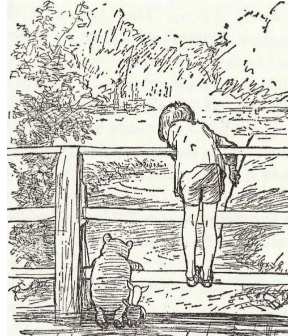
Informal Logic for Computer Science

By Jeff Edmonds

This chapter will cover the basics of the logic needed by any computer science student.

Lean over to watch the river slipping away beneath you,



you will suddenly know everything there is to be known.

1 The Logic of *True/False* and *And/Or/Not/Implies*

Logic: We start here with the basics of *Propositional Logic*, which covers *True/False* variables and sentences with *And/Or/Not/Implies* connectors.

Statements: Here α and β each denote statements that are either true or false, eg α could be "I love logic" or something more complex like "I love logic and I will do my homework."

True or False: In our logic, each variable and logical sentence/formula/statement α is either *true* or *false* or independent of what we know. Not knowing is caused by not knowing which universe/model/interpretation/assignment we are in. In each of these, α is either true or false. There is no gray. Hoping will not help. We won't consider different people with different beliefs. And we are never wrong. There are no probability distributions. And hence no "likely". There is no time. And hence no "now" or "later".

Connectors: Just to confuse you, logicians use the logical symbol $\wedge$ to represent *and*, $\vee$ to represent *or*, $\neg$ to represent *not*. and $\rightarrow$ to represent *implies*. Just like the addition table that states $4+3=7$, the table states that $\alpha \rightarrow \beta$ is true whenever α being true implies that β is also true. Equivalently, it is only false when α is true and β is false.

α	β	$\alpha \wedge \beta$	$\alpha \vee \beta$	$\neg\beta$	$\alpha \rightarrow \beta$
T	T	T	T	F	T
T	F	F	T	T	F
F	T	F	T		T
F	F	F	F		T

Axioms/Assumptions: Few things can be proved without making some kind of assumptions. An *axiom* is a sentence that we will assume is true. Γ denotes the set of axioms.

Tautology/Valid: The sentence $(\alpha \rightarrow \beta) \rightarrow \gamma$ iff $(\neg\alpha \rightarrow \gamma) \wedge (\beta \rightarrow \gamma)$ is said to be a *tautology* because under all 2^3 assignments $\langle \alpha, \beta, \gamma \rangle \in \{true/false\}^3$ the two sentences evaluate to the same T/F value.

Valid: A statement α is said to be *valid* if it is true in every possible model/assignment in which your axioms/assumptions Γ are true. This is equivalent to the sentence $\Gamma \rightarrow \alpha$ being a tautology.

Brute Force Proof: The most mindless proof makes a table listing all 2^n possible T/F assignments to the n variables checking for each that whenever the axioms Γ are true, so is α. The problem is that for large n this proof is too long. It also fails to give you any intuition.

A Hilbert Style Proof: A *Hilbert proof* is a sequence of sentences each of which is either an *axiom* $\in \Gamma$ or follows by some mechanical symbol-manipulation rule from previous sentences in the proof. We want by induction on the length of the proof that every sentence in the proof is valid. By definition the axioms themselves are valid. Hence it is sufficient that each rule is *sound*, namely if all the previous sentences that it depends on are valid, then so is the new sentence being added.

Modus Ponens: One rule your a proof system needs for sure is *Modus Ponens*, which states that if you have proved both α and $\alpha \rightarrow \beta$ are valid, then you can conclude that β is also valid. Here $\alpha \rightarrow \beta$ is read "α implies β". You also need a rule that groups a number of proved sentences $\alpha_1, \alpha_2, \ldots$, and α_r into a single sentence that is their *And*, namely $(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_r)$.

Rule vs Axiom: After these, it is a matter of taste whether you add something as a rule or as an axiom. The rule version would state that if you have proved sentences $\alpha_1, \alpha_2, \ldots$, and α_r are valid, then you can conclude that β is also valid. An equivalent axiom to add to Γ would be $(\alpha_1 \wedge \alpha_2 \wedge \ldots \wedge \alpha_r) \rightarrow \beta$.

Formal Proof System: A *proof system* describes the rules being allowed. Though one could prove everything with an extremely small list of rules, we will include every sound rule that we can think of that a student might feel is useful.

Table of Rules: Figure 1 summarizes some axioms, proof techniques, and lemmas having to do with *Propositional Logic*.

Proof Techniques/Lemmas

	Using:	From:	Conclude:	Proving:	From:	Conclude:
And $\wedge$: Both are true	Separating And	$\alpha \wedge \beta$	α & β	Eval/Build $\wedge$	α & β $\neg\beta$	$\alpha \wedge \beta$ $\neg(\beta \wedge \gamma)$
Or $\vee$: At least one is true	Selecting Or	$\alpha \vee \beta$ & $\neg\alpha$	β	Eval/Build $\vee$	α $\neg\alpha$ & $\neg\beta$	$\alpha \vee \gamma$ $\neg(\alpha \vee \beta)$
	Cases	$\alpha \vee \beta$, $\alpha \rightarrow \gamma$, & $\beta \rightarrow \gamma$	γ	Excluded Middle		$\alpha \vee \neg\alpha$ & $\neg(\alpha \wedge \neg\alpha)$
Implies $\rightarrow$: If LHS is true, then so is RHS.	Modus Ponens	α & $\alpha \rightarrow \beta$	β	Deduction	Assume α, prove β	$\alpha \rightarrow \beta$
	Cases	$\alpha \vee \beta$, $\alpha \rightarrow \gamma$, & $\beta \rightarrow \gamma$	γ	Eval/Build $\rightarrow$	β $\neg\alpha$ α & $\neg\beta$	$\gamma \rightarrow \beta$ $\alpha \rightarrow \gamma$ $\neg(\alpha \rightarrow \beta)$
	Equivalence	$\alpha \rightarrow \beta$ & $\beta \rightarrow \alpha$	α iff β	Contrapositive	$\alpha \rightarrow \beta$ iff $\neg\beta \rightarrow \neg\alpha$ iff $\neg\alpha \vee \beta$	
Negation $\neg$: Flips truth	Transitivity	$\alpha \rightarrow \beta$ & $\beta \rightarrow \gamma$	$\alpha \rightarrow \gamma$	De Morgan's Law	$\neg(\alpha \wedge \beta)$ iff $\neg\alpha \vee \neg\beta$	

Double Negation: $\neg\neg\alpha$ iff α By Contradiction: From $\neg\alpha \rightarrow \beta$, and $\neg\alpha \rightarrow \neg\beta$, conclude α.
Commutative: $\alpha \vee \beta$ iff $\beta \vee \alpha$ and $\alpha \wedge \beta$ iff $\beta \wedge \alpha$ Inconsistent: From β and $\neg\beta$, conclude anything.
Distributive: $\gamma \wedge (\alpha \vee \beta)$ iff $(\gamma \wedge \alpha) \vee (\gamma \wedge \beta)$ and $\gamma \vee (\alpha \wedge \beta)$ iff $(\gamma \vee \alpha) \wedge (\gamma \vee \beta)$

Figure 1: A summary of *rules/proof techniques/axioms* having to do with logical operators. On the left of the figure are those rules which use a $\wedge$, $\vee$, or $\rightarrow$ statement after it has already been proved. On the right are those which can be used to prove such a statement. *Selecting Or*, for example, states that if you already know that (α or β) is true and you know that α is not true, then you can conclude that β must be true. *Eval/Build* states that to prove that (α or γ) is true, it is sufficient to prove that α is true. We call it an evaluation rule because from only $\alpha = T$, you can *evaluate* $\alpha \vee \gamma = T$. We call it a *build* rule because from α in your list of statements that you know are true, you can add the larger statement $\alpha \vee \gamma$.

Sound Rule: A rule is said to be *sound* if when all the previous sentences that it depends on are valid, then so is the new sentence being added. As an example, we will give the brute force proof that the Selecting Or rule is sound, i.e. for each of the 2^n settings of the variables, either one of conditions is false so the rule does not apply or in fact the conclusion is true. For the Selecting Or rule, this amounts to ($\alpha \vee \beta$ and $\neg\alpha$) being false or β being true.

α	β	$\alpha\vee\beta$	$\neg\alpha$	$\neg((\alpha\vee\beta)$ and $\neg\alpha)$	β	$\neg(\alpha\vee\beta$ and $\neg\alpha)$ or β
T	T	T	F	T	T	T
T	F	T	F	T	F	T
F	T	T	T	F	T	T
F	F	F	T	T	F	T

Exercises:

1. For each rule in Figure 1, give an example, explain what it means, and informally argue that it is true. Also give an example in which $\alpha\rightarrow\beta$ is true and $\beta\rightarrow\alpha$ is not.

2. Suppose your proof system is given the *Selecting Or* axiom $((\alpha \vee \beta)\wedge\neg\alpha)\rightarrow\beta$, the *Eval/Build* $\wedge$ rule, and *Modus Ponens*. Use them to effectively apply the *Selecting Or* rule.

3. Prove the other rules in the figure using the *brute force* (table) proof.

4. It can be fun to assume that a subset of these rules are true and from them prove that others are true. Proof the *Selecting Or* statement $((\alpha \vee \beta)\wedge\neg\alpha)\rightarrow\beta$ using any of the rules except for the *Selecting Or* rule.

5. I just don't understand my dad. Maybe if I do well at school, he will love me. Maybe if I follow all his rules, he will love me. I know that at least one of these is true. I just don't know which. What do I need to do to guarantee his love? Let α denote "I do well at school," β denote "I follow all his rules," γ denote "He will love me," and X denote "The situation", i.e., $(\alpha\rightarrow\gamma)\vee(\beta\rightarrow\gamma)$.

   (a) Suppose I believed that I do not need to both do well at school and follow his rules so I only do one of them. Does this guarantee his love? Give a True/False value of all the variables and statements in which it does not work out for me.

   (b) To be careful, I both do well at school and follow his rules. Prove that this guarantees his love.

Answers:

1. Explanations of the rules:

   Separating And: If you know "I love logic and I will do my homework," then you know both that "I love logic" and that "I will do my homework." This is the meaning of the connector *and*. The logical symbol $\wedge$ denotes *and*. Hence, if the statement $\alpha\wedge\beta$ is true then by definition the statement α is true, as is the statement β.

   Eval/Build $\wedge$: Conversely, if both α and β are true, then their *and* is true. We call it an *evaluation* rule because from $\alpha = \beta = T$, you can evaluate $\alpha\wedge\beta = T$. We call it a *build* rule because if both α and β are in your list of statements that you know are true, you can add the larger statement $\alpha\wedge\beta$.

   A standard proof only lists things that it has proven to be true. Hence, you state that β is false by stating that $\neg\beta$ is true. Here $\neg$ denotes the logical negation symbol. The second Eval/Build rule, from $\neg\beta$ concludes that (the *and* of β with anything) is false, i.e., $\neg(\beta\wedge\gamma)$ is true. One fun thing about this is that we can evaluate $\neg(\beta\wedge\gamma) = T$ without knowing the value of γ.

   Selecting Or: If you know "I will do my homework or I love logic," then you know that at least one of these statements is true, perhaps both. This is the meaning of the connector *or*. From this, you do not know whether or not "I love logic." On the otherhand, if in addition you know that "I will not do my homework," then you because at least one is true, you can conclude that "I love logic." The logical symbol $\vee$ denotes *or*. The rule is that from the statements $\alpha\vee\beta$ and $\neg\alpha$, one can conclude β.

Eval/Build $\vee$: If you know that "I love logic," then you can conclude that "I love logic or I will do my homework," independent from knowing whether or not "I will do my homework." More generally, this rule states that to prove that either α or γ is true, it is sufficient to prove that α is true. We call it an evaluation rule because from only $\alpha = T$, you can *evaluate* $\alpha \vee \gamma = T$. We call it a *build* rule because from α in your list of statements that you know are true, you can add the statement $\alpha \vee \gamma$. On the other hand, in order to prove $\alpha \vee \beta$ is not true, then you need to prove that both α and β are not true.

Excluded Middle: The statements $\alpha \vee \neg\alpha$ and $\neg(\alpha \wedge \neg\alpha)$ are axioms of our logic, i.e., assumed to be true. They state that in our logic, each variable and logical sentence/formula/statement α is either true or false but not both.

De Morgan: If it is not the case that "I love logic" and "I will do my homework," then at least one of them must be false. More generally, $\neg(\alpha \wedge \beta)$ and $\neg\alpha \vee \neg\beta$ are equivalent statements because if it is not true that both α and β are true, then at least one of them must be false.

Modus Ponens: If you already know that "I love logic implies I will do my homework" and that "I love logic," then a classic logic conclution is that "I will do my homework." More generally, from α and $\alpha \rightarrow \beta$, you can conclude β.

Deduction: The standard way to prove $\alpha \rightarrow \beta$ is by temporarily assuming that α is true and using that belief to prove β. See more below.

No-Causality: The statement $\alpha \rightarrow \beta$ is widely missunderstood by beginners. Formally, it means "If is α true, then so is β." The statement, however, is not supposed to imply any sense of causality. A better interpritation might be, "In every universe in which α true, it happens to be the case that β is also true." Equivalently, one might say "There are no universes in which α true and β is mot." An example is "Is hound" $\rightarrow$ "Is dog." Note how the set hounds is a subset of the set of dogs. As such, the set of universes in which you are a hound is a subset of the set in which you are a dog. Note that modus ponens also follows from this understanding, because knowing that you are in a universe in which α is true still tells you that β is true in this universe as well.

Reverse Direction: Note that $\alpha \rightarrow \beta$ does not imply $\beta \rightarrow \alpha$. For example, "Is hound" $\rightarrow$ "Is dog" is true and the reverse "Is dog $\rightarrow$ "Is hound" is not.

Eval/Build $\rightarrow$: The statement (I am a billionaire) $\rightarrow$ $(1 + 1 = 2)$ is automatically true whether or not I am a billiionaire. More generallry, if β happens to be true, then the statement $\alpha \rightarrow \beta$ is automatically true because the contract is not broken. Similarly, the statement $(1 + 1 = 3) \rightarrow$ (I am a billionaire) is automatically true again whether or not I am a billionare. If α happens to be false, then again $\alpha \rightarrow \beta$ is automatically true. Finally, the statement $(1 + 1 = 2) \rightarrow$ (I am a billionaire) is false because the math is wrong and I am a not billionare. If α is true and β is false, then $\alpha \rightarrow \beta$ is false.

Contrapositive: If you know that "I love logic" implies "I will do my homework," then it follows that "I will not do my homework" implies "I do not love logic," because if "I do love logic" then "I would do my homework." More generally $\alpha \rightarrow \beta$ and $\neg\beta \rightarrow \neg\alpha$ are considered equivalent statement. They are the *contra positive* of each other. Other equivalent statements are $\neg(\alpha \wedge \neg\beta)$ and by De Morgan $\neg\alpha \vee \beta$. Recall, however, that $\beta \rightarrow \alpha$ is a different statement.

Equivalence: If you know that "I love logic" implies "I will do my homework" and you know that "I will do my homework" implies "I love logic," then one of these is true *if and only if* the other is true. These become *equivalent/interchangeable* statements.

Cases: A very common proof technique is *proof by cases.* Suppose, for example, you know that you will either do your homework today or tomorrow. You know that if you do it today, then you will pass the exam. You also know that if you do it tomorrow, then you will pass the exam. Either way, you know that you will pass the exam. In general, our goal is to prove γ. We prove that there are only two cases α and β, i.e., at least one of these is true. For the first case, we assume α and prove γ. In the second, we assume β and again prove γ. Either way we can conclude γ.

Trying a case and backtracking to try another case is called *branching* and is the basis of *Recursive Backtracking*. See Chapter **??**. A common thing to branch on is the value of a variable x that

seems to appear in a lot of places, namely $((x\!=\!T)\vee(x\!=\!F))\wedge((x\!=\!T)\rightarrow\gamma)\wedge((x\!=\!F)\rightarrow\gamma))\rightarrow\gamma$. For each such case, the algorithm simplifies the sentence and recurses.

One can build more elaborate decision tree of cases. For each path from the root to a leaf, the proof proves that for this case, the statement is valid. Having a tree ensures that all the cases are handled. See Tree of Options page **??** and in Exercise**??** (used to be III.13).

Simplifying: If a value of a variable/subsentence is substituted into a sentence, then it can be simplified as follows: $F\wedge\beta\equiv F$; $T\wedge\beta\equiv\beta$; $F\vee\beta\equiv\beta$; $T\vee\beta\equiv T$; $F\rightarrow\beta\equiv T$; $T\rightarrow\beta\equiv\beta$; $\alpha\rightarrow F\equiv\neg\alpha$; $\alpha\rightarrow T\equiv T$.

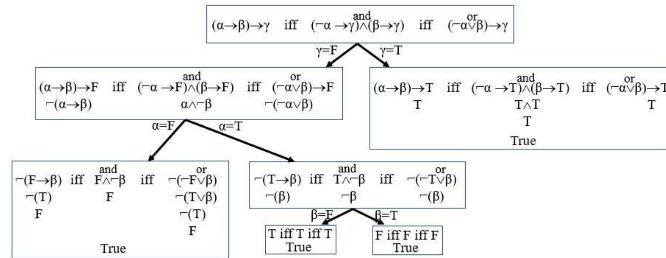


Figure 2: Proving logical equivalent statements using the Davis Putnam algorithm

Deduction: Let's revisit deduction. The standard way to prove $\alpha\rightarrow\beta$ is by temporarily assuming that α is true and using that belief to prove β. People struggle with understanding why we can assume α. The reason is that we really are doing proof by cases. In the first, case α is false, in which case $\alpha\rightarrow\beta$ is automatically true. We are now doing the second case, in which we assume α is true.

Transitivity: If you can travel/prove from α to β and from β to γ, then you can travel/prove directly from α to γ. Similarly, from $\alpha\rightarrow\beta$ and $\beta\rightarrow\gamma$, you can conclude $\alpha\rightarrow\gamma$.

Circuit: A *circuit* is like the *and/or/not* sentences described above, but it reuses some of the values already computed. As such it is described by a directed acyclic graph of *and/or/not* gates. Such circuits can compute any function from the 2^n assignments of the input variables to the required output. However, by simple counting, we know that most such functions require circuits of size 2^n, i.e. The number of functions on 2^n possible inputs is 2^{2^n}. The number of circuits described by s bits is 2^s. Hence to have as many circuits as functions we need $s\geq 2^n$.

2. Applying the *Selecting Or* rule.

1	$\alpha\vee\beta$	Proved before
2	$\neg\alpha$	Proved before
3	$(\alpha\vee\beta)\wedge\neg\alpha$	Eval/Build from (1) and (2)
4	$((\alpha\vee\beta)\wedge\neg\alpha)\rightarrow\beta$	Axiom
5	β	Modus Ponens from (3) and (4)

3. No answers provided.

4. The following is a prove of the *Selecting Or* statement.

1	Deduction Goal: $((\alpha\vee\beta)\wedge\neg\alpha)\rightarrow\beta$.	
2	$(\alpha\vee\beta)\wedge\neg\alpha$	Assumption
3	$\alpha\vee\beta$	Separating And from (2)
4	$\neg\alpha$	Separating And from (2)
5	$\neg\neg\alpha\vee\beta$	Double Negation from (3)
6	$\neg\alpha\rightarrow\beta$	Contrapositive $\alpha'\rightarrow\beta$ iff $\neg\alpha'\vee\beta$ from (5)
7	β	Modus Ponens from (4) and (6)
8	$((\alpha\vee\beta)\wedge\neg\alpha)\rightarrow\beta$	Deduction Conclusion

5. Love

   (a) Suppose my strategy was to do well in school, but not follow his rules. It turned out that it was the rules he really cares about. Hence, he did not love me. More formally, $\alpha{=}T$, $\beta{=}F$, $\alpha\vee\beta{=}T$, $\gamma{=}F$, and as needed $X = (\alpha\rightarrow\gamma)\vee(\beta\rightarrow\gamma) = (T\rightarrow F)\vee(F\rightarrow F) = F\vee T = T$.

   (b) Being careful:

1	$(\alpha\rightarrow\gamma)\vee(\beta\rightarrow\gamma)$	X is given
2	α	Being careful
3	β	Being careful
4	Deduction Goal:	$(\alpha\rightarrow\gamma)\rightarrow\gamma$.
5	$\quad\alpha\rightarrow\gamma$	Assumption
7	$\quad\gamma$	Modus Ponens from (2) and (5)
8	$(\alpha\rightarrow\gamma)\rightarrow\gamma$	Deduction Conclusion
9	$(\beta\rightarrow\gamma)\rightarrow\gamma$	Similar to (8)
10	γ	Cases (1), (8), and (9)

2 Existential and Universal Quantifiers over Objects

This section introduces *objects* into our logic. Existential $\exists$ and universal/forall $\forall$ quantifiers provide an extremely useful language for making formal statements about them. A game between a prover and a verifier is a level of abstraction within which it is easy to understand and prove such statements.

The Domain of a Variable: As in Section 1, α represents a sentence/formula/statement that is either *true* or *false*. Now each variable x represents an object. There must be an understood *domain/set* of objects that the variables might take on. For example, this could be the set of integers or the set of people. If one wants to talk about two different types of objects within the same logic, then one always has to keep using *predicates* like $girl(x)$ to test whether object x is a girl. This can be cumbersome. Informally, it might be assumed that x is a real, i is an integer, p is a person, g is a girl, b is a boy and so on. Even "the" set of girls needs to be clarified whether it means all girls in the room or all that have ever existed.

The *Loves* Example: Suppose the relation $Loves(p_1, p_2)$ means that person p_1 loves person p_2.

Expression	**Meaning**
$\exists p_2\ Loves(Sam, p_2)$	"Sam loves somebody."
$\forall p_2\ Loves(Sam, p_2)$	"Sam loves everybody."
$\exists p_1 \forall p_2\ Loves(p_1, p_2)$	"Somebody loves everybody."
$\forall p_1 \exists p_2\ Loves(p_1, p_2)$	"Everybody loves somebody."
$\exists p_2 \forall p_1\ Loves(p_1, p_2)$	"There is one person who is loved by everybody."
$\exists p_1 \exists p_2\ \big(\ Loves(p_1, p_2) \text{ and } \neg Loves(p_2, p_1)\ \big)$	"Somebody loves in vain."

Definition of Relation: A relation like $Loves(p_1, p_2)$ states for every pair of objects $p_1 = Sam$ and $p_2 = Mary$ that the relation either holds between them or does not. Though we will use the word *relation*, $Loves(p_1, p_2)$ is also considered to be a *predicate*. The difference is that a predicate takes only one argument and hence focuses on whether the property is *true* or $false$ about the given tuple $\langle p_1, p_2\rangle = \langle Sam, Mary\rangle$.

Representations: Relations (predicates) can be represented in a number of ways.

Functions: A relation can be viewed as a function mapping tuples of objects either to $true$ or to $false$, for example $Loves : \{p_1 \mid p_1 \text{ is a person }\} \times \{p_2 \mid p_2 \text{ is a person }\} \Rightarrow \{true, false\}$.

Set of Tuples: Alternatively, it can be viewed as a set containing the tuples for which it is true, for example $Loves = \{\langle Sam, Mary\rangle, \langle Sam, Ann\rangle, \langle Bob, Ann\rangle, \ldots\}$. $\langle Sam, Mary\rangle \in Loves$ iff $Loves(Sam, Mary)$ is true.

Directed Graph Representation: If the relation only has two arguments, it can be represented by a directed graph. The nodes consist of the objects in the domain. We place a directed edge $\langle p_1, p_2\rangle$ between pairs for which the relation is true. If the domains for the first and second objects are disjoint, then the graph is bipartite. Of course, the *Loves* relation could be defined to include $Loves(Bob, Bob)$. See Figure 3.

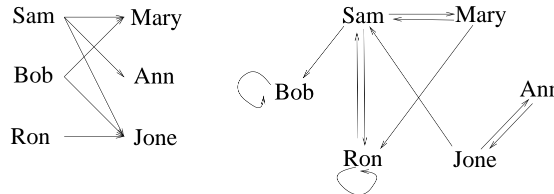


Figure 3: A directed graph representation of the *Loves* relation.

Functions: Functions like $f(x)$ and $x{+}y$ takes a single or a tuple of objects from the universe as input and returns a single object.

Quantifiers: You will be using the following quantifiers and properties.

The Existence Quantifier: The exists quantifier $\exists$ means that there is at least one object in the domain with the property. This quantifier relates the boolean operator *OR*. For example, $\exists p_1\ Loves(Sam, p_1) \equiv [Loves(Sam, Mary)\ OR\ Loves(Sam, Ann)\ OR\ Loves(Sam, Bob)\ OR\ \ldots]$.

The Universal Quantifier: The universal quantifier $\forall$ means that all of the objects in the domain have the property. It relates the boolean operator *AND*. For example, $\forall p_1\ Loves(Sam, p_1) \equiv [Loves(Sam, Mary)\ AND\ Loves(Sam, Ann)\ AND\ Loves(Sam, Bob)\ AND\ \ldots]$.

Combining Quantifiers: Quantifiers can be combined. The order of operations is such that $\forall p_1 \exists p_2\ Loves(p_1, p_2)$ is understood to be bracketed as $\forall p_1\ [\exists p_1\ Loves(p_1, p_2)]$, namely "Every person has the property "he loves some other person"". It relates to the following boolean formula.

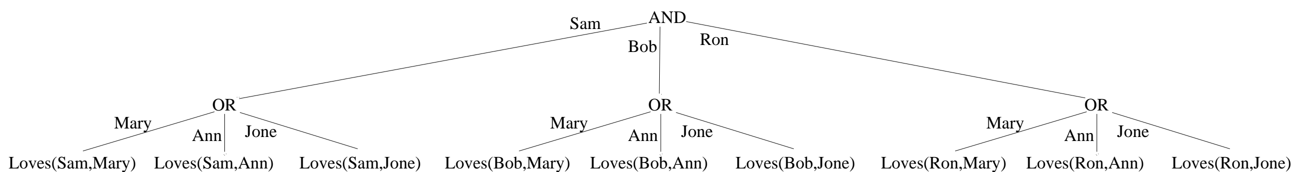


Figure 4: The tree representing $\forall$ boys b $\exists$ girl g $Loves(b, g)$. Note that $\forall$ relates to *AND* and $\exists$ relates to *OR*.

Order of Quantifiers: The order of the quantifiers matters. For example, if b is the class of boys and g is the class of girls, $\forall b \exists g\ Loves(b, g)$ and $\exists g \forall b\ Loves(b, g)$ mean different things. The second one states that "The same girl is loved by every boy." To be true, there needs to be a Marilyn Monroe sort of girl that all the boys love. The first statement says that "Every boy loves some girl." A Marilyn Monroe sort of girl will make this statement true. However, it is also true in a monogamous situation in which every boy loves a different girl. Hence, the first statement can be true in more different ways than the second one. In fact, the second statement implies the first one, but not vice versa.

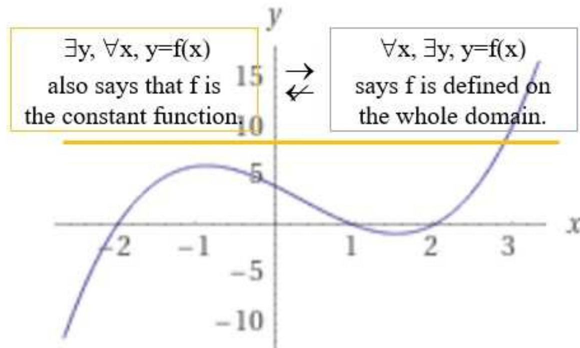


Figure 5: The order of the quantifiers matters.

α(x,y) is a relation between integers.
α(0,0) is true.
α(0,1) is false.

Each input instance is a tuple
⟨0,1⟩ ∈U×U ={0,1,2,3}×{0,1,2,3}
= { ⟨0,0⟩, ⟨0,1⟩, ⟨0,2⟩, ⟨0,3⟩
⟨1,0⟩, ⟨1,1⟩, ⟨1,2⟩, ⟨1,3⟩
⟨2,0⟩, ⟨2,1⟩, ⟨2,2⟩, ⟨2,3⟩
⟨3,0⟩, ⟨3,1⟩, ⟨3,2⟩, ⟨3,3⟩ }
There are |U|×|U|=n×n such tuples:
n choices for x and then n for y.

α(x,y) y, x

α	0	1	2	3
0	T	F	T	F
1	F	T	T	T
2	T	F	F	T
3	T	T	F	F

There are $2^{n\times n}$ relations α:
2 choices for α(0,0), T or F …
$2^{10\times 10}$ is bigger than the universe!
I don't like all of these.

Suppose I want to state that the α that I care about
are each true on some subset S of the tuples.
I don't care about the other tuples.

Denote this set with
S = { ⟨x,y⟩ | x∈U & y=2 }
= { ⟨0,0⟩, ⟨0,1⟩, ⟨0,2⟩, ⟨0,3⟩
⟨1,0⟩, ⟨1,1⟩, ⟨1,2⟩, ⟨1,3⟩
⟨2,0⟩, ⟨2,1⟩, ⟨2,2⟩, ⟨2,3⟩
⟨3,0⟩, ⟨3,1⟩, ⟨3,2⟩, ⟨3,3⟩ }
There are n tuples in this set.
n choices for x and then 1 for y.

α(x,y) y, x

α	0	1	2	3
0	?	?	T	?
1	?	?	T	?
2	?	?	T	?
3	?	?	T	?

You could say
∀α ∀x α(x,2)
that we care about

Figure 6: Considering which inputs $\langle x, y\rangle$ tuples the statement talks about.

Sets of Tuples: Understanding what a logic statement like $\forall x\ \alpha(x, 2)$ says about the relation α can be confusing. The following figure and examples explain this by considering which inputs $\langle x, y\rangle$ tuples the statement talks about.

Similar we could build tables $\alpha(x, y)$ giving examples of the minimum number of input tuples that need to be true so that the following logic statements are true.

$\forall x\ \alpha(x, 2)$: This states that the $y=2$ column of the α table must be entirely true.

$\exists y \forall x\ \alpha(x, y)$: This is the same as $\forall x\ \alpha(x, 2)$ in that one column of the table must be entirely true, but here we do not care which column this is.

$\forall x\ \alpha(x, x)$: The diagonal in the table must be entirely true.

Let $f(0) = 3$, $f(1) = 0$, $f(2) = 1$, $f(3) = 0$, and $\forall x\ \alpha(x, f(x))$: The tuples $\langle 0, 3\rangle$, $\langle 1, 0\rangle$, $\langle 2, 1\rangle$, and $\langle 3, 0\rangle$ must be true. Note that each value of x has at least one value of y for which $\alpha(x, y)$ is true.

$\exists f \forall x\ \alpha(x, f(x))$: This is states the same as the previous example in that each value of x has at least one value of y, but here we do not care which values of y those are.

$\forall x \exists y\ \alpha(x, y)$: This too says the same that each value of x has at least one value of y for which $\alpha(x, y)$ is true.

Skolem/Auxiliary Functions: We have seen that $\forall x \exists y\ \alpha(x, y)$ states that each value for x has at least one value for y for which $\alpha(x, y)$ is true. One can use this fact to define a function $y = f(x)$ which gives you one such value of y for each value of x. It is useful to hold this view of the statement in mind when trying to understand it.

Definition of Free and Bound Variables: The statement $\exists p_2\ Loves(Sam, p_2)$ means "Sam loves someone." This is a statement about Sam. Similarly, the statement $\exists p_2\ Loves(p_1, p_2)$ means "p_1 loves someone." This is a statement about person p_1. Whether the statement is true depends on who p_1 is referring to. The statement is *not* about p_2. The variable p_2 is used as a local variable (similar to $for(i = 1; i <= 10; i++)$) to express "someone." It could be a brother or a friend or a dog. In this expression, we say that the variable p_2 is *bound*, while p_1 is *free*, because p_2 has a quantifier and p_1 does not.

Defining Other Relations: You can define other relations by giving an expression with free variables. For example, you can define the relations $LovesSomeone(p_1) \equiv \exists p_2\ Loves(p_1, p_2)$.

Building Expressions: Suppose you wanted to state that "Every girl has been cheated on" using the *Loves* relation. It may be helpful to break the problem into three steps.

Step 1) Assuming Other Relations: Suppose you have the relation $Cheats(Sam, Mary)$, indicating that "Sam cheats on Mary." How would you express the fact that "Every girl has been cheated on"? The advantage of using this function is that we can focus on this one part of the statement. We are not claiming that every boy cheats. One boy may have broken every girl's heart.

Given this, the answer is $\forall g \exists b\ Cheats(b, g)$.

Step 2) Constructing the Other Predicate: Here we do not have a *Cheats* function. Hence, we must construct a sentence from the *loves* function stating that "Sam cheats on Mary."

Clearly, there must be someone else involved besides Mary, so let's start with $\exists p$. Now, in order for cheating to occur, who needs to love whom? (For simplicity's sake, let's assume that cheating means loving more than one person at the same time.) Certainly, Sam must love p. He must also love Mary. If he did not love her, then he would not be cheating on her. Must Mary love Sam? No. If Sam tells Mary he loves her dearly and then a moment later he tells Sue he loves *her* dearly, then he has cheated on Mary regardless of how Mary feels about him. Therefore, Mary does not have to love Sam. In conclusion, we might define $Cheats(Sam, Mary) \equiv \exists p\ \big(Loves(Sam, Mary) \text{ and } Loves(Sam, p) \big)$.

However, we have made a mistake here. In our example, the other person and Mary cannot be the same person. Hence, we define the relation as $Cheats(Sam, Mary) \equiv \exists p\ \big(Loves(Sam, Mary) \text{ and } Loves(Sam, p) \text{ and } p \neq Mary \big)$. One could argue that Sam does not cheat if Mary knows and is okay with the fact that Sam also love p, but we won't get into this.

Step 3) Combining the Parts: Combining the two relations together gives you $\forall g \exists b \exists p\ \big(Loves(b, g) \text{ and } Loves(b, p) \text{ and } p \neq g \big)$. This statement expresses that "Every girl has been cheated on." See Figure 7.

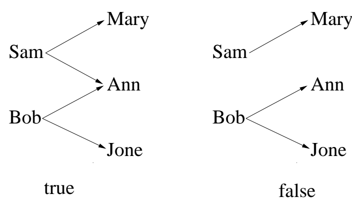


Figure 7: One states that "every girl has been cheated on" with $\forall g \exists b \exists p\ (Loves(b, g) \text{ and } Loves(b, p) \text{ and } g \neq p)$. On the left is an example in which the statement is true, i.e. all three girls have been cheated on. On the right is one in which it is false, i.e. Mary has not been cheated on.

The Negation of a Statement: The negation of a statement is formed by putting a negation on the left-hand side. (Brackets sometimes help.) A negated statement, however, is best understood by moving the negation as deep (as far right) into the statement as possible. This is done as follows.

Negating AND and OR: A negation on the outside of an AND or an OR statement can be moved deeper into the statement using De Morgan's law. Recall that the AND is replaced by an OR and the OR is replaced with an AND.

$\neg \big(Loves(S,M)\ AND\ Loves(S,A) \big)$ iff $\neg Loves(S,M)\ OR\ \neg Loves(S,A)$**:** The negation of "Sam loves Mary and Ann" is "Either Sam does not love Mary or he does not love Ann." He can love one of the girls, but not both.
A common mistake is to make the negation be $\neg Loves(Sam, Mary)\ AND\ \neg Loves(Sam, Ann)$. However, this says that "Sam loves neither Mary nor Ann."

$\neg \big(Loves(S,M)\ OR\ Loves(S,A) \big)$ iff $\neg Loves(S,M)\ AND\ \neg Loves(S,A)$**:** The negation of "Sam either loves Mary or he loves Ann" is "Sam does not love Mary and he does not love Ann."

Negating Quantifiers: Similarly, a negation can be moved past one or more quantifiers either to the right or to the left. However, you must then change these quantifiers from existential to universal and vice versa. Suppose d is the set of dogs. Then

$\neg \big(\exists d\ Loves(Sam, d) \big)$ iff $\forall d\ \neg Loves(Sam, d)$**:** The negation of "There is a dog that Sam loves" is "There is no dog that Sam loves" or "All dogs are not loved by Sam." A common mistake is to state the negation as $\exists d\ \neg Loves(Sam, d)$. However, this says that "There is a dog that is not loved by Sam."

$\neg \big(\forall d\ Loves(Sam, d) \big)$ iff $\exists d\ \neg Loves(Sam, d)$**:** The negation of "Sam loves every dog" is "There is a dog that Sam does not love."

$\neg \big(\exists b \forall d\ Loves(b, d) \big)$ iff $\forall b\ \neg \big(\forall d\ Loves(b, d) \big)$ iff $\forall b \exists d\ \neg Loves(b, d)$**:** The negation of "There is a boy who loves every dog" is "There are no boys who love every dog" or "For every boy, it is not the case that he loves every dog." or "For every boy, there is some dog that he does not love."

$\neg \big(\exists d_1 \exists d_2\ Loves(Sam, d_1)\ AND\ Loves(Sam, d_2)\ AND\ d_1 \neq d_2 \big)$
iff $\forall d_1 \forall d_2\ \neg \big(Loves(Sam, d_1)\ AND\ Loves(Sam, d_2)\ AND\ d_1 \neq d_2 \big)$
iff $\forall d_1 \forall d_2\ \neg Loves(Sam, d_1)\ OR\ \neg Loves(Sam, d_2)\ OR\ d_1 = d_2$**:**
The negation of "There are two (distinct) dogs that Sam loves" is "Given any pair of (distinct) dogs, Sam does not love both" or "Given any pair of dogs, either Sam does not love the first or he does not love the second, or you gave me the same dog twice."

The Domain Does Not Change: The negation of $\exists x \geq 5\ x+2=4$ is $\forall x \geq 5\ x+2 \neq 4$. It is NOT $\exists x < 5 \ldots$. Both the statement and its negation are asking a question about numbers greater or equal to 5. More formally one should write $\exists x\ (x \geq 5 \text{ and } x+2=4)$ and $\forall x\ (x \geq 5 \rightarrow x+2 \neq 4)$. As an exercise, check that these are negations of each other.

3 Proving Via Prover/Adversary/Oracle Game

There are a number of seemingly different techniques for proving that an existential or universal statement is true. The core of all these techniques, however, is the same. Personally, I like to use a *context free grammar* to parse the sentence to be proved and then to have the traversal of its parse tree guide a game between a *prover*, an *adversary*, and an *oracle*. A proof of the sentence is then a winning strategy for the prover.

Proving Via Prover/Adversary Game: We will start by proving sentences that have exists and forall but not imply.

Techniques for Proving $\exists d\ Loves(Sam, d)$:

Proof by Example or by Construction: The classic technique to prove that something with a given property exists is by example. You either directly provide an example, or you describe how to construct such an object. Then you prove that your example has the property. For the above statement, the proof would state "Let d be Fido" and then would prove that "Sam loves Fido."

Proof by Adversarial Game: Suppose you claim to an adversary that "There is a dog that Sam loves." What will the adversary say? Clearly he challenges, "Oh, yeah! What dog?" You then meet the challenge by producing a specific dog d and proving that $Loves(Sam, d)$, that is that Sam loves d. The statement is true if you have a strategy guaranteed to beat any adversary in this game.

- If the statement is true, then you can produce some dog d.
- If the statement is false, then you will not be able to.

Hard Proof: The reason that proving $\exists d\ Loves(Sam, d)$ may be really hard is that you must find a solution d to what might be a very hard computational problem. It could be an uncomputable problem like "will my algorithm ever halt" or an exponentially large search like finding a sequences of n Yes/No inputs that satisfies some circuit.

Techniques for Proving $\forall d\ Loves(Sam, d)$:

Proof by Example Does NOT Work: Proving that Sam loves Fido is interesting, but it does not prove that he loves all dogs.

Proof by Case Analysis: The laborious way of proving that Sam loves all dogs is to consider each dog, one at a time, and prove that Sam loves it.
This method is impossible if the domain of dogs is infinite.

Proof by "Arbitrary" Example: The classic technique to prove that every object from some domain has a given property is to let some symbol represent an arbitrary object from the domain and then to prove that that object has the property. Here the proof would begin "Let d be any arbitrary dog." Because we don't actually know which dog d is, we must either prove $Loves(Sam, d)$ (1) simply from the properties that d has *because* d is a dog or (2) go back to doing a case analysis, considering each dog d separately.

Proof by Adversarial Game: Suppose you claim to an adversary that "Sam loves every dog." What will the adversary say? Clearly he challenges, "Oh, yeah! What about Fido?" You meet the challenge by proving that Sam loves Fido. In other words, the adversary provides a dog d'. You win if you can prove that $Loves(Sam, d')$.
The only difference between this game and the one for existential quantifiers is who provides the example. Interestingly, the game only has one round. The adversary is only given one opportunity to challenge you.
A proof of the statement $\forall d\ Loves(Sam, d)$ consists of a strategy for winning the game. Such a strategy takes an arbitrary dog d', provided by the adversary, and proves that "Sam loves d'." Again, because we don't actually know *which* dog d' is, we must either prove (1) that $Loves(Sam, d')$ simply from the properties that d' has because he is a dog or (2) go back to doing a case analysis, considering each dog d' separately.

- If the statement $\forall d\ Loves(Sam, d)$ is true, then you have a strategy. No matter how the adversary plays, no matter which dog d' he gives you, Sam loves it. Hence, you can win the game by proving that $Loves(Sam, d')$.
- If the statement is false, then there is a dog d' that Sam does not love. Any true adversary (and not just a friend) will produce this dog and you will lose the game. Hence, you cannot have a winning strategy.

Proof by Contradiction: A classic technique for proving the statement $\forall d\ Loves(Sam, d)$ is proof by contradiction. Except in the way that it is expressed, it is exactly the same as the proof by an adversary game.

> By way of contradiction (BWOC) assume that the statement is false, that is, $\exists d\ \neg Loves(Sam, d)$ is true. Let d' be some such dog that Sam does not love. Then you must prove that in fact Sam *does* love d'. This contradicts the statement that Sam does not love d'. Hence, the initial assumption is false and $\forall d\ Loves(Sam, d)$ is true.

Proof by Adversarial Game for More Complex Statements: The advantage to this technique is that it generalizes into a nice game for arbitrarily long statements.

The Steps of Game:

Left to Right: The game moves from left to right, providing an object for each quantifier.

Prover Provides $\exists b$: You, as the prover, must provide any existential objects.

Adversary Provides $\forall d$: The adversary provides any universal objects.

To Win, Prove the Relation $Loves(b', d')$: Once all the objects have been provided, you (the prover) must prove that the innermost relation is in fact true. If you can, then you win. Otherwise, you lose.

The Game Tree: This game can be represented by a game tree.

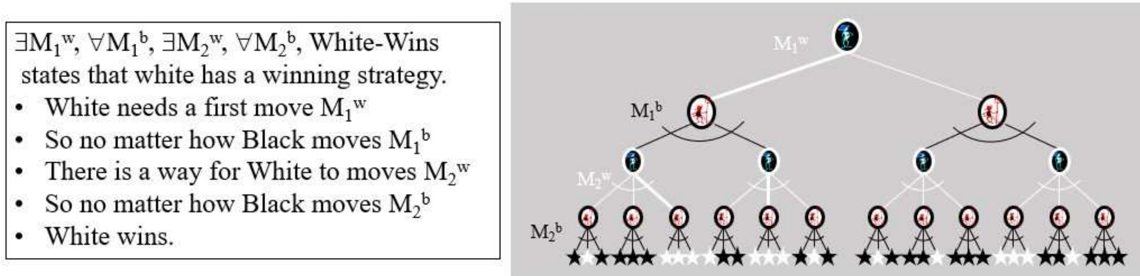


Figure 8: The game tree

Proof Is a Strategy: A proof of the statement consists of a strategy such that you win the game no matter how the adversary plays. For each possible move that the adversary takes, such a strategy must specify what move you will counter with.

Negations in Front: To prove a statement with a negation in the front of it, first put the statement into "standard" form with the negation moved to the right. Then prove the statement in the same way.

Examples:

$\exists b \forall d\ Loves(b, d)$: To prove that "There is a boy that loves every dog", you must produce a specific boy b'. Then the adversary, knowing your boy b', tries to prove that $\forall d\ Loves(b', d)$ is false. He does this by providing an arbitrary dog d' that he hopes b' does not love. You must prove that "b' loves d'."

$\neg \big(\exists b \forall d\ Loves(b, d) \big)$ iff $\forall b \exists d\ \neg Loves(b, d)$: With the negation moved to the right, the first quantifier is universal. Hence, the adversary first produces a boy b'. Then, knowing the adversary's boy, you produce a dog d'. Finally, you prove that $\neg Loves(b', d')$.
Your proof of the statement could be viewed as a function D that takes as input the boy b' given by the adversary and outputs the dog $d' = D(b')$ countered by you. Here, $d' = D(b')$ is an example of a dog that boy b' does not love. The proof must prove that $\forall b \neg Loves(b, D(b))$

Models and Free Variables: If you want prove something, then you need to start with an open mind about the possibilities. Maybe, for example, for some really really big integers $x{+}y \neq y{+}x$. Can you prove otherwise? A *model* in logic defines the *universe* $\mathcal{U}$ of objects which $\forall x$ considers and for each tuple of objects $\langle x, y\rangle \in \mathcal{U}\times\mathcal{U}$ whether or not the relations $Loves(x, y)$ and $x{<}y$ are defined to be true and what the functions $f(x)$ and $x{+}y$ evaluate to. The model also fixes an object for each *free* variable like Sam and x in statements like $\exists p_2\ Loves(Sam, p_2)$ and $\alpha(x)$.

A Given Model: One option is to clearly state the assumed model. For example, a paper could state that it is working over the *standard model* of the integers in which $1{+}1 = 2$ as we all know and love. In this model, the "variable" 0 is not free but is bound to being the constant *zero*.

Valid: A *proof* of a statement ensures that it is true in every possible model in which your axioms/assumptions are true. Such a statement is said to be *valid*. This is the formal approach

to take even when proving things about the integers. One assumes **nothing** about the integers except a small list of axioms like $\exists 0\ \forall x\ x+0=x$ and from those alone prove statements like $\exists 0\ \forall x\ x\times 0=0$ must also be true. See Exercise 4.9. One advantage of doing this is that you are can be sure your proof does not make any unstated assumptions. Another is that it is fun to see what follows from what. Another is that then any statement you prove will automatically also be proved true in any initially unintended models in which the axioms are true, for example the integers mod a prime.

Universal Closure: Wanting valid statements to be true under every model, means that in front of a statement like $\exists p_2\ Loves(Sam, p_2)$, a forall quantifier is implied, namely

$\forall$ universes $\mathcal{U}$ $\forall$ relations $Loves$ $\forall$ objects Sam

[All axioms/assumptions true $\rightarrow \exists p_2\ Loves(Sam, p_2)$].

Adversary: In our game, forall objects are provided by the adversary. Hence, you can start every proof by having him provide a universe of objects $\mathcal{U}$, a worst case relation $Loves$, and an object Sam for which the axioms/assumptions are true.

Counter Example: The negation of the above statement is that there exists at least one model in which the axioms are true and the statement is false. A proof that a statement is not valid only requires describing one such model. It does not need to be model that anyone would ever had in mind. It says that even if your statement is true in your favorite model, you can't prove it from these axioms if it is not true in this strange model for which the axioms are also true. If you are still convinced that your statement is true in your model, you could try adding some extra axioms/assumptions that are true in your model and false in the strange one. With these extra axioms, you might be able to prove your statement.

Free Variable Fail: The assumed forall in front means that the three statements $\alpha(x)$ and $\forall x\ \alpha(x)$ and hence $\forall x'\ \alpha(x')$ are interchangeable. A common mistake is to think that this means that the statement $\alpha(x) \rightarrow \forall x'\ \alpha(x')$ is always true. It is not. Its implied universal closure is not $[\forall x\ \alpha(x)] \rightarrow [\forall x'\ \alpha(x')]$ which is trivially true, but is $\forall x\ [\alpha(x) \rightarrow \forall x'\ \alpha(x')]$ which is not. The adversary can choose that $\mathcal{U} = \{0,1\}$, $\alpha(0)$ is true, $\alpha(1)$ is false, and $x=0$. Then $\alpha(x)$ is true, while $\forall x'\ \alpha(x')$ is not. See Exercise 4.5.

Being Assured by an Oracle: One can't even prove that $1+1=2$ without assumptions about what the symbols $1, 2, +$, and $=$ mean. Hence, we add to our proof system a lot of axioms like $x+y=x+y$ and $x+0=x$. Knowing that these are true, the prover can use them to prove other things. Once other things have been proved, knowing that they are true, he can use them to prove yet other things. Similarly, if the prover wants to prove that $\alpha \rightarrow \beta$, then he temporarily assumes that α is true. A useful way to use the fact that these statements are true is by assuming that the prover has an *oracle* that will assure him of the fact. If the oracle assures the prover of $\forall x\ \alpha(x)$, i.e. $\alpha(x)$ is true for all values of x, then she can allow the prover to specify his favorite value for x_{prover} and she will assure him that $\alpha(x)$ is true for this value. To assure that $\exists y\ \alpha(y)$, i.e. $\alpha(y)$ is true some y, the oracle must construct a value y_{oracle} for y for which it is true.

Example: A famous proof is as follows. Suppose that an oracle assures that $\forall x\ (Human(x) \rightarrow Mortal(x))$ and that $Human(Socrates)$ and the prover wants to prove $Mortal(Socrates)$. This has the form $(\alpha \wedge \beta) \rightarrow \gamma$. No universe is defined, so we have to prove this in every possible universe under any definition of human and mortal. The prover needs to prove $Mortal(Socrates)$ without knowing which object $Socrates$ represents. Because his oracle assures that something is true for every value of x, he can give her this object $Socrates$ and she will assure him of the statement for this object, namely that $Human(Socrates) \rightarrow Mortal(Socrates)$. The prover points out to his oracle that she happens to be assuring him that $Human(Socrates)$. Hence, by Modus Ponens, she can assure him that $Mortal(Socrates)$. This competes the proof.

Follow the Parse Tree: When proving a complex statement, it might be hard to know which rule to follow next. There are can be so many moving parts. The first step is to really understand what you are trying to prove. You can't understand an English sentence until you build a parse tree for it clarifying

which is the subject and which is the verb. Similarly, complex logic statements also need to be parsed. Chapter ?? explains how a context free grammar will both inform you of which sequences of characters form syntactically correct logical statement and allow you to build a parse tree for such a statement that will clarify what it means.

Context Free Parse Tree: A *context free grammar* parses the string $\forall a\ \exists b\ a+b=0$ as follows. See Figure 9. You start with the start non-terminal S_{prover} at the root of the tree. The rule $S_{prover} \Rightarrow \forall X_{adversary}\ S_{prover}$ builds under any node labeled with the LHS, two children labeled with the RHS parts $\forall X_{adversary}$ and S_{prover}. Recurse on all *non-terminal* nodes. The rule $S_{prover} \Rightarrow \exists Y_{prover}\ S_{prover}$ builds under this second node, two children labeled $\exists Y_{prover}$ and S_{prover}. The rule $S_{prover} \Rightarrow R(T,T,T)$ builds under this second node, a nodes labeled $R(T,T,T)$. The rule $T \Rightarrow a|b|0$ builds under the three T, the three *terms* a, b, and 0. The rule $R \Rightarrow _+_=_$ plugs these terms into this relation producing $a+b=0$. The rules $X_{adversary} \Rightarrow a$ and $Y_{prover} \Rightarrow b$ does the same for these variables. The parsing stops here because only *terminal* symbols remain. Reading these off the leaves gives the string $\forall a\ \exists b\ a+b=0$ being parsed.

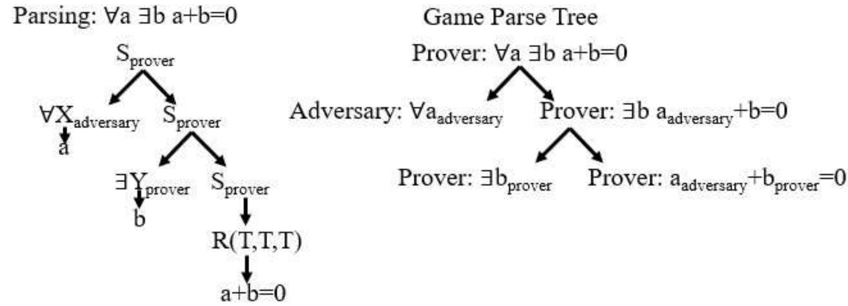


Figure 9: Parse tree for $\forall a \exists b\ a+b=0$.

Game Parse Tree: The proof follows a game dictated by traversing this tree in depth-first-search order. To be clearer, we replace each node label $\forall X_{adversary}$ and $\exists Y_{prover}$ with "Adversary: $\forall a_{adversary}$" and "Prover: $\exists b_{prover}$". When located at such a node, the player in question must provide a specific object for the variable in question. We replace each label S_{prover} and S_{oracle} with "Prover" or "Oracle" followed by the string parsed by this subtree, with the objects constructed so far substituted in. In the game, this is the string that the prover must prove and the oracle must assure the truth of when located at this node. The prover wins if he is able to prove the statement at the leaf. This proof will bubble back up the tree to a proof of the root.

Table of Rules: The table in Figure 10 lists the *contest free grammar* rules I recommend. Each is modified to clarify how the object produced is used. For example, the first rule $S_{prover} \Rightarrow \forall X_{adversary}\ S_{prover}$ is modified to $S_{prover} \Rightarrow \forall x_{adversary}\ S_{prover}(x_{adversary})$. The description on the right of each rule explains how the game plays out for the nodes being traversed. For example, this first rule says "The prover proves $\forall x$ by having his adversary provide him a value $x_{adversary}$ and him going on to prove the statement for this value."

Play Game: Prove the statement $\forall a\ \exists b\ a+b=0$ by traversing the tree. The prover receives a value $a_{adversary}$ from his adversary. He constructs b_{prover} to be the additive inverse $-a_{adversary}$ of the adversary's value. The winner wins because $a_{adversary}+b_{prover}=a_{adversary}+(-a_{adversary})=0$.

Mechanical: The advantage of the game is that it is completely mechanical except for knowing which objects the prover should construct at his nodes labeled "Prover: $\exists g'_{prover}$" and at his oracle's nodes labeled "Prover: $\forall b_{prover}$".

Grammar Rule	Explanation
$S_{prover} \Rightarrow \forall x_{adversary}\ S_{prover}(x_{adversary})$	The prover proves $\forall x$ by having his adversary provide him a value $x_{adversary}$
$\Rightarrow \exists y_{prover}\ S_{prover}(y_{prover})$	The prover proves $\exists y$ by constructing such a value.
$\Rightarrow S_{adversary} \rightarrow S_{prover}$	The prover proves an *implies* by having his oracle assure him of $S_{adversary}$ and using this to prove S_{prover}.
$\Rightarrow S_{prover} \wedge_{both} S_{prover}$	The prover proves an *and* by proving both statements.
$\Rightarrow S_{prover} \vee_{one} S_{prover}$	The prover proves an *or* by proving one of the statements. The prover gets to decide which one.
$\Rightarrow R(T,T)$	Here R represents some relation and the Ts each represent some term which represents some object from your universe. At this point in the game, the prover checks whether or not the relation is true when the constructed values are plugged in.
$S_{oracle} \Rightarrow \forall x_{prover}\ S_{oracle}(x_{prover})$	The oracle assures $\forall x$ by allowing the prover to provide her a value x_{prover}
$\Rightarrow \exists y_{oracle}\ S_{oracle}(y_{oracle})$	The oracle assures $\exists y$ by constructing such a value.
$\Rightarrow S_{prover} \rightarrow S_{oracle}$	The oracle assures an *implies* by having the prover prove S_{prover} and then going on to assure him of S_{oracle}. Sometimes the prover does this with cases.
$\Rightarrow S_{oracle} \wedge_{both} S_{oracle}$	The oracle assures an *and* is true by assuring that both statements are true.
$\Rightarrow S_{oracle} \vee_{cases} S_{oracle}$	The oracle assures an *or* is true by assuring that at least one of the statements is true. Careful, the oracle not the prover gets to decide which one. Hence, the prover will have to consider both cases.
$\Rightarrow R(T,T)$	The oracle assures that the relation is true when the constructed values are plugged in.

Figure 10: The contest free grammar rules for parsing a logical sentence. The prover/adversary/oracle game follows the resulting parse tree.

A Second Example: Let's prove the statement $\exists g \forall b\ Loves(b,g)$ implies $\forall b' \exists g'\ Loves(b',g')$. Clearly, if there is a girl like Marilyn Monro that is loved by every boy then every boy loves at least her. But we want to see how to mechanically produce this proof.

Game Parse Tree: Figure 11 gives the parse tree for the sentence. The root includes the implied universal closure $\forall$ universes $\mathcal{U}$ $\forall$ relations $Loves$. This quantifier is within the left child of the root. For the right child, the grammar rule $S_{prover} \Rightarrow S_{adversary} \rightarrow S_{prover}$ has the prover prove this implies by having his oracle assure him of $S_{adversary}$ and using this to prove S_{prover}. Note that the oracle has her own set of parsing rules, changing "prover proves" to "oracle assures" and "adversary challenges" to "prover asks for help". For example, she assures $\forall x$ not with an object from an adversary but by allowing the prover to provide her a value x_{prover}.

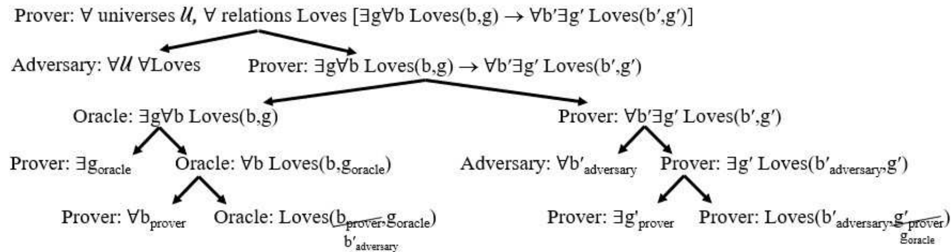


Figure 11: Parse tree for $\exists g \forall b\ Loves(b,g)\ \rightarrow\ \forall b' \exists g'_{prover}\ Loves(b',g')$.

Play Game: The prover proves the statement at the root of the tree by following a "merged" depth first search traversal this tree. He starts by traversing left to the node labeled "Adversary: $\forall$ universes $\mathcal{U}$ $\forall$ relations $Loves$ at which his adversary specifies the universe of people $\mathcal{U}$ and for each tuple of objects $\langle b, g\rangle \in \mathcal{U}\times\mathcal{U}$ whether or not the relation $Loves(b, g)$ is defined to be true. Moving to the right node, the prover must prove the statement. From here he traverses left to the node labeled "Oracle: $\exists g \forall b\ Loves(b, g)$" at which his oracle assures him that the given statement is true. The oracle will follow a depth first search traversal of her subtree, but only in an as need basis. Our focus being on the prover, the oracle is left here and the prover moves right to the node labeled "Prover: $\forall b' \exists g'\ Loves(b', g')$" which gives the sub-statement he currently needs to prove. Traversing to the node labeled "Adversary: $\forall b'_{adversary}$", the adversary provides $b'_{adversary}$ and then to the node labeled "$Prover: \exists g'\ Loves(b'_{adversary}, g')$" which gives the statement he now must prove. He traverse to the node labeled "Prover: $\exists g'_{prover}$", but here the prover gets stuck. Not being able to produce this girl on his own, he talks to his oracle. We left her assuring at the node labeled "Oracle: $\exists g \forall b\ Loves(b, g)$". She traverses left, giving the prover the girl g_{oracle} assumed to exist. Going back to the prover at the node labeled "Prover: $\exists g'_{prover}$", he chooses g'_{prover} to be the oracle's girl g_{oracle}, i.e. $g'_{prover} = g_{oracle}$. Traversing to the node to the right, he still must prove $Loves(b'_{adversary}, g'_{prover})$ or equivalently $Loves(b'_{adversary}, g_{oracle})$, i.e. that $Loves$ is true for the adversary's $b'_{adversary}$ and the oracle's g_{oracle}. Again he gets the oracle's help. Traversing to her right node, she assures the prover of $\forall b\ Loves(b, g_{oracle})$. Traversing to "Prover: $\forall b_{prover}$," she allows him to give her his favorite boy b_{prover}. He gives her the boy $b'_{adversary}$ given to him by his adversary, i.e. $b_{prover} = b'_{adversary}$. Traversing right, she assures him that $Loves(b_{prover}, g_{oracle})$ or equivalently $Loves(b'_{adversary}, g_{oracle})$ is true. This proves the prover's leaf statement. The proof wraps up by bubbling back up the tree to prove the statement at the root. Namely, because he constructed a girl g'_{prover} for which $Loves(b'_{adversary}, g'_{prover})$ is true, he knows $\exists g'\ Loves(b'_{adversary}, g')$. Because he did this for a boy $b'_{adversary}$ provided by his adversary, he knows that $\forall b' \exists g'\ Loves(b', g')$. Because he had his oracle assure him of the LHS and using this he proved the RHS, he proofs that LHS$\rightarrow$RHS. Because he did this with the Adversary's worst case relation $Loves$, he know that the statement is true in every model.

Faster: Yes, this story feels unnecessarily complicated, but once you get the hang of it you can do it faster and with the players removed. Assume the LHS: $\exists g \forall b\ Loves(b, g)$. Let g denote the girl stated to exist such that $\forall b\ Loves(b, g)$ is true. To prove the RHS: $\forall b' \exists g'\ Loves(b', g')$, let b' be an arbitrary boy. The assumption $\forall b\ Loves(b, g)$ states that something is true for all b and hence it is true for b', namely $Loves(b', g)$. Having this true for g, proves $\exists g'\ Loves(b', g')$. Given we did this for an arbitrary b' proves $\forall b' \exists g'\ Loves(b', g')$.

A Harder Example: See Exercise 5.1.

More Examples: Below are some more complicated parse trees.

$\boldsymbol{\alpha \rightarrow (\beta \rightarrow \gamma)}$: See Figure 12. The prover proves $\alpha \rightarrow (\beta \rightarrow \gamma)$ by having an oracle assure him of α and then going on to prove $\beta \rightarrow \gamma$. The prover proves $\beta \rightarrow \gamma$ by having a second oracle assure him of β and then going on to prove γ. It would be the same if there was only one oracle that assured him of α and β.

$\boldsymbol{(\alpha \wedge \beta) \rightarrow \gamma}$: The prover proves $(\alpha \wedge \beta) \rightarrow \gamma$ by having an oracle assure him of $\alpha \wedge \beta$ and then going on to prove γ. Amusingly is effectively the same game as that in our first example. It turns out that these two statements are logically equivalent.

$\boldsymbol{(\alpha \rightarrow \beta) \rightarrow \gamma}$: Suppose α is "The material is hard", β is "I will study", and γ is "I will pass". I, the oracle, assure my father $\alpha \rightarrow \beta$, i.e. "if the material is hard, then I will study". From this, my dad the prover whats to prove to himself γ, i.e. "I will pass the test". The table states that the oracle assures the prover of an implies by having the prover prove S_{prover} and then going on to assure him of S_{oracle}. As suggested, the prover will do this by cases. In the first case, α i.e. "The material is hard". Then I as the oracle as promised assure the prover that β i.e. "I will study". From this, the prover on his own can prove γ i.e. "I will pass". In the second case, $\neg\alpha$ i.e. "The

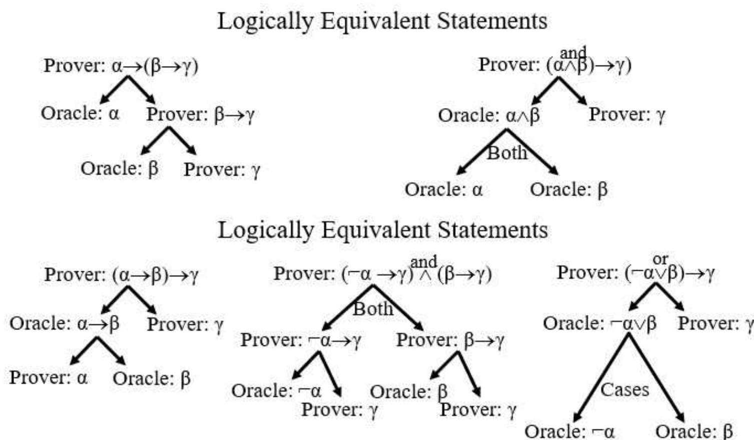


Figure 12: More parse trees

material is not hard". From this, the prover on his own can prove γ i.e. "I will pass". Either way we are good.

$(\neg\alpha \to \gamma) \wedge (\beta \to \gamma)$: To prove this, the prover needs to prove both statements. To prove the first, his oracle assures him that $\neg\alpha$ i.e. "The material is not hard" and he on his own goes on to prove γ i.e. "I will pass". To prove the second, his oracle assures him that β i.e. "I will study" and he on his own goes on to prove γ i.e. "I will pass". Amusingly these are the exact two things that ultimately need to be proved in our previous example. It turns out that these two statements are logically equivalent. See Figure 2.

$(\neg\alpha \vee \beta) \to \gamma)$: To prove this, the prover has his oracle assure him that $\neg\alpha \vee \beta$. Because he does not know which of these the oracle assures him of, he needs to consider both cases. In the first case, his oracle assures him that $\neg\alpha$ i.e. "The material is not hard", and he on his own goes on to prove γ i.e. "I will pass". In the second case, his oracle assures him that β i.e. "I will study", and he on his own goes on to prove γ i.e. "I will pass". Amusingly these are the same things proved for the previous two examples. It turns out that these three statements are logically equivalent.

4 Formal Proof System

We will now very quickly cover *formal proofs*. To ensure that personal beliefs don't lead to false proofs it should adhere to strict mechanical symbol-manipulation rules. No meaning or intuition is needed. I, however, am not convinced that as taught in most texts this teaches students to understand or produce correct proofs. I also feel that the way mathematicians actually write papers is closer to the game described above. If one wants a formal mechanical proof system, I recommend the following. We start with the logic of *true/false* and *and/or/not/implies* given in Section 1. Then we add some new rules for handling objects.

A Formal Proof: A *formal proof* is a sequence of sentences each of which is either an *axiom* or follows from previously proved sentences by one of the *proof systems* rules.

Sound and Complete: A proof system is said to be *sound* if for any statement α, having a proof for it ensures that it is true in every possible model in which your axioms/assumptions are true. Conversely, a proof system is said to be *complete* if every statement α that is true in every possible model in which your axioms/assumptions are true has a proof. Such statements are said to be *valid*. See Gödel's incompleteness and completeness theorems in Section 5.

Equality: Symbols like $+$ and $<$ might get reinterpreted by your model, but the symbol $=$ will not be. In our logic, $x = y$ means that in our model x and y represent the same object. Formalizing *equivalence*

relations requires it to be reflexive: $x\!=\!x$, symmetric: $x\!=\!y$ iff $y\!=\!x$, transitive: $x\!=\!y$ and $y\!=\!z$ implies $x\!=\!z$, and substitutions: $x\!=\!y$ implies $f(x)\!=\!f(y)$ and $\alpha(x)$ is true iff $\alpha(y)$ is true.

Lemmas/Theorems: Proofs can be shortened by proving a lemma once and then using it many times in different settings.

Substitute Objects: For example if $\forall x\ x\times 0 = 0$ is something you have proved as a *lemma*, then this must be true in your model no matter which object x represents. A *term* is any sequence of symbols that represents an object, eg. $2^{f(y)}$. We can substitute such a term into the lemma and be assured that the resulting statement is also true in our model, i.e. $2^{f(y)}\times 0\!=\!0$.

Substitute True/False: For example if $[(\alpha \text{ or } \beta) \text{ and } \neg\alpha] \rightarrow\ \beta$ is is something you have proved as a *lemma*, then this must be true in your model no matter which statement α represents. We can substitute in $\alpha \equiv (\forall x\ \alpha(x))$ and $\beta \equiv (\exists y\ \gamma(x))$ into the lemma and be assured that the resulting statement is also true in our model, i.e. $[((\forall x\ \alpha(x)) \text{ or } (\exists y\ \gamma(x))) \text{ and } \neg(\forall x\ \alpha(x))] \rightarrow\ (\exists y\ \gamma(x))$.

Free Variables x vs Fixed Objects $x_\exists$: Generally, the sentence $\alpha(x)$ means that α is true for an arbitrary value x and hence from it $\forall x\ \alpha(x)$ can be concluded. In order to make each line of the proof have a clear meaning, I introduce the symbol $y_\exists$ to mean a fixed object which can depend on the definition of α but not on value of free variables x. This can be extended to $y_\exists(x)$ being a fixed function from values of x to values of y. As such $\alpha(x, y_\exists)$ means $\exists y \forall x\ \alpha(x, y)$ and $\alpha(x, y_\exists(x))$ means $\exists$ function $y_\exists\ \forall x\ \alpha(x, y_\exists(x)))$ which in turn means $\forall x \exists y\ \alpha(x, y)$.

Deduction with Free Variables: Above we proved that the statement $\alpha(x)\ \ \rightarrow\ \ \forall x'\ \alpha(x')$ is not true in every model. The following rule is included in order to ensure that our proof system is unable to prove it. Before we showed that one proves $\alpha\rightarrow\beta$ by assuming α, proving β, and concluding $\alpha\rightarrow\beta$. This is complicated when the assumption α (or an axiom) contains free variables. To fix the problem, we introduce a new type of symbol $x_\rightarrow$ and change the rule to assuming $\alpha(x_\rightarrow)$, proving $\beta(x_\rightarrow)$, and concluding $\alpha(x) \rightarrow \beta(x)$. Within the assumption block we are only assuming that $\alpha(x_\rightarrow)$ is true for some object $x_\rightarrow$. Hence, from $\alpha(x_\rightarrow)$ you can conclude $\exists x\ \alpha(x)$, but the subscript $\rightarrow$ on the x prevents you from concluding $\forall x\ \alpha(x)$. After the assumption block, we conclude $\forall x\ [\alpha(x) \rightarrow \beta(x)]$, because we did this for an arbitrary object $x_\rightarrow$.

Rules (Adding/Removing $\forall/\exists$): These help define and to work with quantifiers.

Removing $\forall$: From line $\forall x\ \alpha(x)$, include line $\alpha(term(x))$ (eg $\alpha(x)$).

Adding $\forall$: From line $\alpha(x)$, include line $\forall x\ \alpha(x)$. Cannot be done for fixed $x_\exists$ or $x_\rightarrow$.

Removing $\exists$: From line $\exists y\ \alpha(y)$, include line $\alpha(y_\exists)$. From line $\exists y\ \alpha(x, y)$, include line $\alpha(x, y_\exists(x))$. Note $y_\exists$ is a fixed object while $y_\exists(x)$ depends on x. If needed use $y1_\exists$, $y2_\exists$, ... to make sure they are not reused.

Adding $\exists$: From line $\alpha(term)$, include line $\exists y\ \alpha(y)$. Cannot be done if *term* depends on x bounded with $\forall x$.

Negating $\forall$ & $\exists$: $\neg\exists x\ \alpha(x)$ iff $\forall x\ \neg\alpha(x)$.

Proof: The following is a formal proof of $[\exists g \forall b\ Loves(b, g)] \rightarrow [\forall b' \exists g'\ Loves(b', g')]$. I could not help but surround the formal sentences with lots of intuition, but mechanical rules are being followed.

Prove by *Deduction*:

$\exists g \forall b\ Loves(b, g)$	Assumption
$\forall b\ Loves(b, g_\exists)$	Let $g_\exists$ denote the object stated to exist.
$Loves(b, g_\exists)$	If true forall b, then true for b as a *free variable*.

The assumed meaning in each line is that the exists variables $g_\exists$, are fixed first so as to not depend on the free variables b.

$\exists g'\ Loves(b, g')$	Shown to be true for $g_\exists$.

This is a weakening of the previous statements.

There is an implied "forall values of the free variables b"
on the outside on which the value of g that exists might depend.

$\forall b' \exists g'\ Loves(b', g')$ Shown to be true for free b.

LHS $\rightarrow$ RHS Deduction conclusion

The reverse proof is not true.
Prove by *Deduction*:

$\forall b' \exists g'\ Loves(b', g')$ Assumption

$\exists g'\ Loves(b', g')$ If true forall b', then true for b' as a *free variable*.

$Loves(b', g'_{\exists}(b'))$ Let $g'_{\exists}$ denote the object stated to exist.

The assumed meaning in each line is that the exists variables $g'_{\exists}$ are fixed first.
But here $g'_{\exists}$ is fixed to be a function mapping the boy indicated by free variable b'
to the girl $g'_{\exists}(b')$ stated to exist.

$\forall b\ Loves(b, g'_{\exists}(b))$ Shown to be true for free b'.

All previous steps are correct and the meaning of each line is identical to that of the first.

$\exists g \forall b\ Loves(b, g)$ Shown to be true for $g_{\exists}$.

This RHS does not follow from our LHS.
The rules are designed to disallow the last step by not allowing an $\exists g$ to be added
for term $g_{\exists}(b)$ if the b has already been quantified with $\forall b$.

Exercise 4.1 *Let $Loves(b, g)$ denote that boy b loves girl g. If Sam loves Mary and Mary does not love Sam back, then we say that "Sam loves in vain."*

1. *Express the following statements using universal and existential quantifiers. Move any negations to the right.*

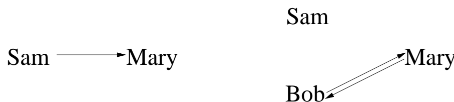
   (a) *"Sam has loved in vain."*

   (b) *"There is a boy who has loved in vain."*

   (c) *"Every boy has loved in vain."*

   (d) *"No boy has loved in vain."*

2. *For each of the above statements and each of the two relations below either prove that the statement is true for the relation or that it is false.*



Exercise 4.2 *(See solution in Section 6) For each prove whether true or not when each variable is a real value. Be sure to play the correct game as to who is providing what value.*

1) $\forall x\ \exists y\ x + y = 5$ *2)* $\exists y\ \forall x\ x + y = 5$

3) $\forall x\ \exists y\ x \cdot y = 5$ *4)* $\exists y\ \forall x\ x \cdot y = 5$

5) $\forall x\ \exists y\ x \cdot y = 0$ *6)* $\exists y\ \forall x\ x \cdot y = 0$

7) $[\forall x\ \exists y\ P(x, y)] \Rightarrow [\exists y\ \forall x\ P(x, y)]$

8) $[\forall x\ \exists y\ P(x, y)] \Leftarrow [\exists y\ \forall x\ P(x, y)]$

9) $\forall a\ \exists y\ \forall x\ x \cdot (y + a) = 0$

10) $\exists a\ \forall x\ \exists y\ [x = a \text{ or } x \cdot y = 5]$

Exercise 4.3 *The game Ping has two rounds. Player-A goes first. Let m_1^A denote his first move. Player-B goes next. Let m_1^B denote his move. Then player-A goes m_2^A and player-B goes m_2^B. The relation $AWins(m_1^A, m_1^B, m_2^A, m_2^B)$ is true iff player-A wins with these moves.*

1. *Use universal and existential quantifiers to express the fact that player-A has a strategy in which he wins no matter what player-B does. Use $m_1^A, m_1^B, m_2^A, m_2^B$ as variables.*

2. *What steps are required in the Prover/Adversary technique to prove this statement?*
3. *What is the negation of the above statement in standard form?*
4. *What steps are required in the Prover/Adversary technique to prove this negated statement?*

Exercise 4.4 *Why does* $[\forall n_0,\ \exists n > n_0,\ P(n)]$ *prove that there are an infinite number of values* n *for which the property* $P(n)$ *is true?*

Exercise 4.5 *(See solution in Section 6) Building the parse tree for* $[\forall x\ \alpha(x)] \rightarrow [\forall x'\ \alpha(x')]$ *and for* $\forall x\ [\alpha(x) \rightarrow \forall x'\ \alpha(x')]$. *Play the proof game for each.*

Exercise 4.6 *(See solution in Section 6) For each of the following either play the logic game to prove the statement is true or find a counter example, i.e. a table* $\alpha(x,y)$ *in which the left hand side is true and the right hand side is false.*

1. $\forall x\ \alpha(x) \rightarrow \exists y\ \alpha(y)$
2. $\exists y\ \alpha(y) \rightarrow \forall x\ \alpha(x)$
3. $\exists y \forall x\ \alpha(x,y) \rightarrow \exists y'\ \alpha(y',y')$
4. $\forall x \exists y\ \alpha(x,y) \rightarrow \exists y'\ \alpha(y',y')$
5. $\forall y\ \alpha(y,y) \rightarrow \forall x' \exists y'\ \alpha(x',y')$

Exercise 4.7 *(See solution in Section 6) For which* α *and* f *are the following true and for which is is false?*

1. $(\forall y\ \alpha(y)) \rightarrow (\forall x\ \alpha(f(x)))$
2. $(\forall x\ \alpha(f(x))) \rightarrow (\forall y\ \alpha(y))$
3. *Answer the previous question knowing that* $\forall y' \exists x'\ y' = f(x')$.
4. $(\exists x\ \alpha(f(x))) \rightarrow (\exists y\ \alpha(y))$
5. $(\forall x\ \alpha(f(x))) \rightarrow (\exists y\ \alpha(y))$

- Answer:
    1. This is true for every α and every f. The oracle assures the prover that $\forall y\ \alpha(y)$ is true. The prover proves $\forall x\ \alpha(f(x))$ by getting his adversary to give him a value for $x_{adversary}$ and then proving $\alpha(f(x_{adversary}))$ for this value. Because his oracle assures him $\forall y\ \alpha(y)$, the prover can give the value $f(x_{adversary})$ to her for y_{prover} and she will assure $\alpha(y_{prover})$ for his value, namely she assures him of $\alpha(f(x_{adversary}))$ as needed. This completes his game.
    2. This is not true when $\forall x\ f(x)=0$ and $\alpha(y)$ is true only for $y=0$ because $\forall x\ \alpha(f(x))$ is then true and $\forall y\ \alpha(y)$ is false.

       The statement is only true when function f is *onto*, namely every value of y has some value of x for which $y=f(x)$, i.e. $\forall y' \exists x'\ y'=f(x')$.
    3. The first oracle assures the prover that $\forall y' \exists x'\ y'=f(x')$ is true and the second that $\exists x\ \alpha(f(x))$ is true. In order to prove $\forall y\ \alpha(y)$, the prover gets from his adversary a value $y_{adversary}$ and he must prove $\alpha(y_{adversary})$ for this value. Because his first oracle assures him of $\forall y' \exists x'\ y'=f(x')$, he can give her this value $y_{adversary}$ for y'_{prover} and she assures him of $\exists x'\ y_{adversary} = f(x')$. Assuring him of this, she gives him a value x'_{oracle} such that $y_{adversary}=f(x'_{oracle})$. Because his second oracle assures him of $\forall x\ \alpha(f(x))$, he can give her this value x'_{oracle} for x_{prover} and she assures him of $\alpha(f(x'_{oracle}))$. Because $y_{adversary}=f(x'_{oracle})$, he knows $\alpha(y_{adversary})$ as needed. This completes his game.

4. This is true for every α and every f. The oracle assures the prover that $\exists x\ \alpha(f(x))$ is true. In order to prove $\exists y\ \alpha(y)$, the prover must construct a value for y_{prover} and prove $\alpha(y_{prover})$ for this value. Because his oracle assures him of $\exists x\ \alpha(f(x))$, she gives him a value x_{oracle} for which $\alpha(f(x_{oracle}))$ is true. The prover sets y_{prover} to be $f(x_{oracle})$ which gives him $\alpha(y_{prover})$ as needed. This completes his game.
5. Exercise 4.6.1 proved $\forall x\ \alpha(x) \rightarrow \exists y\ \alpha(y)$. This is almost the same as $(\forall x\ \alpha(f(x))) \rightarrow (\exists x\ \alpha(f(x)))$. The last question proves $(\exists x\ \alpha(f(x))) \rightarrow (\exists y\ \alpha(y))$. Transitivity then gives us what we want.

Exercise 4.8 *For each either play the logic game to prove the statement is true or find a counter example, i.e. a tables $\alpha(x)$ and $\beta(x)$ in which the left hand side is true and the right hand side is false.*

1. $[\forall x\ (\alpha(x) \rightarrow \beta(x))] \rightarrow [\forall x\ \alpha(x) \rightarrow \forall x\ \beta(x)]$
2. $[\forall x\ \alpha(x) \rightarrow \forall x\ \beta(x)] \rightarrow [\forall x\ (\alpha(x) \rightarrow \beta(x))]$
3. $[\exists x(\alpha(x) \vee \beta(x))] \rightarrow [(\exists x \alpha(x)) \vee (\exists x \beta(x))]$
4. $[(\exists x \alpha(x)) \vee (\exists x \beta(x))] \rightarrow [\exists x(\alpha(x) \vee \beta(x))]$
5. $[\exists x(\alpha(x) \wedge \beta(x))] \rightarrow [(\exists x \alpha(x)) \wedge (\exists x \beta(x))]$
6. $[(\exists x \alpha(x)) \wedge (\exists x \beta(x))] \rightarrow [\exists x(\alpha(x) \wedge \beta(x))]$

Exercise 4.9 *(See solution in Section 6) A* Field *has a universe U of values, two operations: $+$ and $\times$, and the following axioms.*

+ Identity: $\exists 0\ \forall a\ a{+}0{=}a$
× Identity: $\exists 1\ \forall a\ a{\times}1{=}a$
Associative: $a{+}(b{+}c) = (a{+}b){+}c$ *and* $a{\times}(b{\times}c) = (a{\times}b){\times}c$
Commutative: $a{+}b = b{+}a$ *and* $a{\times}b = b{\times}a$
Distributive: $a{\times}(b{+}c) = (a{\times}b){+}(a{\times}c)$
+ Inverse: $\forall a\ \exists b\ a{+}b = 0$, *i.e.* $b{=}{-}a$
× Inverse: $\forall a{\neq}0\ \exists b\ a{\times}b = 1$, *i.e.* $b{=}\frac{1}{a}$

1. *Which of these are fields:* Reals*;* Complex Numbers*;* Rationals/Fractions*;* Integers*; and* Invertible Square Matrices*?*
2. *Let "3" be a short form notation for* $1{+}1{+}1$*. Prove from the above axioms that* $3{\times}4{=}12$*.*
3. *Does it follow from these axioms that $a{\times}0 = 0$? Warning: The proof is hard. Be sure not to use any facts about the reals that is not listed above. There is no rule that mentions both $\times$ and 0. All rules are paired giving a symmetry between $\langle +, 0\rangle$ and $\langle \times, 1\rangle$ except the distributive law which tie the two together. The same symmetry ties $a{\times}0 = 0$ and $a{+}1 = 1$. Clearly the later is not true. Hence, any proof of $a{\times}0 = 0$ must use the distributive law.*
4. *What goes wrong with these axioms if zero also has a multiplicative inverse?*
5. *The integers mod prime p form a field with only the objects $\{0, 1, \ldots, p{-}1\}$. From the additional axiom that $7 =_{mod\ 7} 0$, find the multiplicative inverse of 3.*

   The integers mod 6 do not form a field. $2{\times}3{=}6{=}_{mod\ 6}0$ *is called a* zero divisor*. What problems arise from this?*

5 Theory of Computation Via Logic

Let's try covering much of the theory of computation taught in a standard undergrad computer science degree: data structures, induction, computable vs uncomputable, deterministic vs probabilistic algorithms, time complexity, big-oh notation, coding vs execution phase, compiling a JAVA program into a TM, universal TM, computable vs acceptable, polynomial time vs NP, reductions, proof that the halting problem

is uncomputable, and Gödel's incompleteness and completeness theorem. You must be saying "too hard"!!! but our goal is much more humble. We want first year students to be able to read and understand the key quantification statements that arise from these topics. Such statements are hugely expressive. For any mathematical statement you want to understand and prove, it is best to first write it as a logic sentence. From this, the mechanics of the proof sometimes just falls out.

Graph Data Structure: A *directed graph* is an object consisting of nodes $u, v, \ldots$, and edges $\langle u, v\rangle$. In logic, we might use a predicate/relation $Edge(u,v)$ to be true if and only if $\langle u, v\rangle$ is an edge of our graph.

Out Degree One: We can use logic to specify some property of the graph. For example, we could say that every node has degree exactly one as follows.

Logic: $\forall$ nodes u, $\exists$ a node v, $[Edge(u,v)$ and $\forall$ nodes $v' \neq v$, $\neg Edge(u,v')]$.

Game: If the prover is claiming this is true, then when a adversary gives him some node u, he can name a node v such that $\langle u, v\rangle$ is an edge. Then to test that this is the only such edge, the adversary can give him another node v' different from v and the prover can verify that $\langle u, v'\rangle$ is not an edge.

Connected: If you can only talk about nodes and edges, then you can't say that the graph is connected or that it is finite. On the other hand, if you can talk about paths, then you can define whether the graph is connected.

Logic: $\forall$ nodes s and t, $\exists$ a path p from s to t.
p is such a path if it is an encoding of a sequence of nodes $\langle s, u_1, u_2, \ldots, t\rangle$ such that $\forall i$, $\langle u_i, u_{i+1}\rangle$ is an edge.

Induction: Let $S(i)$ be a yes/no statement, for each positive integer i. For example, it could be the fact that "$1+2+3+\ldots+i = \frac{1}{2}i(i+1)$", that "my iterative algorithm maintains its loop invariants for the first i iterations" or that "my recursive algorithm gives the correct answer on every input of size at most i". The goal is to prove $S(i)$ is true for each i, but there are an infinite number of such i. We do this by assuming that *induction* is an *axiom* of our proof system, namely we will go under the assumption that it is true even though there are models/interpretations/universes within which it is not.

Logic: $\forall S$ [$[S(0)$ and $\forall i$ $[S(i) \rightarrow S(i{+}1)]] \rightarrow \forall i\ S(i)$]

Finite Proof: Given any integer i, it is clear that from $S(0)$ and $\forall i$ $[S(i) \rightarrow S(i{+}1)]$, there is a finite proof of $S(i)$, namely from $S(0)$ and $S(0) \rightarrow S(1)$ we get $S(1)$; from $S(1)$ and $S(1) \rightarrow S(2)$ we get $S(2)$; and so on. But this does not mean that there is necessarily a finite proof of $\forall i\ S(i)$.

Oracle Game: Let's not give the logic game that proves this logic sentence, but instead assume that we have an oracle that assures us of it. The prover uses this to prove $\forall i\ S(i)$ for his favorite statement S as follows. He first proves $S(0)$. Then he let's an adversary give him an arbitrary positive integer i. He assumes that $S(i)$ is true. From that he proves that $S(i{+}1)$ is true. From this he concludes that $\forall i\ S(i)$.

False in Model: Suppose that in addition to the integers, our model/interpretation/universe contains the object ∞. It is possible that $S(i)$ is true for every finite integer i but not for ∞.

Countable: A set S may contain an infinite number of objects, but is considered *countable* if its objects can be "counted", i.e. mapped $1, 2, 3, \ldots$ to the integers. This proves that the "size" of S is the same as that for the set of integers.

Logic: $\exists$ a *List* mapping integers to objects in S, such that $\forall x \in S$, $\exists$ integer i, $List(i) = x$.

Objects with a Finite Description: Let S be the set of all objects that can be described with a finite length English description. For example, it contains strings like "I love you", fractions like $\frac{4}{5}$, finite sets like $\{5, 28, \text{"}cat''\}$, tuples like $[3, [6, 4, [\,], 2], 93]$, special real numbers like π, and algorithms/machines A, because in each case I can describe them in English.

Game: The ith item $List[i]$ in the list is defined as follows. Write down integer i in hexadecimal. Pair the digits and turn every pair into an ASCII character. Read this string of characters and if it makes an understandable description of some object, then make $List[i]$ be this object. The adversary provides an arbitrary object $x \in S$. By the definition of S, x has a finite description. Convert each character in the description into ASCII and string them together to make an integer i in hexadecimal. Note $List(i) = x$.

Reals: A random real number $x \in [0, 1]$ written in binary $0.x_1x_2x_3\ldots$ can be formed by flipping a coin for each bit x_i to the right of the decimal. It almost surely does not have a finite description. We will prove that the set of reals is *uncountable*. We than can prove that the set of functions f from integers to 0/1 is also uncountable by defining a bijection between them and the above reals x, namely by setting $f(i) = 1$ iff the ith bit x_i of x is 1. We can do the same for computational decision problems P by making $P(I)$ be yes iff the ith bit x_i is one, where string I and integer i are related as described above.

Uncountable: $\forall$ $List$, $\exists x \in$ Reals, $\forall$ integer i, $List(i) \neq x$.

Game: Suppose an adversary provides $List$ and claims that it lists every real number. The prover will produce a real number $x = 0.x_1x_2x_3\ldots$ that he claims is not in the list. He chooses each bit $x_{i'}$ to be the flip of the i'th bit of the i'th real number $List(i')$ listed. The adversary provides an integer i claiming that this real number x is listed as the ith real number in his list. But the prover shows that x and $List(i)$ differ in their ith bits. This is called a *diagonalization* proof because the bits considered form a diagonal of $List$.

Computational Problem: Let P be some computational problem. Let I be an input. Let $P(I)$ denote the required output for problem P on input I. For example, $P \equiv Sort$ is an example of a computational problem. We know that the input I to sorting is an array of values. But computer theorists just assume this is encoded by a generic binary string. $Sort(I)$ denotes the same array of numbers as in I but sorted.

Algorithm/Machine: Let A be an algorithm/machine/code. For example, $A \equiv InsertionSort$ is an example of an algorithm. Computer theorists tend to assume that the algorithm is a Turing Machine TM M, but being Java code is the same for our purposes. All we need is that A has a finite description and on any input I what it does is determined.

Let $A(I)$ denote the output produced by algorithm A on input I. Or $A(I)$="Runs for ever", if it does not halt on this input. If bug free, $InsertionSort(I)$ returns the sorted list. If algorithm A is computing problem P, then we need its output $A(I)$ to be as required. Let $A(I) = P(I)$ indicate that algorithm A gives the correct answer for problem P on input I. From a logic perspective, $P(I) = A(I)$ is just another true/false relation between the objects P, A, and I.

Computable: A computational problem P is said to be *computable* if some algorithm A computes it, i.e. gives the correct answer on every input.

Logic: $\exists$ algorithms A, $\forall$ inputs I, $A(I) = P(I)$
Note that this sentence says something about P, because P is free, i.e. not quantified.

Game: The prover proves that P is computable by producing one algorithm A that computes it. He proves that A work on every input by allowing an adversary to choose the input I and then proving that the algorithm returns the correct answer on it, i.e. $A(I) = P(I)$. This is what we do in Computer Science.

WRONG: $\forall$ inputs I, $\exists$ algorithms A, $A(I) = P(I)$.
I want you to feel physical pain when you see this as it allows each input I to have its own algorithm A_I. For example, the algorithm $A_I(I')$ might ignore the input I' and simply outputs the correct answer $P(I)$.

Almost All Computational Problems are Uncomputable: We have seen that the set of algorithms/machines A is countable and the set of computational problems P is uncountable. Because there as so many more problems than algorithms, most problems have no algorithms.

Uncomputable: Every CS student should know the limits of computer science, eg. that the *Halting* problem is *uncomputable*. People tend to struggle trying to understand what this means as their intuition tends to be wrong. As such times, it is helpful to know how to mechanically follow the rules of formal logic. Simply take the negation by moving the negation to the left while switching $\forall$ and $\exists$. P is uncomputable because every algorithm fails to compute it. Algorithm A fails to compute P because there is an input I on which it fails. A fails to compute P on I because it gives the wrong answer.

Logic: $\forall$ algorithms A, $\exists$ input I, $A(I) \neq P(I)$

Game: The prover proves that P is not computable, by allowing an adversary to produce an algorithm A that he claims does compute the stated problem P. To disprove this, the prover just needs to provide one input I on which the algorithm fails to work.

Table Lookup: Table lookup can be viewed as a model of computation. Each algorithm A_{table} is described by a finite table listing input/output pairs. Given an input, it "computes" by looking up the answer. If the input instance is not there, then the algorithm fails. Because the answers are listed, any computational problem can be computed this way. You can build it to work for as many inputs as you like, but this number must always be finite.

Logic: $\forall$ integers k, $\exists$ table algorithm A_{table}, $\forall$ inputs $\langle x, y\rangle \leq \langle k, k\rangle$, $A_{table}(x, y) = x \times y$.

Game: The prover proves that this is true, by allowing an adversary to specify the size k to be as big as he likes, maybe a billion billion billion. The prove then works very hard and builds the multiplication table listing the answer for every pair of integers at most k. When the adversary names one such input $\langle x, y\rangle$, the table algorithm gives the correct answers.

False Logic: $\exists$ table algorithm A_{table}, $\forall$ inputs $\langle x, y\rangle$, $A_{table}(x, y) = x \times y$.

Game: This states that table algorithms can multiply. Let's reverse the game to prove that this is false. An adversary produces an algorithm A_{table} that he claims multiplies. The prover says that this table better be of finite in size, i.e. it can't list an infinite number of answers. Because of this, he is able to find an input $\langle x, y\rangle$ for which the table does not have an answer.

Time Complexity: To be practical, it is important that your algorithms run quickly. We need to allow, however, that the algorithm takes longer on larger inputs. Running in time n^2 or n^3 on inputs of size $n = |I|$ is called *polynomial time* and is completely reasonable. But running in time 2^n is called *exponential time* and is not reasonable.

Size: The *size* $n = |I|$ of input I is formally defined to be the number of bits to describe I. If I is a list of n' objects each described by k bits, then the formal size would be $n = n'k$, but for ease we may say that this size is n'. If I is an integer, DO NOT say that its size is this value because the value $I \approx 2^n$ is exponentially bigger than the number of bits $n = \log I$ needed to describe it.

Time: We could measure time as the number of seconds the program takes on your personal laptop or the number of lines of code executed. These are the same within a multiplicative constant.

Logic: $\exists$ algorithms A, $\exists$ integer c, $\forall$ inputs I, $A(I) = P(I)$ and $Time(A, I) \leq |I|^c$

Game: The prover proves that P is computable in polynomial time, by providing the algorithm A and stating the expected running time. The adversary gives an input I and wins if either the algorithm gives the wrong answer or if the running time is too big.

Constant Time $\mathcal{O}(1)$: We will say a problem can be computed in constant time if the running time does not "depend" on input. Really what we mean is that the time does not grow with the input, i.e. it is bounded by some constant c.

Logic: $\exists$ algorithms A, $\exists$ integer c, $\forall$ inputs I, $A(I) = P(I)$ and $Time(A, I) \leq c$

Game: If the prover is claiming P is computable in constant time, he is allowed to choose the constant c. One might say that this is unfair, because the prover could choose the constant c to be a billion billion. But then the adversary would likely choose an input of size billion billion billion.

BigOh Notation: The exact running time of an algorithm can depend on the programming language and on the hardware, and determining it can be tedious. Hence, theorists tend to ignore the multiplicative constant in front. We say the algorithm runs in $\mathcal{O}(f(n))$ time if the time is bounded by some constant c times $f(|I|)$.

Logic: $\exists$ algorithms A, $\exists$ integer c, $\forall$ inputs I, $A(I) = P(I)$ and $Time(A, I) \leq c \cdot f(|I|)$

Small Inputs: On small inputs, the algorithm may still need a bunch of time for set up. This can be allowed either by not letting the adversary give too small of an input or by simply making c bigger.

Coding vs Execution Phase: We will split the designing and running of an algorithm into two phases. The *coding phase* does not depend on input. At this point, the programmer can decide how many lines of code his algorithm A should have, how many variables and their ranges, and if it is a Turing machine how many states it should have. Compile time errors occur here. In contrast, the *execution phase* can depend on input. It includes the running time and the amount of memory dynamically allocated. Run time errors occur here.

Logic: $\exists$ Algorithm A, $\exists$ number k, $\forall$ Inputs I, $\exists$ time t, $A(I) = P(I)$ and $Lines(A, I) = k$ and $Time(A, I) = t$

Game: Note the prover when designing the algorithm gets to specify the number of lines of code k, before the adversary supplies the input I. But the prover does not need to state the running time t until after he knows the input.

False Logic: $\forall$ numbers k, $\exists$ Algorithm A, $\forall$ Inputs I, $A(I) = P(I)$ and $Lines(A, I) = k$

Game: The adversary is very happy. He knows that problem P is computable. In fact, he knows many algorithms A that solves it. But P is not that easy and it takes some minimum number of lines of code. He does not spend too much time working out the exact minimum. He just says "Let k be two". Now it is the prover that no longer wants to play. He can easily give a two line program. He can easily give a program that solves P. But he can't do both at the same time.

Probabilistic Algorithms: The reason that flipping coins makes writing algorithms easier is because the logic game changes.

Las Vegas Logic: $\exists$ randomized algorithm A, $\forall$ inputs I, [$\forall$ random flips R, $A(I, R) = P(I)$] and $[Exp_R Time(A, I, R) \leq Time(|I|)]$.

Game: The prover proves that P is computable by a randomized algorithm, by telling the adversary his algorithm A, but not telling him the random coin flips R. From this, the adversary must choose his input I. With these, two things must be true. First, whatever coin flip results R the adversary chooses, the algorithm must give the correct answer. This does not sound like an improvement over deterministic algorithms. The advantage the prover has is that for some coin flips R the running time may be really really bad. The second thing that must be true is that for this fixed input, for a random R, the expected running time must be good.

Monte Carlo Logic: $\exists$ randomized algorithm A, $\forall$ inputs I, $Pr_R[A(I, R) \neq P(I)] \leq 2^{-|R|}$.

Game: Unlike Las Vegas algorithms, Monte Carlo ones need not always give the correct answer. As before, the prover provides an algorithm A and the adversary chooses an input I. The prover wins if for most coin flips R, his algorithm works.

Compiling Java to TM: The formal definition of a problem being computable is that a Turing machine can compute it. But writing Turing machines is not fun. Instead, we write our algorithm in some high level programming language like Java and compile it into machine code to run it. To fit the formal definition, one could also compile it into a Turing machine.

Logic: $\forall$ Java Programs J, $\exists$ TM M, $\forall$ inputs I, $J(I) = M(I)$.

Game: To test the prover's compiling ability, the adversary gives some very complicated Java program J, the prover compiles it into a Turing machine M, and the adversary provides an input I on which to test whether the two programs return the same answer.

Exercise: See Exercise 5.1.

Universal TM: We don't tend to build machines whose only job is to run your favorite algorithm A. Instead, we build a *universal* machine that can run any algorithm you give it.

Logic: $\exists$ TM M_U, $\forall$ algorithms A, $\forall$ inputs I, $M_U(\text{“}A\text{”}, I) = A(I)$.

Game: The prover proves that such a universal machine M_U exists by providing one. Then the adversary can give any crazy algorithm A and input I and this machine M_U on input $\langle \text{“}A\text{”}, I\rangle$ will simulate algorithm A on input I and give the same answer $A(I)$. Here “A” denotes the string of characters describing the code for A.

$\exists$ Witness $\Rightarrow$ Acceptable:

A Computable Problem: Given algorithm A, an input I, and a time limit t, the computational problem $Halt(A, I, t)$ determines whether or not $A(I)$ halts in time t.

Algorithm: Run $A(I)$ for t time steps.

An Uncomputable Problem: The computational problem $Halt(A, I)$ determines whether $\exists t$, $Halt(A, I, t)$, i.e. does $A(I)$ ever halt?

Search Space: The number of t to consider is infinite.

A Computable Problem: Given a multi-variable polynomial P eg $x^2y^3 + \ldots$ and an integer assignment/solution S of the variables eg $x = 34325, y = 304, \ldots$, the computational problem $SAT(P, S)$ decides whether or not $P(S) = 0$.

Algorithm: Multiply and sum.

An Uncomputable Problem: The computational problem $SAT(P)$ decides whether $\exists S$, $SAT(P, S)$, i.e. is P satisfiable?

Search Space: The number of S to consider is infinite.

Acceptable Complete: The obvious algorithm for determining whether algorithm A halts on input t is to run it. This algorithm halts when the answer is *yes* and does not halt when the answer is *no*. *Acceptable* is the set/class of computation problems that have such an algorithm. The halting problem is “at least as hard” as every problem in this class.

$\exists$ Witness $\Rightarrow$ Non-Deterministic Polynomial Time:

A Polynomial Time Computable Problem: Given an and-or-not circuit C, a true/false assignment S to the n variables, the computational problem $SAT(C, S)$ determines whether $C(S)$ outputs *true*.

Algorithm: Percolate T/F down wires.

An Non-Poly Computable Problem: The computational problem $SAT(C)$ determines whether $\exists S$, $SAT(C, S)$, i.e. is C satisfiable?

Search Space: The number of S to consider is 2^n.

The Clause-SAT Problem: A common restriction to put on the input circuit C is that it is the *and* of clauses, where each clause is the *or* of literals, where each literal is a variable or its negation, eg $C = (a \text{ or } \neg b \text{ or } d) \text{ and } (\neg a \text{ or } \neg c \text{ or } d \text{ or } e) \text{ and } (b \text{ or } \neg c \text{ or } \neg e) \text{ and } (c \text{ or } \neg d) \text{ and } (\neg a \text{ or } \neg c \text{ or } e)$. A solution is still an assignment, eg $S = \{a = T,\ b = T,\ c = F,\ d = F,\ e = T\}$. The solution S satisfies satisfies the circuit C, if it satisfies each of the clauses. It satisfies a clause, if it satisfies at least one of the literals.

Logic: $SAT(C) \equiv \exists$ a solution S, $\forall$ clauses C_i, $\exists$ variable x, x is an unnegated variable in C_i and is set by S to be true or is an negated variable and is set to false.

NP Complete: *Non-Deterministic Polynomial Time (NP)* is the set/class of computation problems P where each instance I has an exponential number of possible solutions S. Each solution S is either valid or not. There is a poly-time algorithm $Valid(I, S)$ that given an instance I and a solution S, tests whether or not S is a valid solution of I. The output $P(I)$ is *Yes*, if it has a valid solution and *No*, it does not. More formally, $P(I) \equiv \exists S\ Valid(I, S)$. This problem $Clause\text{-}SAT(C)$ is "at least as hard" as every problem in the class NP.

Reductions: It is useful to be able to compare the difficulty of two computation problems, $P_{easier} \leq P_{harder}$. It is hard to prove that P_{harder} is *at least as hard as*, so instead we prove that P_{easier} is *at least as easy as*, by designing an algorithm for P_{easier} using a supposed algorithm/oracle for P_{harder} as a subroutine. Such reductions can be used to create a new algorithm from a known algorithm; to prove that a new problem is likely hard, from knowing a known problem is likely hard; and to learn that two problems have similar "structures".

Logic: $\forall$ inputs I_{easier} to P_{easier}, $P_{easier}(I_{easier}) = P_{harder}(I_{harder})$, where $I_{harder} = InstanceMap(I_{easier})$

Game: In order for the prover to design an algorithm for the computational problem P_{easier}, he starts by getting an input I_{easier} for it from an adversary. From I_{easier}, he produces an input $I_{harder} = InstanceMap(I_{easier})$ for P_{harder}. He gives this input I_{harder} to the supposed algorithm/oracle who returns $P_{harder}(I_{harder})$. He too returns this answer, because he knows that his problem $P_{easier}(I_{easier})$ requires the same answer.

Uncomputable: Every CS student should know the limits of computer science, i.e. that the *Halting* problem is *uncomputable*. Let's start with the statement that some problem is uncomputable.

Logic: $\exists$ problem P_{hard}, $\forall$ algorithms A, $\exists$ input I_A, $A(I_A) \neq P_{hard}(I_A)$

Game: The proof is so easy, you likely won't believe it. Simply play the logic game. The prover needs to produce one computational problem P_{hard} that no algorithm A can solve for every input. An adversary produces one algorithm A that he claims does compute the stated problem P_{hard}. To disprove this, the prove just needs to provide one input I_A on which the algorithm A fails to work.

The prover defines a computational problem P_{hard} by specify for every input I what the required output $P_{hard}(I)$ is. Given some input I, think of it as a sequence of characters and think of this as the JAVA code for some algorithm A_I. If this code compiles and runs, then running this algorithm on its own description $A_I(I)$ does something. Define the required output $P_{hard}(I)$ to be anything different from this. To continue the game, an adversary produces one algorithm A. The prover let's input I_A be the string of characters in the code for A. What algorithm A does on input I_A is $A(I_A)$. By the definition of problem P_{hard}, the required output $P_{hard}(I_A)$ is anything different from this. This completes the game, proving that the sentence is true.

Halting Problem: On input $\langle A, I\rangle$, the halting program asks whether or not algorithm A halts on input I. The intuition is that if A does unpredictable/chaotic things, how can you predict what it will do. One obvious algorithm would be to simply run A on I. If it stops, then your algorithm for the halting problem can report this. But if A does not halt for a billion years, at what point does your algorithm for the halting problem halt and report this.

Halting is Uncomputable: Suppose we had an algorithm that solved the halting problem. We could use this as a subroutine to give a working algorithm for P_{hard}. On input I, ask your halting problem algorithm whether or not algorithm A_I halts on input I. If it does not, then you know what A_I does on I. If it does, then you can safely run A_I on input I to find out what it does. Either way your algorithm for P_{hard} on input I can output something different. This contradicts the fact that P_{hard} is uncomputable. Hence, the assumption that we do have an algorithm for the halting problem must be false.

Hugely Expressive: There are rules of syntax, verses, rhyme, and rhythm that dictate how words can be strung together into poetry. Shakespeare shows how this form is powerful enough to express everything

he has to say about love. The rules of syntax dictate how symbols can be strung together into logical sentences over the integers $(+,\times)$. Variables x and y represent integers. You can use $+$, $\times$, $<$, $=$, $\geq$, $\forall$, $\exists$, $\wedge$ (and), $\vee$ (or), $\neg$ (negation), and $\rightarrow$. The mathematician Gödel wants this form to be powerful enough to express everything he has to say about math. We want this same form to be powerful enough to express everything I have to say about computer science.

"*x* is prime": $\forall a, \forall b, (a \times b = x) \rightarrow (a = 1 \textit{ or } b = 1)$.

***A* on *I* outputs *Y*:** The next example is really quite amazing We are even able to say "Algorithm A on input I outputs Y". One problem, however, is that in our logic we are only able to talk about integers. To get around this, write out the code for A as a string of characters, use ASCII to code each character as a hexadecimal number, concatenated the digits of these numbers together, convert the hexadecimal into decimal and let "A" denote the resulting integer. Similarly, denote "I" and "Y" as integers.

Logic: The logical sentence will be
$\exists C$, "C is an integer encoding a valid computation for algorithm A with input I outputting Y"
Here integer C encodes $\langle C_0, C_1, \ldots, C_T \rangle$, where each C_t encodes the configuration of A at time t.
i.e. C_t specifies the line of code that A is on and the values of everything in memory.
The requirements are that C_0 encodes the starting configuration of A on input I,
$\forall t$, configuration C_{t+1} follows in one time step from configuration C_t and C_T encodes a halting configuration of A with output Y.

Encoding: Consider the tuple $T = [120, 234, 93]$. Gödel encoded this into an integer by considering the first three prime numbers and defining "T" to be the integer $2^{120} \times 3^{234} \times 5^{93}$. Another encoding writes each integers in binary, i.e. $T = [111000_2, 1101010_2, 011101_2]$, replaces the commas with the digit 2 and concatenates, i.e. $T = 1110002110101020111012$, and lets "$T$" be this integer in decimal notation. With either encoding, one can use multiplication to define exponentiation and division and then use these to pick out the individual integers $[120, 234, 93]$ from the integer "T".

Logic Truth is Uncomputable: We can determine whether algorithm A on input I outputs Y by forming the above logical sentence and then asking whether or not this sentence is true. The bad news is that there is no algorithm for the former, hence for knowing whether a sentence is true over the integers $(+,\times)$.

Gödel's Incompleteness Theorem: A theorem that rocked mathematics in 1931 is that there is not a proof system that can correctly prove or disprove every logical sentences over the integers.

The intuition around Gödel's proof is that logic sentences are able to talk about themselves, namely Φ can state that "Φ has no proof in your proof system". If Φ is true, then it true is but has no proof. If it is not true, then it is false yet has a proof of being true. Both are problems.

The computer science proof is easier. If there was a proof system that could prove or disprove every logical sentences over the integers $(+,\times)$, then by simply trying all possible proofs, an algorithm could determine whether a given sentence was true. However, we already stated that this problem was uncomputable.

Gödel's Completeness Theorem: The integers is the standard model/interpretation/universe for which the axioms of your proof system was designed. However, there are always nonstandard models. For example, the integers mod a prime p satisfy all the axioms of the integers $(+,\times)$, but only has p distinct integers. A model specifies which objects are in the universe and defines the result of adding and multiplying any pair of them.

If a sentence Φ is true in some of these and false in others then, your proof system shouldn't be able to proof or disprove Φ. However, Gödel gives a simple proof system and proves that his proof system has a proof of every sentence Φ that is true in every possible model/interpretation/universe. We call such sentences *valid*.

How do you possibly prove that every possible valid statement has a proof when there are true statements about the integers that have no proofs? Given a sentence Φ, start a very simple mechanical but possibly infinite process of building a proof of Φ. If this process terminates then you have a (big but finite) proof that Φ is true in every model M. And if it does not, then there is a simple mechanical method of turning this infinite "proof" into a model M in which is Φ is false. Note that this proves the result. If sentence Φ is true in every model M, then the second option cannot happen. Hence, this process constructs a proof of Φ.

Vector Spaces: Most people think of a *vector* as either being a tuple of values $\langle a_1, a_2, \ldots, a_n \rangle$ or as being an arrow with a direction and a length. However, it can be any abstract object.

Uses: The point of formal proofs is to prove theorems with as few assumptions as possible about the nature of the objects we are talking about so that we can find a wide range of strange new objects for which the same theorems are true.

Rules: The only requirement on a set of vectors is that they can be multiplied by a real number and any two of them can be added together to produce a third with the obvious rules: Associative: $u + (v + w) = (u + v) + w$; Commutative: $u + v = v + u$; Distributive: $a(u + v) = au + av$; and Additive Inverse: $\forall u \exists v \; u + v = 0$, i.e. $v = -u$.

Spanning the Space: From these alone, one can prove the following. Suppose that you have an arbitrary set of n linearly independent basis vectors $\langle w_1, w_2, \ldots, w_n \rangle$, where n is the *dimension* of your space. Then these *span* the entire space, meaning that any vector v in your vector space can be formed from some unique *linear combination* of the basis vectors, namely there are unique real numbers $\langle a_1, a_2, \ldots, a_n \rangle$ such that $v = a_1 w_1 + a_2 w_2 + \ldots + a_n w_n$.

$\forall$ vector spaces V,
  $\forall$ basis $\langle w_1, w_2, \ldots, w_n \rangle$ of vectors,
    $\forall$ vector v in the entire space,
      $\exists$ real coefficients $\langle a_1, a_2, \ldots, a_n \rangle$,
        such that $v = a_1 w_1 + a_2 w_2 + \ldots + a_n w_n$

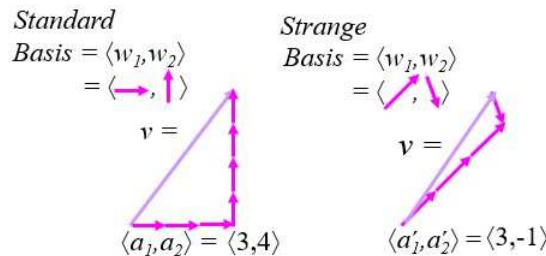


Figure 13: Standard and non-standard bases spanning the vector space.

Standard Euclidean Basis: The *standard* basis $\langle w_1, w_2, \ldots, w_n \rangle$ consists of the vectors of length one that follow each of the n axises, namely $\langle 0, 0, 1, 0, \ldots \rangle$. Then if v is the vector $\langle a_1, a_2, \ldots, a_n \rangle$, then the coefficients giving $v = a_1 w_1 + a_2 w_2 + \ldots + a_n w_n$ are the same values. In this way, the vector v as originally expected can in fact be represented by a tuple $\langle a_1, a_2, \ldots a_n \rangle$.

Change in Basis: What is fun is that one can consider both a standard $\langle w_1, w_2, \ldots, w_n \rangle$ and a *non-standard* basis $\langle w'_1, w'_2, \ldots, w'_n \rangle$. Then the same vector v can be represented with the coefficients $\langle a_1, a_2, \ldots, a_n \rangle$ with respect to the first and $\langle a'_1, a'_2, \ldots, a'_n \rangle$ with respect to the second. Multiplying the first tuple of values $\langle a_1, a_2, \ldots, a_n \rangle$ by an $n \times n$-matrix gives the second $\langle a'_1, a'_2, \ldots, a'_n \rangle$.

Colour: Each colour can be thought of as a vector. Surprisingly this is an infinite dimensional vector space, because there are an infinite number of different frequencies of light and each colour is a linear combination of these. On the other hand, our eyes only have three sensors that detect frequency so our brain only returns three different real values. Hence, we see three dimensional

colour. Each colour we see can be represented by a linear combination of $\langle red, green, blue\rangle$ or of $\langle red, blue, yellow\rangle$. Evidently the colour birds can see is four dimension.

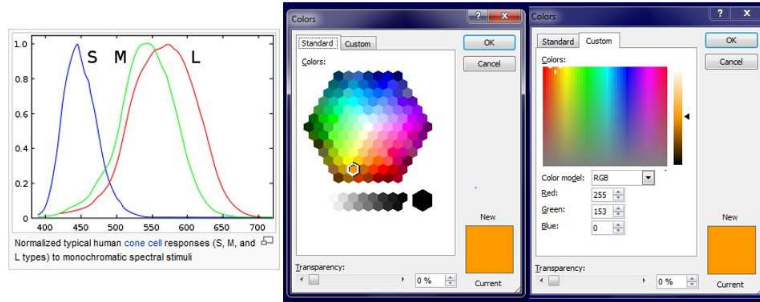


Figure 14: Colour is a vector space.

Fourier Transformations: Consider some signal with strength $v(t)$ at discrete times t. This too can be thought of as a vector. The *time-basis* is $\langle w_1, w_2, \ldots, w_n\rangle$, where each $w_{t'}(t)$ is the signal that is zero everywhere but one at time t'. Note that $v = a_1w_1 + a_2w_2 + \ldots + a_nw_n$ is the signal given by $v(t) = a_t$. The *sin-cos-basis* is $\langle w'_1, w'_2, \ldots, w'_n\rangle$, where each $w'_f(t)$ is either $w'_f(t) = sin(t \cdot \frac{2f\pi}{n})$ with frequency f or the *cos* function. Doing a change of basis to the coefficients $\langle a'_1, a'_2, \ldots, a'_n\rangle$ then gives our signal as a linear combination of sine and cosine waves. This is what a equalizer does.

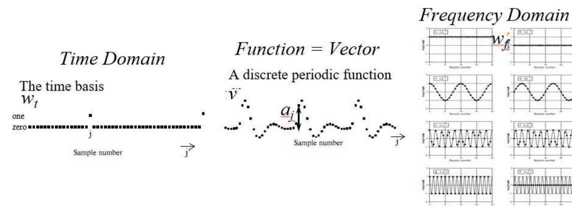


Figure 15: The set of all functions can be represented in the time domain and in the Fourier domain.

JPEG: The image compression program JPEG does the same thing except in two dimensions. It is given an image represented by the brightness of each pixel and converts this into a linear combination of sine and cosine waves. To make the file smaller, it drops details on the high frequency coefficients.

FFT: Generally computing a change of basis takes $\mathcal{O}(n^2)$ time. In some cases, Fast Fourier Transformations can do it in $\mathcal{O}(n \log n)$ time. See Section 9.4.

Polynomial Interpolation: Let $\langle x_1, x_2, \ldots, x_n\rangle$ be any fixed distinct values for x. Consider some function $v(x)$ defined only on these n values. This too can be thought of as a vector. The *time-basis* is $\langle w_1, w_2, \ldots, w_n\rangle$, where each $w_i(x)$ is the function that is zero everywhere but one at x_i. Note that $v = a_1w_1 + a_2w_2 + \ldots + a_nw_n$ is the function given by $v(x_i) = a_i$. The x^d-*basis* is $\langle w'_1, w'_2, \ldots, w'_n\rangle$, where each $w'_d(x)$ is the function x^d. Doing a change of basis to the coefficients $\langle a'_1, a'_2, \ldots, a'_n\rangle$ then gives the coefficients of the unique polynomial $P(x) = a'_0 + a'_1x^+a'_2x^2 + \ldots + a'_{n-1}x^{n-1}$ that passes through these n points. See Section 9.4.

Exercise 5.1 *Define:*

A: "Computational problem P is computable by a Java Program," namely
$\exists$ *Java M* $\forall$ *ASCII I* $\exists$ *time t* $P(I){=}M(I)$ *and* $Time(M,I){=}t$

B: "The computational problem P treats inputs the same whether in binary or ASCII," namely
$\forall$ *binary* I' $P(I'){=}P(I)$ *where* $B(binary){=}ASCII$ *and* $I{=}B(I')$

C: "Java programs can be simulated by TM" and
"The TM takes the square of whatever time Java program takes," namely
$\forall$ *Java* M $\exists$ TM M' $\forall$ *binary*I' $M'(I') = M(I)$
where $M' = Compile_{JAVA \Rightarrow TM}(M)$
and $\forall t$ $[Time(M, B(I')) = t \rightarrow Time(M', I') \leq t^2]$
Z: "Computational problem P is computable by a TM," namely
$\exists$ TM M' $\forall$ *binary* I' $\exists$ *time* t' $P(I') = M'(I')$ *and* $Time(M', I') = t'$
Prove $\langle A, B, C \rangle \rightarrow Z$

1. *Using the prover/adversary/oracle game. Fancy parsing is not required. Prover Z and the three oracles A, B, and C each play their game by reading their statement left to right. These four games merge together. Focus on who gives whom what when.*

2. *Using our formal proof system.*

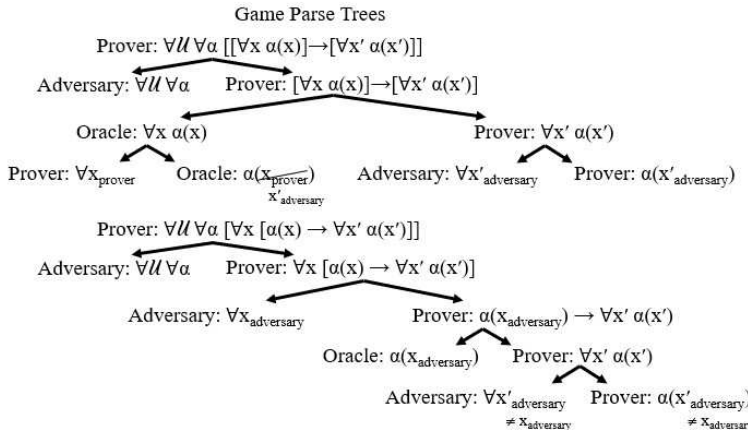
6 Exercise Answers

The following are the answers to some of the exercises.

4.2

1. $\forall x$ $\exists y$ $x + y = 5$ is true. Let x be an arbitrary real value and let $y = 5 - x$. Then $x + y = 5$.
2. $\exists y$ $\forall x$ $x + y = 5$ is false. Let y be an arbitrary real value and let $x = 6 - y$. Then $x + y \neq 5$.
3. $\forall x$ $\exists y$ $x \cdot y = 5$ is false. Let $x = 0$. Then y must be $\frac{5}{0}$, which is impossible.
4. $\exists y$ $\forall x$ $x \cdot y = 5$ is false. Let y be an arbitrary real value and let $x = \frac{6}{y}$ if $y \neq 0$ and $x = 0$ if $y = 0$. Then $x \cdot y \neq 5$.
5. $\forall x$ $\exists y$ $x \cdot y = 0$ is true. Let x be an arbitrary real value and let $y = 0$. Then $x \cdot y = 0$.
6. $\exists y$ $\forall x$ $x \cdot y = 0$ is true. Let $y = 0$ and let x be an arbitrary real value. Then $x \cdot y = 0$.
7. $[\forall x\ \exists y\ P(x, y)] \Rightarrow [\exists y\ \forall x\ P(x, y)]$ is false. Let $P(x, y) = [x + y = 5]$. Then as seen above the first is true and the second is false.
8. $[\forall x\ \exists y\ P(x, y)] \Leftarrow [\exists y\ \forall x\ P(x, y)]$ is true. Assume the right is true. Let y_0 the that for which $[\forall x\ P(x, y_0)]$ is true. We prove the left as follows. Let x be an arbitrary real value and let $y = y_0$. Then $P(x, y_0)$ is true.
9. $\exists a$ $\forall x$ $\exists y$ $[x = a$ or $x \cdot y = 5]$ is true. Let $a = 0$ and let x be an arbitrary real value. If $x \neq 0$, let $y = \frac{1}{x}$. Otherwise, let $y = 5$. Then $[x = a$ or $x \cdot y = 5]$ is true.

4.5



4.6

1. The oracle assures the prover that $\forall x\ \alpha(x)$ is true. In order to prove $\exists y\ \alpha(y)$, the prover must produce some value of y_{prover} for which $\alpha(y_{prover})$ is true. In this case, he can let y_{prover} be anything he wants. His oracle assuring $\forall x\ \alpha(x)$, lets the prover provide any value of x_{prover} that he wants and she will assure him of $\alpha(x_{prover})$ for this value. In this case, the prover provides his y_{prover}. The oracle assures that $\alpha(y_{prover})$ is true for this value. This completes the prover's game.

2. This statement is false when $\alpha(0)$ is true and $\alpha(1)$ is false, because this makes $\exists y\ \alpha(y)$ true and $\forall x\ \alpha(x)$ false.

3. The oracle assures the prover that $\exists y \forall x\ \alpha(x,y)$ is true. In order to prove $\exists y'\ \alpha(y',y')$, the prover needs to find a value y'_{prover} for which α is true on this diagonal. His oracle assuring $\exists y \forall x\ \alpha(x,y)$ gives the prover a value y_{oracle} for which $\forall x\ \alpha(x, y_{oracle})$ is true. Assuring him of $\forall x\ \alpha(x, y_{oracle})$, the oracle allows the prover to give her his favorite x_{prover} and she will assure $\alpha(x_{prover}, y_{oracle})$ for this value. He gives her for x_{prover} this same value y_{oracle}, i.e. $x_{prover} = y_{oracle}$. Hence, what she assures him of is $\alpha(y_{oracle}, y_{oracle})$ is true. In order to prove $\exists y'\ \alpha(y',y')$, the prover sets y'_{prover} to be this value y_{oracle}, $y'_{prover} = y_{oracle}$ giving him as needed that $\alpha(y'_{prover}, y'_{prover})$. This completes the prover's game.

4. This is not true for the following α.

α	$x=0$	1	2	3
$y=0$	F	T	T	T
1	T	F		
2			F	
3				F

   Note that $\forall x \exists y\ \alpha(x,y)$ is true because each column (value of x) has a row (value of y) for which α is true. Note that $\exists y'\ \alpha(y',y')$ is not true because every diagonal is false.

5. The oracle assures the prover that $\forall y\ \alpha(y,y)$ is true. In order to prove $\forall x' \exists y'\ \alpha(x',y')$, the prover gets from his adversary a value for $x'_{adversary}$ and he must prove $\exists y'\ \alpha(x'_{adversary}, y')$ for this value. Because his oracle assures him of $\forall y\ \alpha(y,y)$, he can give this value $y_{prover} = x'_{adversary}$ and she will assure him of $\alpha(x'_{adversary}, x'_{adversary})$. In order to prove $\exists y'\ \alpha(x'_{adversary}, y')$, he choose y'_{prover} to be $x'_{adversary}$. Because his oracle assures him of $\alpha(x'_{adversary}, x'_{adversary})$ and $y'_{prover} = x'_{adversary}$, he knows that $\alpha(x'_{adversary}, y'_{prover})$. This completes his game.

4.9

1. The following are fields: *Reals*; *Complex Numbers*; and *Rationals/Fractions*. The set of *Integers* is not because the multiplicative inverse of 2 is $\frac{1}{2}$ which is not an integer. The *Invertible Square Matrices* is not because it is not multiplicatively communicative, i.e. $M \times N$ might be different than $N \times M$ because spinning an object 90 degrees along the z and then 90 degrees along the x results in a different orientating than doing them in the reverse order.

2. $3 \times 4 = (1+1+1) \times (1+1+1+1) = ((1+1)+1) \times (1+1+1+1) = (1+1) \times (1+1+1+1) + 1 \times (1+1+1+1) = 1 \times (1+1+1+1) + 1 \times (1+1+1+1) + 1 \times (1+1+1+1) = 1+1+1+1+1+1+1+1+1+1+1+1 = 12$.

3. To prove $a \times 0 = 0$, lets start in the middle with the distributive law, $a \times (0+1) = (a \times 0) + (a \times 1)$. The $LHS = a \times (1) = a$ and the $RHS = (a \times 0) + (a)$. Adding $-a$ to both sides gives $LHS = a + (-a) = 0$ and the $RHS = (a \times 0) + a + (-a) = a \times 0$. This gives the result.

4. If $a = \frac{1}{0}$ is zero's multiplicative inverse, then by definition $0 \times a = 1$. Commutativity then gives $a \times 0 = 1$. But our last proof shows that $a \times 0 = 0$. This can be resolved by having $1 = 0$. But then by the $\times$ identity $\forall a\ a \times 0 = a$. But again we just proved that $a \times 0 = 0$. This can be resolved by having $a = 0$, for all values a. This makes a fine field with one element 0.
   Even if ∞ is added to your set of objects, you better have $\infty \times 0 = 0$.

5. In the integers mod 7, the multiplicative inverse of 3 is 5 because $3 \times 5 = 15 = 7+7+1 =_{mod\ 7} 0+0+1 = 1$. The problem that arises with $2 \times 3 = 6 =_{mod\ 6} 0$ is that if 2 has a multiplicative inverse $\frac{1}{2}$, then multiplying both sides by it gives that $3 = 0$.