Arguments

Arguments are everywhere: from such-and-such assumptions follow these-and-those conclusions. Arguments vary in their level of formality, but they all follow similar rules:

- They start from assumptions;
- They proceed in a sequence of steps;
- For each step, they introduce some new conclusions that are claimed to follow from earlier facts (that is, from assumptions and previous conclusions);
- Each step is at least potentially checkable.

The more formal an argument is, the more assiduously it follows these rules: an informal argument might skip steps (in the hopes that the reader or listener will fill them in), or might include steps that are heuristic (don't work 100% of the time, and therefore can't be perfectly checked). A formal argument will try to be more detailed and rigorous.

Even quite formal arguments, though, tend to take shortcuts. It's moderately rare to see an argument that is so completely and rigorously written out that every step is perfectly specified and airtight, since such an argument would be so long and detailed that nobody would bother to follow the whole thing.

> *In fact, the only place where fully-detailed arguments are commonly encountered is inside a computer-assisted proof system, such as* L∃∀N*, Coq, or Keymaera. These systems try to make proofs both human-readable and perfectly formal by hiding details that an automated proof search can fill in, and expanding hidden details on demand.*

Even when we hide them, though, the details are what makes or breaks an argument. Just like a bug in a program, a single missing detail in an argument can render the whole thing useless. It's even worse when the missing detail is hidden inside a step we skipped over.

For this reason, every time we take a shortcut in an argument, we want to make sure we understand how the longer, fully-rigorous version of the argument would work — else a bug could be hidden in the part we skipped. When in doubt, write it out: if you're not completely sure that a step works, then even if it does, many of your readers or listeners may have trouble following it.

A final thing to remember about all arguments: their purpose is to convince the listener.

That means that the style of an argument is important, not just its content. For example, it's a good idea to

- Keep an *audience model*, a mental picture of what the readers or listeners already know vs. what we need to convince them of;
- Organize arguments into digestible chunks that can be understood independently;
- Be clear: state what is an assumption, what each new conclusion depends on, and what reasoning method is being used at each step;
- Annotate the structure, so that the reader or listener can understand at every point what conclusions we're currently trying to argue for.

Formal logic

We're going to try to understand arguments by developing a completely precise system for describing them: the system of *formal logic*. An argument in formal logic is called a *proof*.

There's not just one system of formal logic. Instead, there are many related systems, each with a different focus and purpose. Examples include:

- Classical propositional logic
- Constructive propositional logic
- Classical first-order logic
- Linear logic
- Temporal logic
- ... and many more.

Different systems might be able to represent different concepts, or might be able to represent the same concepts more or less efficiently. Different systems might even contradict each other: the same conclusion might be provable in one system but not another.

> *An example of this kind of difference is classical vs. constructive logic: the inference system for constructive logic allows only constructive proofs, and so there are some conclusions that we can prove in classical logic that turn out not to hold in constructive logic.*

But, these systems all share some features. The biggest is that they all have two complementary parts: first, a language for stating our assumptions and conclusions, and

second, a set of rules for manipulating these statements and using them to construct proofs.

Perhaps the simplest and best known of these systems is classical propositional logic; it's also at the heart of many other, more complicated reasoning systems. So, we'll start there.

Propositional logic expressions

In propositional logic, variables $a, b, \ldots$ (called *propositions*) represent truth values T or F. These variables can be combined using *connectives* such as $\vee, \wedge, \neg, \rightarrow$, meaning AND, OR, NOT, IMPLIES. The expressions of propositional logic are exactly the ones we can construct recursively by starting from propositions and applying connectives: for example, $a \wedge \neg b$.

The precedence is that $\neg$ binds tightest, followed by $\vee, \wedge$ (equal precedence), followed by $\rightarrow$. In case of ambiguity, or if we want a different order, we can use parentheses for grouping: $[(\{\})]$.

There are alternate notations for many of these concepts: you may see $\bar{a}$ or $\sim a$ instead of $\neg a$; you may see $a \& b$ or ab instead of $a \wedge b$; you may see $a \mid b$ or $a + b$ instead of $a \vee b$; you may see $a \supset b$ or $a \Rightarrow b$ instead of $a \rightarrow b$; and you may see $\top, \bot$ instead of T, F. Falsehood F is sometimes also called *absurdity* or *contradiction*.

For readability, we'll often use longer variable names: e.g., done or $\mathrm{happy}(\mathrm{John})$. The latter name looks like a function call, and in fact we'll give it that semantics later; but for now it is just a string of symbols that represents a propositional variable.

Propositional logic semantics

We can interpret each expression of propositional logic as a true-or-false statement that we might have evidence for. For example, if we assert the expression a, we are claiming to have have evidence for the proposition a.

Similarly, if we assert $a \wedge b$, we are claiming to have evidence for both a and b. If we assert $a \vee b$, it means that we have evidence for either a or b (or both). And, $\neg a$ means that we have evidence that a cannot be true.

The connective that causes the most confusion is $\rightarrow$. If I state $a \rightarrow b$, I am making a

promise: if you give me evidence for a, I will provide evidence for b. The confusing part is what happens when a is false. In this case, nobody can provide me evidence for a. So, $a \to b$ is an empty promise: I'll never have to provide evidence for b. Therefore, in classical propositional logic, whenever $\neg a$ holds, $a \to b$ is vacuously true.

Because of this semantics, in classical logic, $a \to b$ is equivalent to $\neg a \vee b$. That is, out of all possible settings for a and b, the only one that makes $a \to b$ false is when a is true and b is false:

a	b	$a \to b$
F	F	T
F	T	T
T	F	F
T	T	T

Inference rules

To work with statements and evidence, we use inference rules. The most famous rule is probably *modus ponens*: from premises ϕ and $\phi \to \psi$, conclude ψ. For example, from $\mathrm{dog}(\mathrm{Spot})$ and $\mathrm{dog}(\mathrm{Spot}) \to \mathrm{fuzzy}(\mathrm{Spot})$, we would conclude $\mathrm{fuzzy}(\mathrm{Spot})$. We can translate modus ponens to natural language as: if I have evidence for ϕ, and if I have a way to translate evidence for ϕ into evidence for ψ, then I can get evidence for ψ.

All inference rules have this same general structure: given some premises, we can draw some conclusions. A premise is a statement that we already have evidence for; perhaps it was an assumption, or perhaps we constructed evidence for it in a previous step. A conclusion is a new statement that we are constructing evidence for.

Each inference rule gives us a template to follow. That is, the premises and the conclusions must fit given forms. For modus ponens, our premises have to look like ϕ and $\phi \to \psi$.

The symbols ϕ and ψ in the template can represent single propositions or more complicated expressions containing connectives. We took $\phi = \mathrm{dog}(\mathrm{Spot})$ and $\psi = \mathrm{fuzzy}(\mathrm{Spot})$ above, but if we had a longer implication like

$$\mathrm{dog}(\mathrm{Spot}) \wedge \mathrm{happy}(\mathrm{Spot}) \to \mathrm{wags}(\mathrm{Spot})$$

we'd want to take ϕ to be the compound expression $\mathrm{dog}(\mathrm{Spot}) \wedge \mathrm{happy}(\mathrm{Spot})$.

Modus ponens is sometimes also called $\rightarrow$ elimination, since the premises contain an implication but the conclusion does not.

Proofs

If we chain several inference rules together, we get a *proof*. We will write a proof as a sequence of lines, where each line is either an assumption or an application of an inference rule. We can number or label the lines so that we can refer back to them. For example:

1. Assume $\mathrm{dog}(\mathrm{Spot})$.
2. Assume $\mathrm{dog}(\mathrm{Spot}) \rightarrow \mathrm{fuzzy}(\mathrm{Spot})$.
3. Conclude $\mathrm{fuzzy}(\mathrm{Spot})$ by modus ponens from 1, 2.

Each line in the proof must have a justification! The only possible kinds of justification are the two above: either a line is an assumption, or it follows from some previous lines via an inference rule. Sometimes people omit justifications or skip steps when they are clear from context; but beware: "clear from context" is in the eye of the beholder.

If a proof is a bit longer than the one above, it can be worth annotating its structure: e.g., adding blank lines between sections, indenting parts or drawing boxes around them to indicate grouping, and including comments. We'll see examples of this sort of annotation below.

More inference rules

To get a complete logical system, we'll need some inference rules to work with each of our constants and connectives. There are multiple equivalent systems of inference rules for classical propositional logic, but for concreteness we'll describe a particular one here.

No matter which system we use, it's important to pick one and stick with it: we should always know which inference rules are allowed, and use only those. We can introduce subtle errors into our reasoning if we accidentally mix two different systems of rules.

We'll start with rules for introducing and eliminating $\vee$ and $\wedge$. For $\vee$:

- $\wedge$ introduction: for any expressions ϕ and ψ, if we separately prove ϕ and ψ, that constitutes a proof of $\phi \wedge \psi$.
- $\wedge$ elimination: if we know $\phi \wedge \psi$, then we can conclude ϕ, and we can also conclude ψ.

For $\vee$:

- $\vee$ introduction: from ϕ we can conclude $\phi \vee \psi$ for any ψ. Similarly, we can also conclude $\psi \vee \phi$.
- $\vee$ elimination (also called proof by cases): if we know $\phi \vee \psi$, and if we have both $\phi \to \chi$ and $\psi \to \chi$, then we can conclude χ. Here ϕ and ψ are the cases; if we can prove a desired conclusion χ in both cases, then χ holds, even if we don't know which case we are in.

We have an introduction rule for T and an elimination rule for F. The introduction rule for T is perhaps the simplest of all of our rules:

- T introduction: from no premises, we can conclude T.

The elimination rule for F is a bit more conterintuitive, but we'll see examples below of where we need it.

- F elimination: from the assumption F, we can conclude an arbitrary expression ϕ. This rule is sometimes called *ex falso* or *ex falso quodlibet*, meaning "from falsehood, anything" in Latin. One way to think about ex falso is this: we should never be able to prove F, so there's no danger in having F let us conclude an arbitrary formula.

> *You may notice that we haven't given any rules for $\neg$, and we haven't given an introduction rule for $\to$. We'll say more about how to handle $\neg$ and $\to$ below, but it requires a bit more setup and discussion first.*

From the above rules we can derive a few other useful ones, like:

- Associativity: both $\vee$ and $\wedge$ are associative, meaning that it doesn't matter how we parenthesize an expression like $a \vee b \vee c \vee d$. This rule is so common that we often skip over it: we just leave out parentheses when they don't matter, as we did above.
- Distributivity: the connectives $\vee$ and $\wedge$ distribute over one another. For example, $a \wedge (b \vee c)$ is equivalent to $(a \wedge b) \vee (a \wedge c)$.
- Commutativity: both $\vee$ and $\wedge$ are symmetric in the order of their arguments. So for

example $b \vee c \vee a$ is equivalent to $a \vee b \vee c$.

- Idempotence: repeated arguments of $\vee$ or $\wedge$ are superfluous. So for example, $a \vee b \vee b$ is equivalent to $a \vee b$.

These last few rules are a great example of where we might want to simplify a proof by grouping multiple steps and skipping explicit justifications: it would take forever if we listed out every application of associativity and commutativity. Nonetheless, we might ask you to be this detailed in some of our exercises, just for practice.

> Exercise: prove each of the above derived rules. For example, to prove associativity, assume $(a \vee b) \vee c$, and show $a \vee (b \vee c)$. Be fully detailed, and use only the introduction and elimination rules given above, even if you've learned other useful inference rules in previous courses.

Lemmas

One of the most useful tools for organizing a proof is a lemma: we package up a part of our proof so that we can use it repeatedly. To prove a lemma, we start by making some assumptions; call them $\phi_1, \phi_2, \ldots, \phi_k$. We proceed from these assumptions, using them along with any other known facts to reach some conclusion ψ. Finally, we package up our lemma: we conclude

$$\phi_1 \wedge \phi_2 \wedge \ldots \wedge \phi_k \rightarrow \psi$$

We then get to use this lemma as often as we want later on.

For example, here's a proof of associativity of $\wedge$:

1. Assume $(a \wedge b) \wedge c$.
2. Conclude c from 1 by $\wedge$ elimination.
3. Conclude $(a \wedge b)$ from 1 by $\wedge$ elimination.
4. Conclude a from 3 by $\wedge$ elimination.
5. Conclude b from 3 by $\wedge$ elimination.
6. Conclude $b \wedge c$ from 5 and 2 by $\wedge$ introduction.
7. Conclude $a \wedge (b \wedge c)$ from 4 and 6 by $\wedge$ introduction.
8. We have now proved the lemma $[(a \wedge b) \wedge c] \rightarrow [a \wedge (b \wedge c)]$ from lines 1-7.

This recipe for proving a lemma is our introduction rule for $\rightarrow$: it is the way to prove any

expression that contains an implication. So, we could alternately write the last line as

8. Conclude $[(a \wedge b) \wedge c] \rightarrow [a \wedge (b \wedge c)]$ from 1-7 by $\rightarrow$ introduction.

Note how we used whitespace to organize our proof: we indented every line between the assumption and the final statement of the lemma.

Scoping

When we use an assumption to prove a lemma via an application of $\rightarrow$ introduction, it's called *discharging* the assumption. To help organize our proofs, we will make the convention that each assumption needs to be discharged exactly once; we can use it as many times as we want between the line where we assume it and the line where we discharge it. (We can always make the same assumption more than once, if we want to discharge it more than once.)

The region where we can use an assumption is called its *scope*. We will restrict scopes to be *nested*: that is, we must discharge assumptions in the reverse order of when we make them. This nesting of scopes is what lets us use indentation to mark out our applications of $\rightarrow$ introduction. And, it makes proof organization much clearer, since it helps us keep track of exactly which assumptions a lemma depends on; without scoping rules like this we'd have to manually trace back to see which assumptions we used in deriving each lemma.

Note that we can discharge several assumptions at once, if we are proving a lemma like $a \wedge b \rightarrow c$. The rule is always that we collect all of the assumptions we're discharging, connect them using $\wedge$, and make them into the left-hand side of the new implication. We still have to follow scoping, which means that we must use only the most recent not-yet-discharged assumptions; but we can choose how many of these to use.

The resulting lemma has its own scope: we can continue to use it until the end of the scope where we proved it. That is, suppose we have a proof with the following outline:

1. Assume something
2. lorem ipsum...
3. lorem ipsum...
4. Assume something else
5. lorem ipsum...
6. lorem ipsum...

7. Now we have a lemma
8. lorem ipsum...
9. lorem ipsum...
10. Now the enclosing scope ends
11. lorem ipsum...
12. lorem ipsum...

We can use our lemma from line 7 anywhere in the green part of the proof, lines 8–10: after the lemma has been proven, up until the enclosing scope ends.

Why? After the enclosing scope ends, its assumptions are no longer available. Since we might have used these assumptions in proving our lemma, the lemma is no longer safe to use. (In our example, the assumption on line 1 goes out of scope after line 10.) If we didn't actually use these assumptions, we can always reorganize: we can pull the proof of our lemma out by one scoping level, so that the result remains available longer.

> *For convenience, we'll also allow "global" assumptions: assumptions we make throughout the proof. These are implicitly or explicitly assumed at the beginning and discharged at the end of our proof. Global assumptions are only a syntactic convenience, and don't change the rules of scoping.*

This description of scopes and nesting might remind you of function definitions in a programming language: just as a function definition begins with a header that introduces arguments, a lemma definition begins with a header that introduces assumptions. And just as we can use the function's arguments throughout the function definition block, up until we close the block by using `return`, we can use the lemma's assumptions throughout the lemma definition block, up until we close the block by using $\rightarrow$ introduction.

This similarity is not an accident: lemma definition in proofs plays a role that's completely analogous to function definition in programming languages. In both cases, one of the main purposes is *encapsulation*: we package up a part of a program or a proof so that we can use it repeatedly in the future.

> *In fact, just as many programming languages organize related function definitions into modules and namespaces, we can optionally organize related lemma definitions the same way. We won't explore the idea of modules or namespaces in proofs any further here, but this sort of organization can become very important when managing larger proofs.*

Example

In complicated proofs, we might need several lemmas, possibly even nested inside one another. Here's a slightly larger example that illustrates this idea.

In this example we will use three propositions: PB, J, and Sandwich. Their intended interpretations are "we have peanut butter," "we have jelly," and "we have a sandwich." The proof connects two different ways of making a sandwich.

1. Assume $PB \wedge J \to \text{Sandwich}$.
2. Assume PB.
3. Assume J.
4. Conclude $PB \wedge J$ by $\wedge$-intro from 2, 3.
5. Conclude Sandwich by $\to$-elim from 1, 4.
6. Conclude $J \to \text{Sandwich}$ from 3-5 by $\to$-intro.
7. Conclude $PB \to (J \to \text{Sandwich})$ from 2-6 by $\to$-intro.
8. Conclude $[PB \wedge J \to \text{Sandwich}] \to [PB \to (J \to \text{Sandwich})]$ from 1-7 by $\to$-intro.

In English, we can read the last line as: "Suppose that peanut butter and jelly together are enough to make a sandwich. Then if you give me peanut butter, it will mean that jelly is enough to make a sandwich."

There are three nested scopes in this proof: the outermost lemma starts at line 1, the next lemma starts at line 2, and the innermost lemma starts at line 3. We close these off in reverse order: the innermost scope concludes at line 6, then the middle scope at line 7, and the outermost at line 8.

If we look at the innermost scope, its job is to prove $J \to \text{Sandwich}$. In doing so, it gets to use assumptions from enclosing scopes. In particular it uses PB, which was assumed in the middle scope (at line 2), to conclude $PB \wedge J$; and it uses $PB \wedge J \to \text{Sandwich}$, which was assumed in the outermost scope (at line 1), to get Sandwich.

The middle and outer scopes are much simpler than the inner scope: in each of them, we just make an assumption that we'll need later (lines 1 and 2 for the outer and middle scopes respectively). Note that the innermost scope uses these assumptions but does not discharge them. So, these assumptions don't appear in the left-hand side of the innermost scope's lemma. And, these assumptions remain active, in case we wanted to use them for something else in the middle or outer scopes.

The kind of result that we've just proven holds more generally: whenever we have an AND on the

*left-hand side of an implication, like $a \wedge b \wedge c \wedge \ldots \to z$, we can turn it into a nested sequence of implications, like $a \to b \to c \to \ldots \to z$. This transformation is called **currying**, after Haskell Curry (a famous logician and programming language theorist who introduced it). For this reason, we'll call the above proof the **curry sandwich**.*

Our example illustrates the rules of scoping, which we recap here:

- Assumptions made outside a lemma, in an enclosing scope, are available to use inside the lemma.
- But these assumptions do not get collected in the lemma's result; they'll get collected later on, when we end the enclosing scope.
- Assumptions made at the beginning of a lemma are no longer available once we end the lemma. Instead, only the result of the lemma is available.
- So, when we collect the assumptions for each lemma, we skip any assumptions from sub-lemmas: these assumptions have already been discharged, and are no longer available in the current scope.
- We can keep using the lemma until the enclosing scope ends.

The above proof illustrates a good rule of thumb: always try breaking down a longer proof by focusing on the outermost connective first. We are likely going to need to finish our proof by using the introduction rule for this connective. So, we can figure out what the premises have to be for this rule, and work backward by trying to prove these premises.

In our curry sandwich example, the outermost connective is $\to$, and in fact the last step of our proof (line 8) is to use $\to$ introduction to produce this connective. For this use of $\to$ introduction, we have to assume the LHS of the implication (which we do at line 1), and prove the RHS (which we do at line 7). We've now fixed the first line and the last two lines of our proof, and we can proceed to the subgoal of proving $PB \to (J \to \mathrm{Sandwich})$. (To do so, we can use our heuristic again: the outermost connective is $\to$ again, so we again want to use $\to$ introduction. This time we have to assume PB (line 2) and prove $J \to \mathrm{Sandwich}$ (line 6).)

Negation

The easiest way to handle negation is to define $\neg\phi$ as a shorthand for $\phi \to F$. That is, $\neg\phi$ is true precisely when assuming ϕ would lead to a contradiction. With this definition, to handle $\neg$, we turn $\neg$ into $\to$ and F, which we already have rules for.

*If we stop here, we get a system called **intuitionistic** or **constructivist** propositional logic. In this system, all proofs have to be constructive: e.g., there's no notion of proof by contradiction. Counterintuitively, though, unlike classical logic, a formula of intuitionistic can be neither true nor false. Intuitionistic logic is a fascinating system, since its semantics nicely mirror the semantics of many programming languages; but that's a topic for a different course.*

To get a complete reasoning system for classical propositional logic, we need one additional rule:

- Double-negation elimination: for any expression ϕ, from $\neg\neg\phi$ we can conclude ϕ.

There are actually several equivalent rules we could add instead: we could pick

- Law of the excluded middle (also called *tertium non datur*, "there is no third choice"): a formula has to be either true or false. For any ϕ, we can conclude $\phi \vee \neg\phi$.
- Contraposition: $\phi \to \psi$ is equivalent to $\neg\psi \to \neg\phi$.
- Pearce's law: for any ϕ and ψ, it holds that $((\phi \to \psi) \to \phi) \to \phi$.
- De Morgan's laws: $\neg(\phi \vee \psi)$ is equivalent to $\neg\phi \vee \neg\psi$, and $\neg(\phi \wedge \psi)$ is equivalent to $\neg\phi \wedge \neg\psi$.

All of these choices are equivalent: from any one of them (plus the other inference rules we introduced above), we can prove all of the others.

To be precise, we can't actually prove an inference rule. We can only prove statements of propositional logic, and inference rules are not statements but tools for working with statements. Instead, we can make a recipe: whenever we were about to use (say) De Morgan's laws, here are the steps we'd take to get the same effect using (say) only double-negation elimination.

Exercise (difficult): show that double-negation elimination, the law of the excluded middle, and Pearce's law are equivalent. You can do this by putting them in a cycle and proving each one from the previous one.

Resolution

There's one more inference rule that deserves mentioning: the rule of *resolution*. This rule is not as well known as some of the others, but it forms the basis of many automated reasoning systems.

Suppose we have two statements of the form

$$\phi \vee \psi \qquad \neg\phi \vee \chi$$

Then resolution lets us conclude

$$\psi \vee \chi$$

That is, we delete ϕ and $\neg\phi$ from our premises, and join what's left with $\vee$.

As usual, ϕ, ψ, χ stand for any logical expressions. Note that the formula ϕ appears in both premises: negated in one and not negated in the other.

In resolution, it's common for ϕ to be a *literal*, i.e., a single proposition or its negation. We call the two instances of ϕ the *negative* and *positive* literals (meaning with and without the $\neg$ sign.) It's also common for ψ and χ to be *clauses*, i.e., disjunctions of several literals.

In this case, the premises of resolution are two clauses. We can apply resolution whenever a positive literal in one clause matches a negative literal in another. This matching literal is called the *resolvent*.

For example, from

$$a \vee b \vee \neg c \vee d \vee \neg e \qquad b \vee \neg d \vee f$$

we can use resolution on d to conclude

$$a \vee b \vee \neg c \vee \neg e \vee f$$

The literals in the conclusion are the union of the literals in the premises, minus d and $\neg d$. Note that we've applied one additional (commonly needed) simplification: we've used idempotence to remove the duplicated literal b, which would otherwise appear twice in the conclusion.

We can treat resolution as a derived inference rule: it follows from the other inference rules that we've already introduced. (For example, we can derive it using case analysis and the law of the excluded middle.) But resolution is particularly interesting since it can greatly simplify our inference system: we can replace the bulk of any proof with a proof that only uses resolution, so that we don't need general-purpose implementations of any other inference rules. For this reason, resolution forms the basis of many automated reasoning systems.

> *One rule to rule them all.*

More precisely, if we start from any set of assumptions in classical propositional logic, we can mechanically transform them into a conjunction of clauses. The clauses in this conjunction form our initial set of known facts.

If we want to conclude a formula ϕ from our assumptions, we look for a proof by contradiction: we assume $\neg\phi$ and try to derive F. So, we translate $\neg\phi$ into a conjunction of clauses, and add these clauses to our set.

Now, if ϕ follows from our assumptions, we are guaranteed to be able to derive F by successive applications of resolution: in each step we pick two clauses from our set that contain matching positive and negative literals, apply resolution, and add the resulting new clause back to our set. If the new clause is empty (a disjunction of zero literals), it is equivalent to F, and our sequence of resolutions constitutes a proof by contradiction that ϕ must hold.

Further reading: the Stanford Encyclopedia of Philosophy has entries on a huge variety of formal logic systems. A good place to start is the entry on classical logic.