Things to take with you

80-310/610 Formal Logic, Fall 2020

Pretty soon, your days as a student of 80-310/610 Formal Logic will be behind you. We hope you got a taste of the exciting field of mathematical logic, and enjoyed doing so! As you embark on your post-310/610 life, I wanted to type up a quick summary of some of the key takeaways from the course. Even if you aren't planning on studying any more formal logic, I think you'll find that some of these ideas can pop up in unexpected places. So may this serve as a small reminder of *how to think like a logician* – I hope it comes in handy.

1 Formal Syntax

Before we could even get started with the mathematical apparatus of formal logic, we had to be clear and explicit on one thing: what language we were speaking. Indeed, the manner in which we express our reasoning is *the very thing we're studying when we do formal logic*. Trying to do formal logic without explicitly laying out your language is like trying to do group theory without defining groups: you're missing the object of your inquiry. This is a general trend in formal logic: since we're trying to formally study *how* other inquiries (like the various branches of math) take place (and what their theoretical limits are), we have to precisely and explicitly define things which you would usually take for granted or leave implicit (e.g. what constitutes a "proof").

So logicians must take care to specify the formal language they're studying. As I mention more in the next section, this is usually done recursively (often by a formal grammar). But once we have our language (call it $\mathcal{L}$) defined, there's an important thing to realize: *the elements of $\mathcal{L}$ are – at this point – completely meaningless*. Sure, we'll take a formula, say $\varphi \wedge \psi$, and heavily *imply* a particular meaning by pronouncing it "φ and ψ". But they're just symbols on a page; they don't mean anything until we explicitly give then meaning.

It's also worth noting that our language $\mathcal{L}$ – even before we give it semantics or a deductive calculus – has a lot of its own structure. We can define (again by recursion) functions articulating various properties of formulas (e.g. their rank), we can discuss relationships among formulas (such as the "subformula" relation), and so on. With richer languages (like that of predicate logic), there are even more features we can look into like variable binding & capture, quantifier rank, and substitution. This will be important for our study of semantics and deduction, for several reasons.

- We'll often need to make syntactic "side conditions" (e.g. "t is free for x in φ") to correctly define our semantics & deductive calculus, and in some cases make use of the syntactic structure for defining the semantics (see the notion of an **extended signature**).
- We'll be interested in whether (and how) these syntactic relationships are reflected in the semantics. In other words: what does the syntactic *form* of φ (and its syntactic relationships to other formulas) tell us about what φ *means* (and how that meaning relates to the meaning of other formulas)?
- For proving key meta-results about our logic (in particular the "**Henkin construction**" to prove the completeness of predicate logic – see Sect. 5), we'll cleverly deploy the

syntax *to study the semantics*. This is just one of many curious ways that logic *talks about itself*.

How does this apply outside of mathematical logic? Well, there are certainly many apparent problems which are in fact *linguistic confusions*. While it might not suit your purposes to go all the way down to defining what "and" means, don't underestimate the value of precise syntax. Maybe a problem is evading solution simply because you haven't articulated it carefully enough. Maybe some disagreements are not disagreements at all, but two people saying essentially the same thing in different languages. Choosing the right syntax can be critical.

2 Recursion and Structural Induction

One technique we made use of this semester which shows up all over the place is *definition by* ***grammar***. For instance, we defined our propositional language using the grammar

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \mid \varphi \to \psi \qquad (p \in \mathsf{PROP})$$

or some variant of that. This is a concise way to express a recursive definition (in this case, the definition of "well-formed formula"): to write out in words what this grammar says, we would need a lot of tedious verbage ("if φ is a formula, so is $\neg\varphi$; if φ and ψ are formulas ..."). Among many other advantages, this allows for easy discussion of the difference between languages: if I write out a slightly-different grammar

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \varphi \vee \psi \qquad (p \in \mathsf{PROP})$$

then it's immediate to see the difference – no need to dig through paragraphs of "if φ is a formula...".

Because of their convenience, grammars are used in many branches of logic, math, and computer science. For instance, we can define languages with modal operators

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \Box\varphi \mid \Diamond\varphi \qquad (p \in \mathsf{PROP})$$

or we can define arithmetical expressions

$$e ::= \overline{n} \mid e_1 + e_2 \mid e_1 \cdot e_2 \mid e_1 - e_2 \mid \ldots \qquad (n \in \mathbb{Z})$$

or regular expressions

$$r ::= c \mid 0 \mid 1 \mid r_1 + r_2 \mid r_1 r_2 \mid r^*$$

or the syntax of a programming language

$$\begin{aligned} \tau &::= 1 \mid \tau_1 \times \tau_2 \mid \tau_1 \to \tau_2 \mid \ldots \\ e &::= \star \mid (e_1, e_2) \mid \lambda x : \tau . e \mid e_1(e_2) \mid \ldots \end{aligned}$$

or more sophisticated program logics

$$\begin{aligned} \pi &::= \pi_0 \mid \varphi? \mid \pi_1; \pi_2 \mid \pi_1 \cup \pi_2 \mid \pi^* \\ \varphi &::= p \mid \neg\varphi \mid \varphi \wedge \psi \mid \langle\pi\rangle\varphi \end{aligned}$$

all using grammars! Note that the last example contains two clauses which are *mutually recursive*, i.e. the φs are defined recursively in terms of the πs and the πs are defined recursively in terms of the φs. This takes a bit of care to make formal sense of, but is allowed!

The reason we want to define our language recursively (and therefore use a grammar) is so that our syntax has a **recursive** structure. If we define our language in this way, then each formula φ is built up out of the connectives in a unique way, i.e. it has a unique parse tree. This allows us to make definitions by recursion, e.g.

$$\begin{aligned}
\mathsf{rank}(p) &= 0 \\
\mathsf{rank}(\neg\varphi) &= 1 + \mathsf{rank}(\varphi) \\
\mathsf{rank}(\varphi \wedge \psi) &= 1 + \max(\mathsf{rank}(\varphi), \mathsf{rank}(\psi)) \\
\mathsf{rank}(\varphi \vee \psi) &= 1 + \max(\mathsf{rank}(\varphi), \mathsf{rank}(\psi)) \\
\mathsf{rank}(\varphi \to \psi) &= 1 + \max(\mathsf{rank}(\varphi), \mathsf{rank}(\psi)).
\end{aligned}$$

By supplying these clauses, rank is automatically a well-defined function taking formulas to natural numbers: there cannot be any ambiguity about how to calculate the rank of a given formula, because these clauses exhaust the possibilities for what the topmost connective in φ's parse tree could be. So we have a convenient way of defining properties (like the rank) of our syntax. Also note how the rank of φ is defined in terms of the ranks of its subformulas. This is the heart of recursion: make the "base case" of your definition, and then propagate the definition up using these recursive clauses; each clause is simple, but they combine to define a function defined on all formulas.

In the same way that recursion makes definitions expedient, **structural induction** makes proofs expedient. Indeed, induction is essentially a recursive definition of a proof: to prove that $P(\varphi)$ for all φ, I start with my base case and supply a proof of $P(p)$ for each p. Then I give "recursive clauses" (called **inductive steps**) which say how to obtain a proof of $P(\varphi)$ using the proofs $P(\psi)$ for subformulas ψ of φ. We usually phrase this as *assuming* $P(\psi)$ – the "**inductive hypothesis**" – and then obtaining $P(\varphi)$ from that assumption. But it supplies a method for constructing proofs: if a skeptic didn't believe me that $P((p \wedge q) \to r)$, then they could use my inductive proof to come up with a proof of that specific case:

- by the base case proof, $P(p)$, $P(q)$, and $P(r)$ all hold;
- by the $\wedge$ inductive step and the fact that $P(p)$ and $P(q)$ hold, derive that $P(p \wedge q)$ holds;
- by the $\to$ inductive step and the fact that $P(r)$ and $P(p \wedge q)$ holds, $P((p \wedge q) \to r)$ holds.

In this class, we mainly inducted on the natural numbers and formulas. But these are just two instances of the broader power of structural induction. Whenever you have a recursive definition (e.g. one given by a grammar), then there is a corresponding notion of recursion and an induction principle. For instance, the natural numbers are given by:

$$n ::= 0 \mid n + 1$$

and accordingly their recursion principle (to define a function $f : \mathbb{N} \to X$) asks for $f(0)$ and $f(n + 1)$ in terms of $f(n)$, whereas the induction principle (to prove $P(n)$ for all $n \in \mathbb{N}$)

asks for a proof of $P(0)$ and a proof of $P(n+1)$, assuming $P(n)$. Notice how this exactly matches the grammar. So if your grammar was instead

$$w ::= a \mid b \mid k(w, z) \mid h(w_1, w_2, w_3) \qquad (z \in \mathbb{Z})$$

then to define a recursive function f, you would need to supply

- $f(a)$
- $f(b)$
- $f(k(w, z))$ for each $z \in \mathbb{Z}$, where you're allowed to refer to $f(w)$
- $f(h(w_1, w_2, w_3))$, where you're allowed to refer to $f(w_1)$, $f(w_2)$, and $f(w_3)$

and a structural induction proof would go like:

- Prove $P(a)$
- Prove $P(b)$
- Assume $P(w)$, then show for an arbitrary $z \in \mathbb{Z}$ that $P(k(w, z))$
- Assume $P(w_1)$, $P(w_2)$, and $P(w_3)$, then show $P(h(w_1, w_2, w_3))$.

The possibilities here are endless: what matters is that you design the right syntax for whatever you're trying to do.

3 The Object Language/Metalanguage Distinction and Semantics

As mentioned above, the syntax of propositional (or predicate) logic is, on its own, meaningless. In order to endow these formal symbols with meaning, we need to supply **semantics**. This is where the intention of the name "mathematical logic" starts to come into focus: the semantics we'll give for our formal symbols will specify a precise sense in which the symbols are *describing a mathematical structure*. In the propositional case, this structure was rather simple: a function assigning a truth value to each atomic proposition. In the case of first-order logic, it was more complex: the "structures" we were dealing with were sets with designated elements (to interpret the constants), relations on the set (to interpret the relation symbols) and functions on the set (to interpret the function symbols). If you study more logic, you'll encounter more examples of mathematical structures (e.g. relational structures, algebraic structures, metric spaces, topologies, measure spaces, categories) which can interpret formal languages. But the premise is the same: we attach our formal symbols to mathematical structures, and try to understand how the syntactic structure of the symbols corresponds to the mathematical properties of the structure interpreting that syntax.

We do this, of course, by recursion: $[\![\varphi \wedge \psi]\!]$ is defined in terms of $[\![\varphi]\!]$ and $[\![\psi]\!]$. Indeed, our semantics definition will consist of clauses like

$$[\![\varphi \wedge \psi]\!] = \mathsf{true} \quad \iff \quad [\![\varphi]\!] = \mathsf{true} \text{ and } [\![\psi]\!] = \mathsf{true}.$$

Now, it kinda looks like this clause says nothing at all: we're defining $\wedge$ using "and", but $\wedge$ is supposed to mean "and". So I haven't really defined what "and-ness" is, I've just moved

the goalposts to define one notion of "and" ($\wedge$) in terms of another ("and"). I'm just moving words around and not saying anything. But actually there's something interesting going on here: I am effectively operating *two* layers of meaning. This is called the **object language/metalanguage distinction**, and it sits at the heart of mathematical logic.

The name "mathematical logic" carries an intentional ambiguity. On the one hand, it can mean (as suggested earlier in this section) that we are doing logic by mathematical means, i.e. we are using the tools and objects of mathematics to study logic. This is certainly a defining characteristic of mathematical logic. But the name can be read in another way: mathematical logic is the logic of mathematics. That is, we are seeking to investigate the manner of reasoning utilized in mathematics. Now, there's an obvious problem here: if I need mathematical tools to do mathematical logic, but also don't want to presuppose any math (because that's what I want to construct & study), how can I even get started? The recourse is **metatheory**: I take for granted some basic, informal[1] principles about how to reason, including the commonly-understood meaning of the word "and", and then use that to construct the mechanics of my formal logical system.

So the stuff we take for granted is encapsulated in the **metalanguage**. The right-hand side of my semantic clause asks for $[\![\varphi]\!] = \mathsf{true}$ and $[\![\psi]\!] = \mathsf{true}$: this is the case when both $[\![\varphi]\!] = \mathsf{true}$ and $[\![\psi]\!] = \mathsf{true}$ hold, and does not hold when either does not hold. This is my "meta" version of *and*. But now I'm using that to define the meaning of my formal symbol $\wedge$ *within the logical system I'm setting up*. This logical system is the thing I'm ultimately studying (the *object* of my inquiry), and the language within that system (which includes $\wedge$ but not "and") is called the **object language**. The object language serves as a formally-defined copy of the reasoning that's possible in the metalanguage. Maybe this analogy will help: if I take a picture of you standing next to a tree, I can figure out *just by looking at the picture* that the tree is, say, five times taller than you. I'm reasoning about the real you and the real tree, but I'm doing that reasoning inside a controlled setting with an artificial stand-in for both you and the tree. In mathematical logic, our goal is much more ambitious than comparing the heights of people and trees: it's capturing the essential structure of reason itself.

So this is what our semantics achieved: they took reasoning from the metalanguage, like "and", and then encoded them in the formally-defined structure of the object language. Going back to the analogy from the previous paragraph, if my picture of you next to the tree was accurate enough (e.g. there wasn't a distorted perspective), then I was able to make conclusions about the real world based off my analysis of a picture. Likewise, if my semantics are good, then I can perform reasoning in the object language (see Sect. 4 below), obtain conclusions, and then view these conclusions as applicable in the real world. So my formal system is not just symbol-pushing, I've actually managed to **formalize** reasoning from the metalanguage in a precise way. For instance, the formal deduction $\{\forall x(P(x) \to Q(x)), P(s)\} \vdash Q(s)$ formalizes the classical syllogistic argument, "All men are mortal, Socrates is a man, therefore Socrates is mortal". This argument seemed obvious intuitively (i.e. using our metatheoretic notions of what "all" means), but now we have an airtight logical rendition of this argument.

This, by the way, was why I was a constant stickler about what seemed like minor notational differences. The symbol $\to$ was specified as a connective in our *object language*, and was the formal stand-in for the metatheoretic notion of implication, which we indicated

[1] *Informal* in the sense of not being formalized in mathematical logic. These are still very rigorous.

with either $\implies$ or "if…then…". Others use different conventions (e.g. some will instead use $\implies$ as part of the object language[2]), but this was our convention. And so, if you wrote

$$\text{it was raining} \to \text{it was not sunny}$$

this inappropriately mixed the object and metalanguage: "it is raining" is a statement in our metalanguage (English), whereas $\to$ is supposed to connect two formulas in the object language. I similarly objected to using the symbol $\wedge$ in the middle of a sentence when you meant "and", or $\neg$ when you meant "not".[3] Nitpicky? Perhaps. But it was in service of a profound idea, so I stand by my nitpicking. An inherent part of mathematical logic is that the logic *talks about itself*, and one must be very careful to not get confused – the object language/metalanguage distinction helps sort that out.

4 Explicit Deductions and Soundness & Completeness

When learning to work and think like a mathematician, the main thing one must learn is how to be *rigorous*. Most mathematicians can probably agree on some basic standards of what being "rigorous" means: only working with clearly-defined objects, avoiding ambiguity, not making statements without proving them (or providing a reference to a proof), and so on. When working in the metalanguage (as mathematicians generally are), this is the kind of standard you must hold yourself to. But, ultimately, whether a proof is sufficiently "rigorous" enough depends on the mathematical conventions of the author & their audience, and there doesn't exist an explicit standard of whether a proof is rigorous or not. Reading a mathematical proof and deciding if it's correct requires a trained reader, and there can be room for disagreement about whether a proof succeeds at establishing its intended result. Proofs conducted in the metalanguage are inherently *informal*.

But we're doing *formal* logic. We put a great deal of effort above into establishing exactly what counts as a statement in our object language. If we wish to formalize what counts as a *proof*, we'll need a much better standard than "I know it when I see it". To meet the level of formal detail we want, there needs to be a completely objective way to look at a purported object-language proof and determine if it actually counts as a proof. To do this, we will need to explicitly and exhaustively account for all the "moves" allowed in a formal proof. This is what's achieved with the notion of **deduction**: it gives an exhaustive specification of what you're allowed to do to construct a proof in the object language. So if I claim to have deduced some formula φ and supply a supposed proof, you can check each step to see if it's legal. There is no room for disagreement: it either *is* or *isn't* a valid deduction, and we can figure it out.

Of all the problems you solved on your problem sets, the easiest for me to grade were the formal deductions: I could just go through each step and see if it was legal according to the explicit rules given by the definition of "deduction". I didn't need to have a "sense" of mathematical rigor, honed by years of reading and writing mathematical proofs. I didn't really even need to understand *what* you were proving, how the proof worked, what meaning you intended for the symbols, etc.: I just needed to know how to follow basic syntactic rules. In fact, a computer could do it: I could program up a computer to step through each line of your deduction and check that at each step you either (a) wrote down an instance of one of

[2] Or, for reasons which still confound me, $\supset$.

[3] $\forall$ students $\in$ formal logic, (they unnecessarily use formal symbols $\in$ a sentence) $\to \neg$(Jacob will give them a good grade)

our axiom schemes, or (b) applied one of our rules of deduction. We didn't mention this too much in the course, but this is a requirement for any axiom system in (usual) formal logic: your axiom schemes and inference rules must be laid out so explicitly that a computer can be programmed to recognize legitimate applications of them and reject illegitimate applications.

As an interesting side tangent: the connection between deduction and computation goes much deeper. Indeed, the task of explicitly outlining what formal deductions are *is the problem which computer science was originally invented to solve*. I spoke in the previous paragraph about all our deduction rules being "explicit" and being "basic syntactic rules". But how do I know if my set of axioms & inference rules is "formal enough"?[4] The proper answer to this is what I mentioned in the previous paragraph: that a computer be able to recognize whether the axioms and inference rules are being legally applied. It was in this context that "computation" was explicitly defined for the first time.[5]

So what we've done with the notion of "deduction" is define explicitly how reasoning is conducted in the object language. The object language exists to formalize metalanguage statements about mathematical objects, and deduction seeks to formalize metalanguage proofs. So how did we do? If we did our formalization project well, then the explicit, sanitized deductive reasoning we can do in the object language ought to capture the full range of what can be proved in the metalanguage. If I take my favorite mathematical theory, devise a formal object language to express the features I'm concerned with, and start deducing statements, I want two things to be true:

- Any formula which I can deduce, if read as a statement about the mathematical structure, should turn out to be true
- If there's a statement which I can informally prove to be the case, then there ought to be a formal deduction of the corresponding object-language formula.

These are both properties about the interplay between my informal and formal reasoning, and together they say that my formal apparatus "adequately captures" my informal reasoning. If both are true, then this shows the adequacy of my formal system: it's powerful enough that I can formally deduce *any* true statement, and is also correct (i.e. I can't deduce things that are false). If I have a formal system which satisfies these two properties, then it will turn out to be tremendously useful. For instance, if a colleague and I disagree about the correctness of an informal proof, then to settle our dispute we could try to formalize it. If the informal proof *is* correct, then, by the second property, there must exist a formal deduction of that fact. If the informal proof is incorrect and the claim it purports to prove is actually false, then, by the first property, any attempt to formalize it will fail (and I should – in theory – be able to deduce its negation). This gives us an objective mechanism to turn to when the inherent subjectivity of informal proof becomes an issue.

[4]Consider this: instead of taking (L1)-(L5), (E1)-(E3), Generalization, and MP as my deductive system for first-order logic, why couldn't I just say, "every tautology of first-order logic is an axiom of my system, and there are no inference rules". This defines a deductive system, and it turns out to be sound & complete: every tautology is deducible (in one line), and only tautologies are deducible. So why doesn't that count as an axiomatization of first-order logic?

[5]Arguably *the* foundational work in theoretical computer science is Alan Turing's 1936 paper *On Computable Numbers with an Application to the Entscheidungsproblem*. If you read more about what the *entscheidungsproblem* is, you'll find that it's about the ability of computers to do mathematical proofs, specifically whether there exists a computer algorithm which can take an arbitrary first-order sentence and decide whether it's a tautology or not. There isn't such an algorithm, but in order to prove that, Turing first had to define what an "algorithm" even is. The notion he came up with is now known as a *Turing machine*. Subsequently, logicians (particularly Gödel) recognized that "verifiable by a terminating Turing machine" is the right condition to require of our deductions. Gödel's Incompleteness Theorems – though they originally predate Turing's (and Church's) work on computability – critically rely on this condition (and most modern statements of the Incompleteness Theorems involve lots of computability theory/recursion theory).

The properties in the preceding paragraph are, of course, the soundness and completeness results of formal logic. If I have a mathematical theory I'm interested in (for instance, partially ordered sets) and have devised a formal language for that theory (e.g. a first-order signature with a binary relation $\leq$), and written down a set Γ of axioms (e.g. $\forall x.x \leq x$) which the **models** of the theory must satisfy, then the soundness and completeness theorems assure me that:

- (Soundness) If I can deduce a sentence φ according to the rules of my deductive system (where I'm allowed to write down any element of Γ as an assumption), then for any of the models of my theory (any structure validating all of Γ, in this case any poset), φ must hold of that model too,
- (Completeness) If there's a formula φ which holds on all models of Γ, then there must be a way of deducing φ from Γ.

And so we see that the formal deductive calculus succeeds at capturing the structural properties of the models, or at least the properties that can be formalized in the object language.

The Soundness and Completeness results also prove to be powerful tools for understanding further how the model theory and proof theory interact. In light of the statement of *strong* soundness and completeness (which is the statement mentioned in the previous paragraph), we can focus our attention on sets Γ of object-language sentences, which we might call **theories**. There are several closely-related notions here:

- *consistency*: Γ is **consistent** if, in the deductive calculus, it is impossible to prove a contradiction from Γ
- *satisfiability*: Γ is **satisfiable** if there is a structure (interpreting the appropriate signature) which validates all the formulas in Γ
- *finite satisfiability*: every finite subset of Γ is satisfiable.

Soundness and Completeness gives us that satisfiability and consistency are equivalent. Indeed, the standard way to prove a theory is consistent (which is difficult to prove directly, since it's hard to argue against all possible deductions) is to instead exhibit a model for it. The question of whether certain first-order theories are consistent or not goes on to play a central role in the development of modern logic.

But furthermore, as a corollary of completeness we can prove **compactness**: that finite satisfiability and satisfiability are also equivalent. As we saw, this fact can be exploited to deliver some of formal logic's most startling results: showing that each finite subset of Γ is satisfiable is often much easier than proving Γ itself satisfiable, and we can use that to construct curious **nonstandard** models of familiar structures. The mere existence of these structures (and the fact that they can be designed to satisfy our favorite standard mathematical theories) raises a ton of interesting questions about the nature of mathematical practice and its possibilities & limitations, which you're well-encouraged to consider.

5 Maximally-Consistent Sets and The Henkin Construction

Finally, let me remark on the *manner* which we used to prove the soundness and completeness results. Soundness was proved, of course, by structural induction. Here, it was

the *deductions* which were the inductively-structured object: each deduction is either one line, or consists of a shorter deduction extended by one line (according to strict rules, as mentioned above). In particular, there were exactly four ways (three, in the propositional case) to extend a deduction: write down another instance of an axiom scheme, write down one of the assumptions in your theory Γ, apply modus ponens, or (in the case of predicate logic) apply generalization. All deductions are produced by repeatedly doing these, and so to prove something about *all* deductions we inducted on this structure, showing that it preserved validity.

But it was completeness which really demanded us to work hard. In the propositional case, we were able to prove completeness using **maximally-consistent sets**. We built up a whole theory of maximally-consistent sets, including the key facts (a) that they're closed under deduction and (b) that they contain every formula or its negation. To prove the contrapositive of completeness (that any formula φ such that $\Gamma \nvdash \varphi$, there's a valuation satisfying Γ but refuting φ), we had to pass from the syntactic/deductive side of things to the semantic/model theoretic side, taking care to refute φ all along the way. If φ was not a theorem of Γ (i.e. $\Gamma \nvdash \varphi$), this implied $\Gamma \cup \{\neg\varphi\}$ consistent by the meaning of "consistent". We then had our two key lemmas: Lindenbaum's Lemma[6] to extend $\Gamma \cup \{\neg\varphi\}$ to a maximally-consistent Γ', and then the Truth Lemma to turn Γ' into a valuation $v_{\Gamma'}$ such that

$$[\![\psi]\!]_{v_{\Gamma'}} = \mathsf{true} \qquad \text{for all } \psi \in \Gamma'.$$

In particular, we had that $\neg\varphi \in \Gamma'$ and $\Gamma \subseteq \Gamma'$, so $v_{\Gamma'}$ was the Γ-validating-but-φ-refuting valuation which Completeness demands. Variations on this method of proof abound (in particular, for proving the completeness of logical systems), and the general spirit of specially-building semantic structures out of the syntax is ubiquitous in formal logic.

The most famous place where semantics are fashioned from syntax is in the **Henkin construction**, the standard modern approach for proving the Completeness of first-order logic (the one you learned).[7] There are several interesting moves in this proof, which I encourage your continued reflection on.

- Starting with a consistent set Γ over the signature $\mathcal{S}$, we need to expand both $\mathcal{S}$ and Γ to "declare" a bunch of new terms. This needed to be done because Γ might contain a bunch of existentially-quantified formulas which we need to be satisfied. A lot of effort had to go into showing that we could extend $\mathcal{S}$ and Γ to $\mathcal{S}'$ and Γ_∞ in an appropriate way. Note that we're nominally making our life harder here: satisfying Γ_∞ should be at least as hard as satisfying Γ, since $\Gamma \subseteq \Gamma_\infty$. But actually it makes it easier: the stuff we add will actually make it possible to satisfy them. Sometimes the best way to make your life easier is to make your life temporarily harder.
- Then we imported the logic of maximally consistent sets to extend Γ_∞ to Γ', again producing an arbitrary maximally-consistent set using the key insight of Lindenbaum's Lemma.
- Then we had to find a model satisfying Γ'. And here's where the syntax-into-semantics magic really happens: the model we'll construct will *be built out of terms in the language*. That is, the **term model** is really built out of syntax. In the same way that $v_{\Gamma'}$ was

[6] An important thing to realize about Lindenbaum's Lemma is that it produces an arbitrary result. We usually don't care exactly *which* maximally consistent extension of our set we get (and in general there can be many), we just need *one*.

[7] Gödel's 1929 Completeness proof is slightly different and more convoluted. Henkin came out with this proof a decade later.

specially-defined to exactly agree with Γ' on the truth of formulas, the term model will be designed to interpret terms as themselves. This connects back with our earlier work to throw in a new term to witness each existentially-quantified formula in Γ: those terms *become elements of the domain of discourse*, the exact kind of thing to witness existential statements.

- Then we needed to force the sentences of Γ' to be true in this model. In particular, we need to make it so that terms considered equal in the Γ'-theory (i.e. terms s and t such that $\Gamma' \vdash s = t$) actually are interpreted as the same element of the domain of discourse in the term model. So to do this, we make the domain of discourse of the term model *equivalence classes* of terms, where the equivalence relation is given by Γ'-deducible equality. Again, we're using the syntactic/deductive world to define the semantics in the way we want.
- And then finally, we could prove that terms in $\mathcal{S}'$ are actually interpreted as themselves (or rather, as their equivalence class) and prove the Truth Lemma: that the term model validates exactly those formulas deducible from Γ'.

This was all to prove the Model Existence Lemma, and then Completeness is a quick corollary. This proof had a large impact on the field of formal logic, and many logicians have utilized insights from this proof for various other means. As mentioned above, using the syntax to construct semantics, specifically semantics which interpret the syntactic elements as (some version of) themselves is a really clever approach. And it showcases the ability of logic to "fold in on itself" and "talk about itself". Logic is really unique in its ability to do this sort of thing, and this proof is one of many fine examples.

6 Conclusion

So, as I said at the top, I hope you take a lot of these insights with you, and figure out new and exciting ways to use them. But at the very least, I hope you felt the same awe at this beautiful topic, the same amusement at its clever moves, and same appreciation for its ambitious scope that I did as I learned it. Thanks for a great semester.