 \ \& \ D_3) \supset D_0$:

$$(\square(\oplus q \equiv d) \ \& \ \square(q \supset \ominus T)) \supset \square(q \equiv \ominus d)$$

is a validity of LTL_\ominus^1 . Let v be a valuation such that:

$$v \models (\square(\oplus q \equiv d) \ \& \ \square(q \supset \ominus T)).$$

Then under v , we have $q_{n+1} = d_n$ for all n in \mathbb{N} , and $q_0 = 0$. Hence for all n in \mathbb{N} , we have $q_n = 1$ if and only if $(v, n) \models \ominus d$, noting that at $n = 0$, we have $q_0 = 0$ and $(v, 0) \not\models \ominus d$. So we can conclude that $v \models \square(q \equiv \ominus d)$.

Then we prove the converse $D_0 \supset (D_1 \ \& \ D_3)$:

$$\square(q \equiv \ominus d) \supset (\square(\oplus q \equiv d) \ \& \ \square(q \supset \ominus T))$$

is a validity of LTL_\ominus^1 . Let v be a valuation such that $v \models \square(q \equiv \ominus d)$. Then under v , we have $q_n = 1$ if and only if $(v, n) \models \ominus d$. In particular, $q_0 = 0$ and $(v, 0) \not\models \ominus d$. Hence for all n in \mathbb{N} , we have $(v, n) \models \oplus q$ if and only if $d_n = 1$. Thus $v \models (\square(\oplus q \equiv d) \ \& \ \square(q \supset \ominus T))$.

In this way, we derive an equivalent but more concise set of characterising formulas for the D flip-flop.

D flip-flop characteristic formulas:

$$D_0 : \square(q \equiv \ominus d) \quad \text{and} \quad D_2 : \square(\bar{q} \equiv \sim q)$$

9.4.6 | SYSTEM SPECIFICATION IN LTL WITH PREVIOUS

In this last sub-section of §9.4, we make use of LTL_\ominus^1 validities to rewrite past-focused specification formulas into a logically equivalent future-focused form, and vice-versa.

For formulas A and B, we have an LTL_{\ominus}^1 validity scheme:

$$\square(A \equiv \ominus\ominus B) \equiv \square((\oplus A \equiv B) \ \& \ (A \supset \ominus T))$$

The proof of this generalises the argument for the D flip-flop, replacing q with A and d with B.

Dealing with formulas of the kind we have seen in the specification of the triple-1 detector system TID, we can apply this equivalence three times:

$$\begin{aligned} & \square(A \equiv \ominus\ominus\ominus B) \\ \equiv & \square((\oplus A \equiv \ominus\ominus B) \ \& \ (A \supset \ominus T)) \\ \equiv & \square((\oplus\oplus A \equiv \ominus B) \ \& \ (A \supset \ominus T) \ \& \ (\oplus A \supset \ominus T)) \\ \equiv & \square((\oplus\oplus\oplus A \equiv B) \ \& \ (A \supset \ominus T) \ \& \ (\oplus A \supset \ominus T) \ \& \ (\oplus\oplus A \supset \ominus T)) \\ \equiv & \square((\oplus\oplus\oplus A \equiv B) \ \& \ ((A \vee \oplus A \vee \oplus\oplus A) \supset \ominus T)) \end{aligned}$$

finishing off by applying a propositional logical equivalence, to conclude that the bi-conditional as shown is a validity scheme of LTL_{\ominus}^1 relating \oplus and \ominus across three time steps:

$$\square(A \equiv \ominus\ominus\ominus B) \equiv \square((\oplus\oplus\oplus A \equiv B) \ \& \ ((A \vee \oplus A \vee \oplus\oplus A) \supset \ominus T))$$

To show the logical equivalence of:

$$\square((\oplus\oplus\oplus z \equiv (p \ \& \ \oplus p \ \& \ \oplus\oplus p)) \ \& \ ((z \vee \oplus z \vee \oplus\oplus z) \supset \ominus T))$$

and

$$\square(z \equiv (\ominus p \ \& \ \ominus\ominus p \ \& \ \ominus\ominus\ominus p))$$

as required for reconciling the future-focused and past-focused versions of the triple-1 detector specification, we can apply the previous equivalence with A as z and B as $(p \ \& \ \oplus p \ \& \ \oplus\oplus p)$, and note that:

$$\begin{aligned} z \equiv A & \equiv \ominus\ominus\ominus B \\ \equiv & \ominus\ominus(p \ \& \ \oplus p \ \& \ \oplus\oplus p) \\ \equiv & \ominus\ominus p \ \& \ \ominus\ominus\oplus p \ \& \ \ominus\ominus\ominus\oplus p \\ \equiv & \ominus\ominus p \ \& \ \ominus\oplus p \ \& \ \ominus T \ \& \ \ominus p \ \& \ \ominus T \\ \equiv & \ominus\ominus p \ \& \ \ominus\oplus p \ \& \ \ominus p \\ \equiv & \ominus p \ \& \ \ominus\ominus p \ \& \ \ominus\ominus p \end{aligned}$$

These equivalence steps make use of the logical equivalence of $\ominus\oplus A$ with $(A \ \& \ \ominus T)$, and the validity of $(\ominus p \supset \ominus T)$.

9.5 | FORMAL VERIFICATION OF DIGITAL SYSTEMS

9.5.1 | SPECIFICATION, DESCRIPTION & VERIFICATION OF DIGITAL SYSTEMS

In our chosen logic LTL_{\ominus}^1 , we can:

- Formalise *system specifications* of input–output behaviour as logic formulas.
- Formalise *system descriptions* of next-state and output behaviour as logic formulas in terms of current-state and current input.
- Formalise as logic formulas *system descriptions* of current-state and output in terms of previous input and previous state.
- *Formally verify* that a specification holds of a system in response to all possible input signals by proving validity status for formulas of form $(\Box D_{sys} \supset \Box A_{spec})$

Before getting to formal verification, here is an exercise for you in translating specifications into the language of LTL_{\ominus}^1 .

Consider a clocked sequential system with two 1-bit input signals p_1 and p_2 , and a 1-bit output signal z , such that:

for all time points n , the output signal z is 1 at time n , if and only if, time $n \geq 3$ and the inputs p_1 and p_2 are the same value (both 0 or both 1) at times $(n - 1)$ and $(n - 3)$, and are opposite in value at time $(n - 2)$.

Which of the following formulas express this functional description:

- (a) $\Box(\oplus\oplus\oplus z \equiv ((p_1 \& p_2) \& \oplus(p_1 \& \sim p_2) \& \oplus\oplus(p_1 \& p_2)))$
 (b) $\Box(z \equiv (\ominus(p_1 \equiv p_2) \& \ominus\ominus(p_1 \equiv p_2) \& \ominus\ominus\ominus(p_1 \equiv p_2)))$
 (c) $\Box(z \equiv (\ominus(p_1 \equiv p_2) \& \ominus\ominus(p_1 \equiv \sim p_2) \& \ominus\ominus\ominus(p_1 \equiv p_2)))$
 (d) $\Box(\oplus\oplus\oplus z \equiv ((p_1 \equiv p_2) \& \oplus(p_1 \equiv \sim p_2) \& \oplus\oplus(p_1 \equiv p_2)))$
 $\& \Box((z \vee \oplus z \vee \oplus\oplus z) \supset \ominus T)$

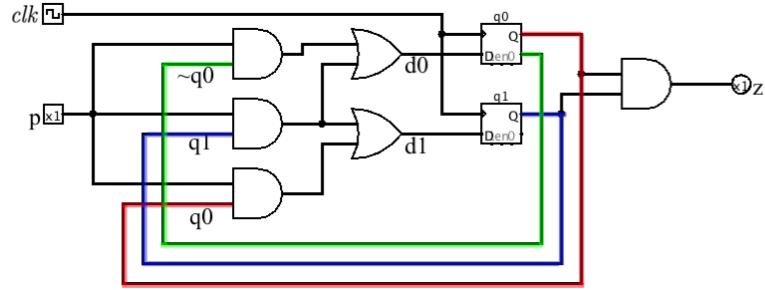
The correct formulas are (c) and (d). (c) expresses the past-focused version of the specification, while (d) expresses the future-focused version of the specification, which includes the extra condition $((z \vee \oplus z \vee \oplus\oplus z) \subset \ominus T)$ ensuring that time $n \leq 3$ if z is 1.
 (a) is incorrect because $(p_1 \not\equiv p_2)$ is not enough to ensure p_1 and p_2 have the same value; for that, you need $(p_1 \not\equiv p_2) \wedge (\sim p_1 \not\equiv \sim p_2)$ or $(p_1 \equiv p_2)$.
 (b) is incorrect because $\ominus(p_1 \equiv p_2)$ says p_1 and p_2 have the same value at time $(n - 2)$; you need $\ominus\ominus(p_1 \equiv p_2)$ for them to have opposite value at time $(n - 2)$.

9.5.2 | FORMAL VERIFICATION OF SYSTEM T1D

Our main job for the rest of §9.5 is to work through the formal verification of the triple-1 detector system T1D. We will finish off with

a brief but more general discussion of the enterprise of verifying and validating digital systems.

Recall the now very familiar TID circuit diagram:



and the next-state and output equivalences for TID implemented in the circuit, which were derived from the state-transition/output table for the system:

$$\begin{aligned}\oplus q_1 &\equiv d_1 \equiv ((p \wedge q_1) \vee (p \wedge \neg q_0)) \\ \oplus q_0 &\equiv d_0 \equiv ((p \wedge q_1) \vee (p \wedge \neg q_0)) \\ z &\equiv (q_1 \wedge q_0).\end{aligned}$$

We will use the future-focused version of the specification:

$$A_{\text{spec}} : \quad \oplus\oplus\oplus z \equiv (p \wedge \oplus p \wedge \oplus\oplus p).$$

From our work in §9.4.5 on validities of the logic relevant to D flip-flops, we can use the past-focused version of the system description.

$$D_{\text{TID}} : \quad \left(\begin{array}{l} (q_1 \equiv \ominus((p \wedge q_1) \vee (p \wedge \neg q_0))) \\ \& (q_0 \equiv \ominus((p \wedge q_1) \vee (p \wedge \neg q_0))) \\ \& (z \equiv (q_1 \wedge q_0)) \end{array} \right)$$

For the *formal verification* of system TID, we need to establish the validity of the conditional:

$$\text{TID verification: } (\Box D_{\text{TID}} \supset \Box A_{\text{spec}}).$$

To piece together the verification, we go back to §9.3.4 and the state-transition/output table for the system, since this was how we derived the future-focused version of formula D_{TID} in §9.3.5. Remember that the state-transition table contains all the content needed to produce the circuit and the description formula for the system. Back in §9.3.3, we generated the content needed for the next-state entries of that table by assigning meaning to each of the four state configurations, and working through the state transitions between them. Back then, we had not yet formalised the semantics of temporal logic formulas, so we used a mix of temporal formulas with the *Previous* operator \ominus and the time conditions $n = 0$, $n = 1$ or $n = 2$. We can now fully formalise these state configuration meanings.

$q_1 q_0$	configuration to be remembered
00	Got nothing: $\ominus\text{p} \vee \sim\ominus\top$
01	Got one 1: $\text{op} \& \ominus(\ominus\text{p} \vee \sim\ominus\top)$
10	Got 11: $(\text{op} \& \ominus\ominus\text{p}) \& \ominus\ominus(\ominus\text{p} \vee \sim\ominus\top)$
11	Got 111: $(\text{op} \& \ominus\ominus\text{p} \& \ominus\ominus\ominus\text{p})$
output z :	$z \equiv (\text{q}_1 \& \text{q}_0)$ Moore output

Let C_{TID} be the conjunction of these current-state/output formulas:

$$C_{TID} : \left(\begin{array}{l} (\sim\text{q}_1 \& \sim\text{q}_0) \equiv (\ominus\text{p} \vee \sim\ominus\top) \\ \& (\sim\text{q}_1 \& \text{q}_0) \equiv (\text{op} \& \ominus(\ominus\text{p} \vee \sim\ominus\top)) \\ \& (\text{q}_1 \& \sim\text{q}_0) \equiv (\text{op} \& \ominus\ominus\text{p} \& \ominus\ominus(\ominus\text{p} \vee \sim\ominus\top)) \\ \& (\text{q}_1 \& \text{q}_0) \equiv (\text{op} \& \ominus\ominus\text{p} \& \ominus\ominus\ominus\text{p}) \\ \& z \equiv (\text{q}_1 \& \text{q}_0) \end{array} \right)$$

Note that this formula C_{TID} expresses our *design intention*, but it does not come for *free*; we have to prove it. Clearly, if we could prove that $(\square D_{TID} \supset \square C_{TID})$ is a validity, then we would be done, because $\square C_{TID}$ implies the equivalent past-focused version of the specification.

However, this is proving more than we need, and would take more time than we have. What we really want to pull out of formula C_{TID} is just the last two conjuncts, since these two equivalences together give us what we want. Let B_{TID} be the formula:

$$B_{TID} : \left(\begin{array}{l} (\text{q}_1 \& \text{q}_0) \equiv (\text{op} \& \ominus\ominus\text{p} \& \ominus\ominus\ominus\text{p}) \\ \& z \equiv (\text{q}_1 \& \text{q}_0) \end{array} \right)$$

and let A_{spec}^* be the equivalent past-focused version of the specification:

$$A_{\text{spec}}^* : z \equiv (\text{op} \& \ominus\ominus\text{p} \& \ominus\ominus\ominus\text{p})$$

It is easy to see validity of:

$$(\square B_{TID} \supset \square A_{\text{spec}}^*) .$$

In §9.4.6, we established the validity of:

$$(\square A_{\text{spec}}^* \supset \square A_{\text{spec}})$$

since $(\square A_{\text{spec}}^* \equiv (\square A_{\text{spec}} \& \square((z \vee \oplus z \vee \oplus \oplus z) \supset \ominus\top)))$.

To establish validity status for the conditional:

$$\text{TID verification: } (\square D_{TID} \supset \square A_{\text{spec}})$$

it remains to prove the validity of:

$$(\square D_{TID} \supset \square B_{TID}) .$$

To fill in this last piece, first observe that there is a valid equivalence between the description formula D_{TID} and the formula D_{TID}^* as shown, which is obtained by distributing the \ominus operator over conjunction and disjunction, and applying the tautology expressing distribution of AND over OR.

$$D_{TID}^* : \begin{array}{l} \begin{array}{ll} q1 & \equiv (\ominus p \ \& \ (\ominus q1 \vee \ominus q0)) \\ \& (\ominus q0 \equiv (\ominus p \ \& \ (\ominus q1 \vee \ominus \sim q0))) \\ \& (z \equiv (q1 \ \& \ q0)) \end{array} \end{array}$$

Furthermore, we also have the validity of $(\Box D_{TID}^* \supset \Box E_{TID})$, where:

$$E_{TID} : (q1 \ \& \ q0) \equiv (\ominus p \ \& \ \ominus q1)$$

This validity can be shown either directly using the semantics, or else by a series of logical equivalences.

So to finish, it suffices to establish the validity of:

$$(\Box D_{TID}^* \supset \Box B_{TID}).$$

From D_{TID}^* , we can “fully unwind” the state signal $q1$, so that there is no dependence on $q1$ or $q0$, only on input p .

$$\begin{aligned} q1 &\equiv \ominus p \ \& \ (\ominus q1 \vee \ominus q0) \\ &\equiv \ominus p \ \& \ (\ominus (\ominus p \ \& \ (\ominus q1 \vee \ominus q0)) \vee \ominus (\ominus p \ \& \ (\ominus q1 \vee \ominus \sim q0))) \\ &\equiv \ominus p \ \& \ ((\ominus \ominus p \ \& \ (\ominus \ominus q1 \vee \ominus \ominus q0)) \vee (\ominus \ominus p \ \& \ (\ominus \ominus q1 \vee \ominus \ominus \sim q0))) \\ &\equiv \ominus p \ \& \ \ominus \ominus p \ \& \ (\ominus \ominus q1 \vee \ominus \ominus q0 \vee \ominus \ominus \sim q0) \\ &\equiv \ominus p \ \& \ \ominus \ominus p \ \& \ (\ominus \ominus q1 \vee \ominus \ominus T) \\ &\equiv \ominus p \ \& \ \ominus \ominus p \ \& \ (\ominus \ominus (q1 \vee T)) \\ &\equiv \ominus p \ \& \ \ominus \ominus p \ \& \ \ominus \ominus T \\ &\equiv \ominus p \ \& \ \ominus (\ominus p \ \& \ T) \\ &\equiv \ominus p \ \& \ \ominus \ominus p \end{aligned}$$

The series of equivalences starts with the $q1$ formula from D_{TID}^* , then on the second line, we substitute what we know from D_{TID}^* into the $q1$ and $q0$ spots in the formula, and then start distributing the \ominus operators over conjunctions and disjunctions. Note that $(A \vee T)$ is equivalent to A , and that $(A \ \& \ T)$ is equivalent to A . In this way, we end up with the equivalence $(q1 \equiv (\ominus p \ \& \ \ominus \ominus p))$, which says $q1$ has value 1 exactly when the previous two input values of p were 1. Note that this formula is a “fully unwound” characterisation of $q1$ because there is no longer any circular dependence of $q1$ on $\ominus q1$ and $\ominus q0$, so we have “unwound” the feedback loops.

We can conclude that validity is established for the conditional:

$$(\Box D_{TID}^* \supset \Box (q1 \equiv (\ominus p \ \& \ \ominus \ominus p))).$$

Using the equivalence $\Box (q1 \equiv (\ominus p \ \& \ \ominus \ominus p))$ and the substitutivity of equivalents applied to formula E_{TID} , we get:

$$\begin{aligned} (q1 \ \& \ q0) &\equiv (\ominus p \ \& \ \ominus q1) \\ &\equiv (\ominus p \ \& \ \ominus (\ominus p \ \& \ \ominus \ominus p)) \\ &\equiv (\ominus p \ \& \ \ominus \ominus p \ \& \ \ominus \ominus \ominus p) \end{aligned}$$

So finally we are done, having established the validity of:

$$(\Box D_{\text{TID}}^* \supset \Box B_{\text{TID}})$$

since B_{TID} is the conjunction of $(q_1 \& q_0) \equiv (\ominus p \& \ominus\ominus p \& \ominus\ominus\ominus p)$ and $z \equiv (q_1 \& q_0)$.

Congratulations!! You have made it all the way through the formal verification of the triple-1 detector system TID. As promised, it does take a bit of effort to formally verify even a small sequential system with four state configurations.

9.5.3 | VERIFICATION & VALIDATION METHODS FOR DIGITAL SYSTEMS

We finish up by briefly looking more generally at the enterprise of verifying and/or validating digital systems.

- *simulation and testing* of a system design on a sample of test input signals to check the system produces the expected and specified output signals in response. While computer engineers have developed extensive principles for the strategic selection of useful test input signals, this approach is still fundamentally limited by the fact that it only considers a small sample of the infinitely many possible input signals.
- *formal verification* in a logic or other formal system rigorously proving that a system design implements the specification on all of the infinitely many possible input signals. In practice, formal verification is particularly valuable in detecting subtle errors in the design that would be difficult to pick up via simulation and testing.
- *timing analysis* to ensure real time constraints on synchronous memory components can be satisfied. For the correct behaviour of basic memory components like the D flip-flop, there are constraints on the *setup time* and the *hold time* for the device, as discussed at the end of §9.2.1. While these times are much smaller than the real-time period of the clock, they are not 0, so the accumulated propagation delays through the next-state combinational sub-circuit do have to be analysed carefully. This sort of analysis is beyond the scope of discrete-time temporal logic and of us here.
- *hardware testing* to search for fabrication errors resulting from layout defects. Unlike the first three methods that are all used at the design stage, before the computer chip is fabricated, hardware testing is what happens after physical construction.

As we have seen with the triple-1 detector system TID, doing formal verification “by hand” for even a quite small digital system requires a non-trivial amount of effort. Clearly some computer assistance would be helpful!

There are a number of both commercial and educational open-source software tools that can be used for the formal verification of digital systems using temporal logic. (Many of the same tools can also be used for the formal verification of computer *software* using temporal logic.) Leading hardware industry companies like IBM, Intel, Samsung and Motorola regularly use formal verification methods, particularly in the design of critical components whose failure could have catastrophic consequences – for human life, property or financial loss.

The most common approach goes under the title *model-checking*. “Model-checking” is really just a verb-phrase to describe the task of determining truth in a model or valuation. All the hard work goes into *representing* models or valuations. Within state-of-the-art model-checking tools, explicit model-checking can be done for up to 30 state signals (2^{30} state values), while implicit or symbolic representations of states allow up to 100 state signals or more (so 2^{100} state values).

FURTHER LEARNING

This brings us to the end of this chapter on the logic of sequential digital systems. We have seen how sequential systems build on combinational systems by adding memory components, a digital clock, state signals and feedback. And we have seen how to specify, describe and formally verify sequential digital systems in temporal logic.

We hope we have achieved the modest aim of providing a basic understanding of digital systems by building on your knowledge of core logic. This chapter and lecture series ended up longer than intended, because getting through the formal verification of even a simple system like the triple-1 detector takes a fair amount of effort. We hope that this has not been off-putting, and you have found the effort worthwhile!

For those with a deeper interest in digital systems, there is a wealth of textbooks and online resources available. These include:

- *Coursera*: introductory undergraduate level 10 weeks course:
Digital Systems - Sistemas Digitales: De las puertas lógicas al procesador bilingual Spanish and English, Universitat Autònoma de Barcelona
<https://www.coursera.org/course/digitalsystems>
- 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-sciences/6-111-introductory-digital-systems-laboratory-spring-2006> (Accessed 5 Mar, 2014). License: CC BY-NC-SA
- Crisp, John. *Introduction to Digital Systems* (Elsevier, 2000); widely available textbook.

REFERENCES

- [1] 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 93]
- [2] JON BARWISE AND ROBIN COOPER. “Generalized Quantifiers and Natural Language”. *Linguistics and Philosophy*, 4:159–219, 1981. [Cited on page 242]
- [3] JC BEALL AND GREG RESTALL. *Logical Pluralism*. Oxford University Press, Oxford, 2006. [Cited on page 114]
- [4] JEROEN GROENENDIJK AND MARTIN STOKHOF. “Dynamic Predicate Logic”. *Linguistics and Philosophy*, 14:39–100, 1991. [Cited on page 246]
- [5] COLIN HOWSON. *Logic with Trees: An introduction to symbolic logic*. Routledge, 1996. [Cited on pages 7, 93]
- [6] ALEXIUS MEINONG. *On Assumptions*. University of California Press, Berkeley and Los Angeles, California, 1983. Translated and edited by James Heanue. [Cited on page 211]
- [7] STANLEY PETERS AND DAG WESTERSTÅHL. *Quantifiers in Language and Logic*. Clarendon Press, Oxford, 2006. [Cited on pages 20, 241]
- [8] GREG RESTALL. *Logic*. Fundamentals of Philosophy. Routledge, 2006. [Cited on page 7]
- [9] BERTRAND RUSSELL. “On Denoting”. *Mind*, 14:479–493, 1905. [Cited on pages 214, 215]
- [10] STEWART SHAPIRO. *Foundations without Foundationalism: A case for second-order logic*. Oxford University Press, 1991. [Cited on page 113]
- [11] R. M. SMULLYAN. *First-Order Logic*. Springer-Verlag, Berlin, 1968. Reprinted by Dover Press, 1995. [Cited on page 93]

IMAGE ACKNOWLEDGEMENTS



WombatZoo.JPG by *Materialsscientist* (own work) [CC-BY-3.0]

<http://commons.wikimedia.org/wiki/File:WombatZoo.JPG>



Bust of Aristotle, Roman copy after a Greek bronze original by Lysippos (330 BCE), photographer *Marie-Lan Nguyen* (2006) [PUBLIC DOMAIN] via *Wikimedia Commons*

http://commons.wikimedia.org/wiki/File:Aristotle_Altemps_Inv8575.jpg



Ada Lovelace [PUBLIC DOMAIN]

<http://people.cs.kuleuven.be/~dirk.craeynest/ada-belgium/pictures.html>



White Night Melbourne, by *Chris Phutully* (2013) [CC-BY-2.0]

<http://www.flickr.com/photos/72562013@N06/8500995539/>



Photograph of Gottlob Frege, circa 1879. Photographer unknown. [PUBLIC DOMAIN] via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Young_frege.jpg



President Barack Obama, 2012 portrait crop

Official White House Photo by Pete Souza (P120612PS-0463)
(direct link) [Public Domain] via *Wikimedia Commons*

http://commons.wikimedia.org/wiki/File:President_Barrack_Obama,_2012_portrait_crop.jpg



Xi Jinping, Mexico 2013. Photograph by Angélica Rivera de Peña (original photo). Modifications by ASDFGH [CC-BY-SA-2.0] via *Wikimedia Commons*

http://en.wikipedia.org/wiki/File:Xi_Jinping_Mexico2013.jpg



Shinzo Abe, cropped by Fæ (original photo)
modifications by Lq12 [Public Domain] via *Wikimedia Commons*
http://en.wikipedia.org/wiki/File:Shinzo_Abe_cropped.JPG



Angela Merkel (2008), the Chancellor of Germany
photograph by N (Aleph): own work, April 2008
via *Wikimedia Commons* [CC-BY-SA-2.5]
[http://en.wikipedia.org/wiki/File:Angela_Merkel_\(2008\).jpg](http://en.wikipedia.org/wiki/File:Angela_Merkel_(2008).jpg)



Goldbach partitions of the even integers from 4 to 50,
artists: Adam Cunningham and John Ringland
via *Wikimedia Commons* [CC-BY-SA-3.0]

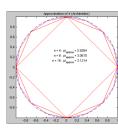
http://en.wikipedia.org/wiki/File:Goldbach_partitions_of_the_even_integers_from_4_to_50_rev4b.svg



Kurt Gödel and Albert Einstein, Princeton 1950,
photographer unknown [PUBLIC DOMAIN]
<http://kgs.logic.at/index.php?id=23>



U2_21081983_01_800b.jpg, by Helge Øverås (own work) [CC-BY-SA-3.0], via *Wikimedia Commons*
http://commons.wikimedia.org/wiki/File:U2_21081983_01_800b.jpg



Archimedes approximation of π . Graphic generated from MATLAB code
by Markus Körner and Andreas Klimke, Stuttgart University, 2002.
http://m2matlabdb.ma.tum.de/download.jsp?MC_ID=9&MP_ID=88



Laptop by mystica [PUBLIC DOMAIN]
<http://openclipart.org/detail/15413/laptop-by-mystica>



Kurt Gödel and Albert Einstein, Princeton 1950,
photographer unknown [PUBLIC DOMAIN]
<http://kgs.logic.at/index.php?id=23>



Alonzo Church,
photographer unknown [copyright status TBD]
<http://www.cs.berkeley.edu/~bh/ss-pics/alonzo.jpg>



Photograph of The Stacked Slate Sculpture of Alan Turing by Stephen Kettle,
Photograph by Jon Callas from San Jose, USA, [CC BY 2.0]
via Wikimedia Commons
http://commons.wikimedia.org/wiki/File:Alan_Turing.jpg



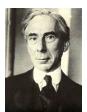
SWI-Prolog logo, University of Amsterdam [CC BY SA 3.0]
<http://www.swi-prolog.org/>



IBM Watson computer, by *Clockready* (Own work)
[CC-BY-SA-3.0] via Wikimedia Commons
http://en.wikipedia.org/wiki/File:IBM_Watson.PNG



Portrait of Louis XVI by Antoine-François Callet, [PUBLIC DOMAIN] via
Wikimedia Commons.
http://commons.wikimedia.org/wiki/File:Ludvig_XVI_av_Frankrike_porträtterad_av_AF_Callet.jpg



Bertrand Russell in 1924, photographer unknown [PUBLIC DOMAIN],
via Wikimedia Commons.
http://commons.wikimedia.org/wiki/File:Bertrand_Russell_in_1924.jpg



Pegasus on roof of Poznan Opera House by user:Radomil (own work),
[CC-BY-SA-3.0] via Wikimedia Commons.



Godzilla statue tokyo.JPG by *Wikiodaiba* (own work) [CC-BY-3.0]

http://commons.wikimedia.org/wiki/File:Godzilla_statue_tokyo.JPG



Photograph of Alexius Meinong (1853–1920), date unknown,
Photographer unknown, [PUBLIC DOMAIN] via *Wikimedia Commons*.

<http://commons.wikimedia.org/wiki/File:Meinong.jpg>



Golden Mountains, by Doun Dounell [CC-BY-2.0], via *Flickr*.

http://www.flickr.com/photos/doun_dounell/6949055387



Dutch Beers by *Nejmlez* (own work) [CC-BY-SA-3.0] via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Dutch_beers.jpg



GRAZ, by *Gerhard Langusch* (own work) [CC-BY-SA-3.0] via *Wikimedia Commons*.

http://commons.wikimedia.org/w/index.php?title=File:Band_GRAZ.jpg



Circus Elephants and Riders at Rosewood, circa 1928, [PUBLIC DOMAIN] via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:StateLibQld_2_76842_Circus_elephants_and_riders_at_Rosewood,_ca._1928.jpg



Schoolgirls in Bamozaï, Afghanistan, by Captain John Severns, U.S. Air Force, [PUBLIC DOMAIN], via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Schoolgirls_in_Bamozaï.JPG



Tenzing Norgay by *Dirk Pons* (own work) [CC-BY-SA-3.0] via *Wikimedia Commons*.

<http://commons.wikimedia.org/wiki/File:Tenzing.jpg>



Everest North Face toward Base Camp Tibet, by *Luca Galuzzi* [CC-BY-SA-2.5], via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Everest_North_Face_toward_Base_Camp_Tibet_Luca_Galuzzi_2006.jpg



Summit of Mount Kosciuszko, by *Cimexu* (own work) [CC-BY-2.0], via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Summit_of_Mount_Kosciuszko.jpg



Mule in Moriles, Córdoba (Spain), by *Juan R. Lascorz* (own work) [CC-BY-SA-3.0], via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:09.Moriles_Mula.JPG



Archery Target, by *Casito* (own work) [CC-BY-SA-3.0], via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Archery_target.jpg



Advertisement for the Georgian Room restaurant on the ninth floor of the Eaton's department store at Yonge and Queen Streets, Toronto, Canada (1924), artist unknown, [PUBLIC DOMAIN] via *Wikimedia Commons*.

http://commons.wikimedia.org/wiki/File:Advertisement_for_the_Georgian_Room_restaurant_at_Eaton's.JPG



Earth globe above tech landscape by *Steve Johnson* [CC BY 2.0]

<http://www.flickr.com/photos/artbystevejohnson/4607812450/>



“Flip-flop splashed by a wave” by *Bermi Ferrer*, April 2008

[CC BY 2.0]

<https://www.flickr.com/photos/bermilabs/2404979521/>



Digital photograph by *stefran*, February 2012 [Public Domain CC0]

<http://pixabay.com/en/thongs-beach-flip-flops-footwear-264411/>