Lecture 1 - Introduction

Aristotelian Logic

- **Syllogism**: logical argument where a conclusion is inferred from two or more premises
  - Ex. "All humans are mortal" and "Socrates is human" → Socrates is mortal
- This is a good argument because it is truth-preserving
- Correctness depends on the form of the argument, not its specifics
  - The above argument has the form "All x are y" and "B is an x" → "B is a y"
- This type of proof is called a **hypothetical syllogism**

Applications of Logic to CS

- Electric computers (electronic digital circuits) are made up of logic gates
- Logic can be used to minimize the number of components in a circuit
- Artificial Intelligence (knowledge base + inference engine)
- Automated theorem proving / automated proof assistants
- Databases
- Programming (program specification, formal verification)
  - The programming logic PROLOG

Introduction to Propositional Logic

- **Logic**: analysis and appraisal of arguments
  - Logic studies forms of reasoning
- **Argument**: set of statements in the form of premises and a conclusion
- A **valid argument** (or **sound argument**) is one where the conclusion is true if the premises are true
  - An **invalid argument** is the opposite
- Arguments can be compound: "The computer program has a bug or an input error" and "There is no input error" → "The computer program has a bug"
- Here, in the first argument, two statements are connected with `OR`

Types of logical arguments

- Hypothetical syllogism
  - "If P then Q" and "If Q then R" → "If P then R"

- Disjunctive Syllogism
  - "P or Q" and "not Q" $\rightarrow$ "P"
- Modus Ponens (method of placement)
  - "If P then Q" and "P" $\rightarrow$ "Q"
- Modus Tollens (contrapositive)
  - "If P then Q" and "not Q" $\rightarrow$ "not P"

Propositions

- **Proposition**: statement that is either true or false
- In the above section, P, Q and R are **propositional variables**
  - They must have the value of a **propositional constant**: `TRUE` or `FALSE`
- Propositional variables are **atomic propositions**
- **Compound propositions** are created by combining atomic propositions using **logical connectives** (`AND`, `OR`, `NOT`, `IF-THEN`, etc.)

Logical Connectives

- **Negation** (NOT): $\neg P$
- **Conjunction** (AND): $P \wedge Q$
  - Only true if both P and Q are true
- **Disjunction** (OR): $P \vee Q$
- **Implication** (IF-THEN): $P \implies Q$

Lecture 2 - Propositional Language and Lp

Implication

- $p \implies q$ is false when we have "true implies false" and true otherwise
  - p: antecedent
  - q: consequent
- English translation (main): If p then q
  - p is sufficient for q
  - p only if q
  - q if p
- When p is false, the implication is always vacuously true

Equivalence

- $p \iff q$ is true when p and q are the same
  - I.e. p is equivalent to q
  - p if and only if q

Imprecision and Ambiguity

- **Ambiguous sentence**: sentence that has several interpretations
  - Can be eliminated by querying the author of the sentence
- **Imprecise sentence**: lacks rigor
  - Can be eliminated by adding quantitative data

Propositional Language

- **Connective**s can combine propositions, whether they are atomic or not
- We will use full parenthesized expressions to avoid ambiguity about logical statements
  - We cannot assume an order of operations
- $\mathcal{L}^p$ is the language of propositional logic
  - Will contain symbols: p, q, $\wedge$, etc.
  - **Expressions**: finite strings of allowed symbols
    - The empty expression has length 0, denoted ϵ
  - Expressions are equal if they have the same length and the same symbols in the same order

- Two expressions can be **concatenated**
  - **Segment**: "substring" of an expression
    - Proper segment: segment that is not the entire expression
  - Types of segments: initial segment (prefix), terminal segment (suffix)
    - These are "proper" if the segment in question is not ϵ

The set of formulae of $\mathcal{L}^p$

- **Atom**: set of expressions in $\mathcal{L}^p$ that consist of a single symbol
- The set of formulas of $\mathcal{L}^p$ $Form(\mathcal{L}^p)$ is defined recursively as
  - BASE CASE: An atom of $\mathcal{L}^p$ ($Atom(\mathcal{L}^p)$)
  - RECURSIVE CASE: if A and B are formulae in $\mathcal{L}^p$, $\neg A$, $A \wedge B$, etc. are all formulae
- This is the set of formation rules
- Applying parentheses at every recursive step ensures that the formula will be fully parenthesized

Parse tree

- CS 146 flashbacks (for some of you)
- Much like an AST, it is a tree that shows how each subexpression is formed

Claims about $\mathcal{L}^p$

- Every formula in $\mathcal{L}^p$ has the same number of left and right parentheses
- Every proper initial segment has more left parentheses than right parentheses
  - Inverse applies to terminal segments and right parentheses
  - Neither of these segments can itself be a formula because of this
- **Unique readability theorem**: Every formula of $\mathcal{L}^p$ is of exactly one of six forms: an atom, $\neg A$, $A \wedge B$, $A \vee B$, $A \implies B$, or $A \iff B$

Review of Mathematical Induction

- We define the natural numbers as $0, 1, 2, 3 \ldots$
- We define $P(n)$ as $n \in \mathbb{N}$ has property P
- Principle of mathematical induction
  - Prove $P(0)$
  - Prove that $P(n) \implies P(n+1)$ for all $n \in \mathbb{N}$
- Strong induction: at each inductive step, we assume not only the base case, but all previous inductive cases as well

    - This is essentially the same thing as regular induction, but we define the property to apply to all $k \in \mathbb{N} \leq n$

Recursively defined sets

- Definition consists of three parts
    - Base: A statement that a certain object is in the set
    - Recursion: Rules that allow existing objects in the set to be combined into new objects that are also in the set
    - Restriction: A statement that posits that only objects formed by rules 1 and 2 are in the set
- $\mathcal{L}^p$ is defined recursively, so we can use structural induction to prove things about $Form(\mathcal{L}^p)$
- Structural induction permits us to induct on any mathematical object
    - The natural number version of structural induction is strong induction
    - Usually, any structural induction argument can be turned into a strong induction argument

Lecture 3 - Propositional Logic Semantics

Inductive proof for $Form(\mathcal{L}_p)$

- Inductive hypothesis: $P(k)$ for some k
- To show that $P(k+1)$ is true, let formula A have $k+1$ connectives

Subcase $\neg$ (unary logical operator)

- $A = (\neg B)$ where $\neg$ is the $k+1$st connective. Here, B is a formula with the properties (a), (b), and (c) (hypothesis). These properties are the B starts with a (, has an equal number of opposing parens, and is a well-formed formula
  - A holds since we still defined it to start with an open paren
  - B holds since we add one left and one right paren
  - C applies due to the recursive definition of a formula (since B is well-formed)

Subcases $\wedge$, $\vee$, $\implies$, $\iff$ (binary logical operators)

- We will only prove C
- First we show that A can only be written one way, i.e.
  $A = (B * C) = (B' *' C') \implies B' = B \wedge C' = C$
  - If B' has the same length as B, it must be the same since it starts at the same spot
  - So, assume for sake of contradiction that B' is shorter
  - Since B is well=formed, B' must have more left parens than right parens
  - This is a contradiction that B' is well-formed, so it is wrong
  - A similar argument can be used to show that B is a proper prefix of B'
- We must also show that $B * C \neq \neg D$, which it cannot because we know B starts with a left paren

Consequences of unique readability

- Ensures unambiguous formulae
- A formula must be one of the six types defined (atom, negation of formula, conjunction of two formulae, etc.)

Precedence rules (for humans)

- In order from most precedent to least precedent: $\neg$, $\wedge$, $\vee$, $\implies$, $\iff$

Module: Propositional Language: semantics

Syntax vs. Semantics

- Syntax: rules for declaring formulas
  - Logical equivalent to syntax error: string that does not belong in $Form(\mathcal{L}_p)$
- Semantics are considered with the meaning of a formula
  - Atoms represent simple propositions
  - Connectives have their intended meanings
  - The "meaning" of a formula comes from its truth table

Truth valuations

- Truth table: list of the values of a formula under any possible set of values of its propositions
  - If there are n propositions, there are 2^n rows in the truth table
- Truth valuation function: $t : Atom(\mathcal{L}^p) \to \{0, 1\}$
- The valuation of a formula is defined using structural recursion
  - Form: P -> value is P
  - Form $\neg P$ -> value is 0 if P is 1, 1 if P is 0
  - Form $P \wedge Q$ -> value is 1 if P = Q = 1, 0 otherwise
  - etc.
- A formula can have different truth valuations if the variables (propositions) have different values
  - A truth valuation t satisfies a formula $A \iff A^t = 1$
- $\Sigma^t = \{1 \text{ if } B \in \Sigma, B^t = 1, 0 \text{ otherwise } \}$
  - Set of formulas that are true under valuation t
- A set of formulas $\Sigma \subseteq Form(\mathcal{L}^p$ is satisfiable $\iff$ there exists a truth valuation t such that $\Sigma^t = 1$. If no such t exists, the set of formulas Σ is unsatisfiable

Strategy for assignment 1

- Write down a truth table for all the formulas
- Σ is satisfiable $\iff$ there exists a row with $A^t = B^t = \cdots = D^t$ on that row
- Otherwise, it is not satisfiable

Tautologies and contradictions

- **Tautology**: a formula that is always true (under any truth valuation)

- **Contradiction**: a formula that is always false (under any truth valuation)
- **Contingent**: a formula that is neither a tautology or contradiction

Tautologies

- Law of the excluded middle: $p \vee \neg p$
  - This is an important tautology
- If A is a tautology that contains the prop. symbol p, p can be replaced with an arbitrary formula and A' will still be a tautology

Law of contradiction

- $\neg(p \wedge \neg p)$ is a tautology; $p \wedge \neg p$ is a contradiction

Three laws of thought (Plato)

1. Law of identity: $p = p$
2. Law of contradiction: $\neg(p \wedge \neg p)$
3. Law of the excluded middle: $p \vee \neg p$

Tautological Consequence

- Suppose $\Sigma \subseteq Form(\mathcal{L}^p)$ and $A \in Form(\mathcal{L}^p)$. Then, A is a **tautological consequence** of Σ *if and only if* we have that $\Sigma^t = 1 \implies A^t = 1$ for every truth valuation t
  - I.e. If Σ is satisfiable by t, then A is 1 (true)
  - This is written $\Sigma \vDash A$
    - This is not a formula; it is a statement in the **metalanguage** about Σ and A
  - Otherwise, we have $\Sigma \nvDash A$
  - We can say that the formulae in Σ tautologically (or just logically) imply the formula A

Special Case: $\emptyset$

- When Σ is the empty set, we get $\emptyset \vDash A$
- In this case, since there are no formulas (and because $\emptyset^t = 1$ is always vacuously true), $\emptyset \vDash A \implies A$ is a tautology

Fitting this in with previous theorems

- Let $\Sigma = A_1, A_2, \ldots, A_n \subseteq Form(\mathcal{L}^p)$ be a set of formulas (premises) and $C \in Form(\mathcal{L}^p)$ be a formula (conclusion). The following are equivalent
  - The argument with premises $A_1 \ldots A_n$ and conclusion C is valid

- $A_1 \wedge \cdots \wedge A_n \implies C$ is a tautology
- $A_1 \wedge \cdots \wedge A_n \implies \neg C$ is a contradiction
- The formula $A_1 \wedge \cdots \wedge A_n \wedge \neg C$ and the set $\{A_1 \ldots A_n, \neg C\}$ are not satisfiable
- C is a tautological consequence of Σ

Observation on valid arguments

- A conclusion C is only true if its arguments $A_1 \ldots A_n$ are all true and the conclusion C is valid

Tautological Equivalence

- $A \equiv B$ *if and only if* $A \vDash B \wedge B \vDash A$
- Tautologically equivalent formulae are assigned the same truth values by any valuation
- This doesn't guarantee that the formulae are all the same, just that the act the same way
- This is subtly different than $\iff$ (same for $\implies$)
  - $(A \equiv B) \iff (A \iff B)$ is a tautology
  - Same for a one direction implication

Proving tautological equivalence

Truth Tables

- To prove tautological equivalence, we may show that any truth valuation that satisfies Σ satisfies A using a truth table
- This can be used to prove that arguments are sound: equivalent to showing $\{A_1 \ldots A_n\} \vDash C$
- The invalidity of an argument can be proven by finding at least one row on a truth table where the premises are true but the conclusion false

Proof by contradiction

- Unfortunately, truth tables often get too large to use effectively, so we may wish to use another method
- Proof by contradiction: assume the opposite of what you are trying to prove and show that it leads to a contradiction (i.e. $p \vee \neg p$)

Proof by counterexample

- Finding a single example that contradicts what is trying to be proven means that it must be false

Important Tautological Equivalences

De Morgan's Laws

- $\neg(p \wedge q) \equiv (\neg p \vee \neg q)$
- $\neg(p \vee q) \equiv (\neg p \wedge \neg q)$

Contrapositive

- $p \implies q \equiv \neg q \implies \neg p$
- Note that the converse $q \implies p$ is not necessarily true; it is only the case if $p \iff q$

Biconditional

- AKA "If and only if", "iff"
- $p \iff q \equiv (p \implies q) \wedge (q \implies p)$
- This essentially checks whether p and q have the same value

Equivalences of connectives

- If $A \equiv A'$ and $B \equiv B'$
  - $\neg A \equiv \neg A'$
  - $A \wedge B \equiv A' \wedge B'$
  - etc.

Replaceability and Duality

- **Replaceability of tautologically equivalent formulas**: Let A be a formula that contains subformula B. Let $B \equiv C$ and let A' be the formula obtained by replacing B with C in A. Then $A \equiv A'$
  - This can be proven by structural induction
- **Duality**: Let A be a formula in $Form(\mathcal{L}^p)$ composed of atoms and the connectives $\neg$, $\wedge$, and $\vee$. Define $\Delta(A)$ as the formula found by replacing in A each atom with its negation, $\wedge$ with $\vee$, and $\vee$ with $\wedge$. Then $A \equiv \neg\Delta(A)$
  - This can also be proven by structural induction

Fuzzy Logic

- Instead of have truth values be 0 or 1, they can be anything in the range $[0, 1]$
  - We still have 1 meaning true and 0 meaning false
- We can redefine connectives as such

    - $p \wedge q$ becomes $\min\{p, q\}$
    - $p \vee q$ becomes $\max\{p, q\}$
    - $\neg p$ becomes $1 - p$
    - Note that these definitions still hold with our binary logic system (interesting!)
- In this system, the laws of the excluded middle and contradiction *do not hold*

Lecture 4 - Propositional Calculus

Propositional Calculus

- When we perform algebra normally, we manipulate equations with a set of rules and identities
  - Most of us experienced enough not to think about each step
- We can do the same thing with logical formulas as well
  - Instead of using algebraic identities, we use tautological equivalences
  - Ex. $(p \wedge q) \wedge \neg q \equiv p \wedge (q \wedge \neg q) \equiv p \wedge 0 \equiv 0$
    - Note that this expression is not in $Form(\mathcal{L}^p)$ since 0 and 1 are not part of this set

Removing $\implies$ and $\iff$

- These connectives are useful for expressing ideas but painful to manipulate
- Thus, we can use their definitions in terms of the other three connectives
  - $A \implies B \equiv \neg A \vee B$
  - $A \iff B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$

Essential Laws of propositional calculus

Name	Law
Excluded Middle	$A \vee \neg A \equiv 1$
Contradiction	$A \wedge \neg A \equiv 0$
Identity	$A \vee 0 \equiv A$, $A \wedge 1 \equiv A$
Domination	$A \vee 1 \equiv 1$, $A \wedge 0 \equiv 0$
Idempotent	$A \vee A \equiv A$, $A \wedge A \equiv A$
Double Negation	$\neg(\neg A) \equiv A$
Commutativity	$A \vee B \equiv B \vee A$, $A \wedge B \equiv B \wedge A$
Associativity	$(A \vee B) \vee C \equiv A \vee (B \vee C)$ $(A \wedge B) \wedge C \equiv A \wedge (B \wedge C)$
Distributivity	$A \vee (B \wedge C) \equiv (A \vee B) \wedge (A \vee C)$ $A \wedge (B \vee C) \equiv (A \wedge B) \vee (A \wedge C)$

Name	Law
De Morgan's	$\neg(A \wedge B) \equiv \neg A \vee \neg B$ $\neg(A \vee B) \equiv \neg A \wedge \neg B$

- These laws are used to simplify formulas
  - Should be used when possible
- These can all be proven using a truth table
- Each law has a dual pair where $\wedge$ and $\vee$ are switched and each atom is replaced with its negation
  - Fittingly, the double negation law is its own dual

Equivalents in regular algebra

- The laws of commutativity, associativity, and distributivity are also present in regular algebra
- In fact, many things seem to act the same way with respect to these laws
  - $\vee$ behaves the same way as $+$
  - $\wedge$ behaves the same way as $\times$
  - $\neg$ behaves the same way as $^-$

Further Laws

Name	Law
Absorption	$A \vee (A \wedge B) \equiv A$ $A \wedge (A \vee B) \equiv A$
Unamed	$(A \wedge B) \vee (\neg A \wedge B) \equiv B$ $(A \vee B) \wedge (\neg A \vee B) \equiv B$

Shortcuts for simplification

Literals

- Formulae in the form p or $\neg p$ are literals
- p and $\neg p$ are complimentary literals

Rules for literals

- If a conjunction contains complimentary literals or a 0, it always yields 0
  - All instances of 1 and duplicate literals can be ignored
- (dual) If a disjunction contains complimentary literals or a 1, it always yields 1

- All instances of 0 and duplicate literals can be ignored

Normal Forms

- Formulas can be manipulated into different **normal forms** that make them easier to compare and/or use for other things
- There are two types of normal forms:
  - **Disjunctive normal form**: A disjunction of conjunctive clauses as its disjuncts
    - Conjunctive clause: formula containing only conjunctions
    - Form: $(p_1 \wedge p_2 \wedge \ldots) \vee (q_1 \wedge q_2 \wedge \ldots) \wedge \ldots$
  - **Conjunctive normal form**: A conjunction of disjunctive clauses
    - Form: $(p_1 \vee p_2 \vee \ldots) \wedge (q_1 \vee q_2 \vee \ldots) \wedge \ldots$
- Atomic formulas are in both conjunctive and disjunctive normal form
  - Formulas that consist only of conjunctions or consist only of disjunctions are also in both normal forms

Obtaining normal forms

- Strategy: use tautological equivalences to manipulate the formula into one of the forms
- Algorithm for conjunctive normal form (CNF):
  1. Eliminate $\implies$ and $\iff$ using their definitions
  2. Use double negations and de morgan's laws to remove negations of expressions (the only negations should occur on individual atoms)
  3. Use the following recursive algorithm on the obtained formula A
     1. If A is a literal return A
     2. If A is $B \wedge C$ then return $CNF(B) \wedge CNF(C)$
     3. If A is $B \vee C$
        1. Determine $CNF(B)$ and $CNF(C)$
        2. Suppose that each of this are conjunctive clauses
        3. Return the result of distributing the disjunction (i.e. $\wedge_{i=1\ldots n, j=1\ldots m}(B_i \vee C_j)$). This is similar to expanding $(n_1 + \cdots + x_n) \times (y_1 + \cdots + y_m)$
  - The dual can be used to find DNF
- We can also find each form by reading off of a truth table
  - Disjunctive: disjunction of all the rows in the table where the formula is true
  - Conjunctive: take the dual of the disjunction of all the rows where the formula is false

Existence of normal forms

- Theorem: Any formula $A \in Form(\mathcal{L}^p)$ is tautologically equivalent to some formula in DNF

- Proof
  - If A is a contradiction, it is tautologically equivalent to $p \wedge \neg p$, which is in DNF
  - If A is not a contradiction
    - There are some valuations under which A is true
    - Each of these valuations becomes a conjunctive clause (hand-waving here but you get the idea)
    - By the definition of $\wedge$, the disjunction of these clauses is equivalent to A
  - We can also say that if A is a tautology, then it is equivalent to $p \vee \neg p$ (this case if covered by the previous one)
- Theorem: Any formula in $A \in Form(\mathcal{L}^p)$ is equivalent to some formula in CNF
  - Taking the dual of the CNF gives the DNF for the negation of the expression
    - We can find it by taking the dual of the DNF of $\neg A$ (i.e. disjunction of all the rows that are false)
  - Since we know DNF exists, CNF must also exist

Lecture 5 - Adequate Connective Circuits

Connectives

- Since $A \implies B \equiv \neg A \land B$, the connective $\implies$ is **definable** in terms of $\neg$ and $\land$
  - $\lor$ is definable in terms of $\neg$ and $\implies$ since $A \lor B \equiv \neg A \implies B$, etc
- There are four possible unary connectives and sixteen possible binary connectives. We can also define n-ary connectives (ex: if-then-else block)
  - These are essentially functions, and are notate $f(A_1, \dots A_n)$
  - Four unary connectives: always true ($\top$), always false ($\bot$), no change ( ), negation ($\neg$)
  - There will be 2^{2^n} n-ary connectives since with n symbols, the truth table has 2^n rows and 2^n possible outputs per row
- Connectives are defined by their truth table

Some more common connectives

- There exist some more common connectives that we haven't seen yet
- **NAND**: $\neg(p \land q)$ (short for not and)
- **NOR**: $\neg(p \lor q)$ (short for not or)
- **XOR**: $\neg(p \iff q)$ (short for *exclusive or*)

Adequate sets of connectives

- A set of connectives is **adequate** if it can express any truth table
- The set of 5 standard connectives $\{\neg, \land, \lor, \implies, \iff\}$ is adequate
  - Note that since $\implies$ and $\iff$ can be defined in terms of the remaining three, the set $\{\neg, \land, \lor\}$ is also adequate
    - This also follows from the theorems proving the existence of CNF and DNF for any formula

Proving adequacy

- We can prove adequacy by showing that each connective in the set $\{\neg, \land, \lor\}$ can be constructed using connectives in our set $\mathcal{S}$
  - I.e. we are creating a tautological equivalence between some formula $A_{\mathcal{S}}$ in $\mathcal{S}$ and some formula A_{S_0} where $S_0 = \{\neg, \land, \lor\}$ and citing the replaceability theorem
- Ex. Since $\neg(\neg A \land \neg B) \equiv A \lor B$, the set $\{\neg, \land\}$ is adequate
  - By duality, the set $\{\neg, \lor\}$ is also adequate

Some more common connectives

- There exist some more common connectives that we haven't seen yet

Name	Notation	Logical definition	Circuit name
Peirce Arrow	$p \downarrow q$	$\neg(p \vee q)$	NOR
Sheffer Stroke	$p \mid q$	$\neg(p \wedge q)$	NAND
Exclusive Or	$p \oplus q$	$\neg(p \iff q)$	XOR

- Note that the Pierce Arrow and Sheffer Stroke are adequate by themselves
- Proof of Pierce Arrow adequacy
  - $\neg p \equiv p \mid p$
  - $p \wedge q \equiv (p \mid q) \mid (p \mid q)$
  - $p \vee q \equiv (p \mid p) \mid (q \mid q)$
- Thus, $\{\mid\}$ is adequate

Proving inadequacy

- We must prove that the set $\{\neg, \wedge, \vee\}$ cannot be defined using the set of connective we wish to prove inadequate

A ternary connective

- Define τ as the ternary connective with the following truth table

p	q	r	$\tau(p,q,r)$
1	1	1	1
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	0

- Here, $\tau(p,q,r)^t$ is the same as q^t if $p^t = 1$, and equals r^t if $p^t = 0$
- This is the if-then-else statement

- There are $2^{2^3} = 256$ possible ternary connectives

Boolean Algebra

- A **Boolean algebra** is a set B together with two binary operations $+$ and $\times$, and the unary operation $^-$. B is the set $\{0, 1\}$ and is closed under the application of the three operations. The following laws must hold
  - Identity laws: $x + 0 = x$ and $x \times 1 = x$
  - Complement laws: $x + \bar{x} = 1$ and $x \times \bar{x} = 0$
  - Associativity laws: $(x + y) + z = x + (y + z)$ and $(x \times y) \times z = x \times (y \times z)$
  - Commutativity laws: $x + y = y + x$, $x \times y = y \times x$
  - Distributivity laws: $x + (y \times z) = (x + y) \times (x + z)$ and $x \times (y + z) = (x \times y) + (x \times z)$
- Any "rule" or "law" in one boolean algebra has an equivalent in any other one
- Boolean algebra is used to model computer circuitry
  - Inputs and outputs are in the set $\{0, 1\}$
  - Individual circuits can implement a boolean function $f : \{0, 1\}^n \to \{0, 1\}$
  - The building blocks of circuits are **logic gates**, which act as the boolean operators

Examples of boolean algebras

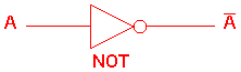
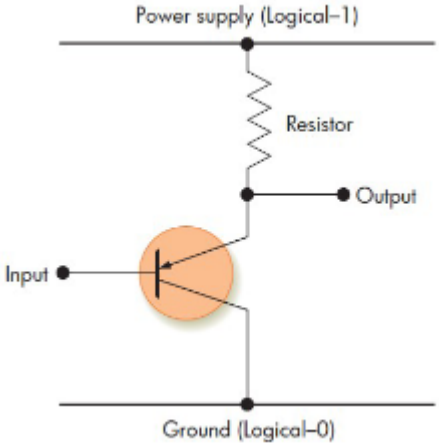
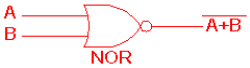
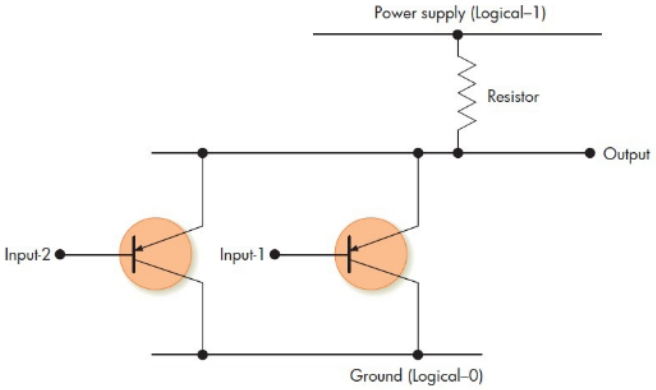
- The set of formulas in $Form(\mathcal{L}^p)$ that use only the connectives $\{\neg, \wedge, \vee\}$ is a Boolean algebra (we are treating $=$ as $\equiv$) is a Boolean algebra
- The set of subsets of the universal set U with the union operator $\cup$, the intersection operator $\cap$, the set complement operator ^{c}, the empty set $\emptyset$ and the universal set U is a Boolean algebra (woah)

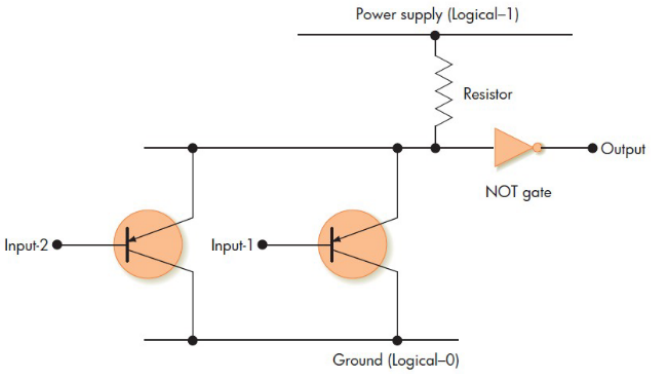
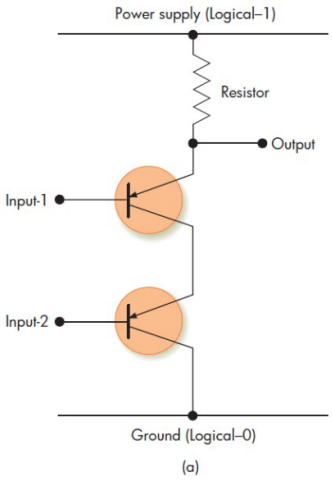
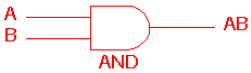
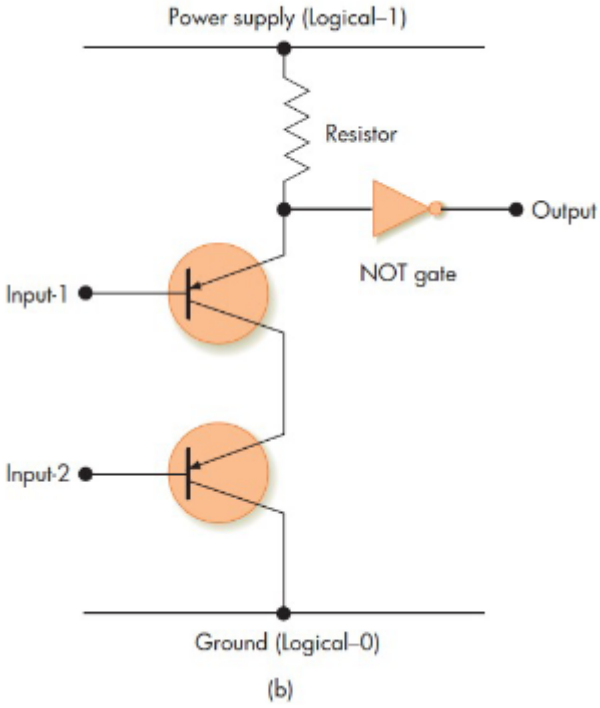
Transistors

- These are the physical implementation of logic gates
- Transistors are switches that may or may not let electricity flow
- Each transistor has an two inputs lines and an output line
  - The output line can either be in an "on" or "off" state, which is equivalent to the binary 0 and 1
  - The first input is the control line, and is used to control the switch
- If a significantly high voltage is applied to the control line, the switch closes and enters the ON state
  - Here, voltage coming through the input goes directly out the output (which can be measured)
- Although transistors are convenient due to their tiny size, any *bistable device* can be used to form a binary computer. Such a device must

    - Have two energy states (0 or 1) separated by a large barrier (so they are not switched by accident)
    - Have a way to sense its own state (i.e. whether it is a 0 or 1)
    - Be able to switch from a 0 to a 1 if enough energy is applied
- Based on these rules, the light switch can be used to create a binary computer (albeit a really slow one)
- It is likely that current transistors will be replaced by better technology that performs the same function

Basic Logic Gates

Gate	Expression	Diagram	Transistor diagram
NOT	$\neg p$		
NOR	$\neg(p \vee q)$		

Gate	Expression	Diagram	Transistor diagram
OR	$p \vee q$		
NAND	$\neg(p \wedge q)$		
AND	$p \wedge q$		

- The NOR gate (and all the gates) uses a resistor

    - It implements negation by using the switch to open the circuit to ground (logical 0)
- The NOR gate is created by wiring two transistors in parallel such that current travelling through either one travels to the output
- The OR gate is created by negating the NOR gate with a NOT gate
- The NAND gate is created by wiring two transistors in parallel so that the current must travel through both in order to set an output as 1
- The AND gate is created by negating the NAND gate

Non-standard gate example: Toffoli gate

x_1	x_2	x_3		y_1	y_2	y_3
1	1	1		1	1	0
1	1	0		1	1	1
1	0	1		1	0	1
1	0	0		1	0	0
0	1	1		0	1	1
0	1	0		0	1	0
0	0	1		0	0	1
0	0	0		0	0	0

- Has a 3-bit input and 3-bit output
- If the first two bits are 1, the second bit is inverted. Otherwise, the output is the same as the input
- It is *universal* and *reversible*
    - Universal: can be used to implement any other logic gate
    - Reversible: one-to-one correspondence between inputs and outputs (if we know the output, we can deduce the input)
- Can be implemented using five 2-bit quantum gates

Circuit notation and conventions

- $x + y$ denotes $x \vee y$
- $x \times y$ and xy denote $x \wedge y$
- $\bar{x}$ denotes $\neg x$
- $=$ denotes $\equiv$
- AND and OR gates may have multiple inputs since they are n-ary inputs

Combinatorial circuits

- Any memoryless circuit where the output(s) is a function of the inputs only
- This can be implemented using NOT, OR, and AND gates
- These are in contrast to sequential logic circuits, which have memory components
- Combinatorial circuits can be created from a boolean algebra expression by following the tree structure of the expression
- Sometimes, multiple logic gates may use the same inputs (i.e. a variable appears multiple times in an expression). There are two options to depict this
  - Each input is drawn multiple times
  - Each input is drawn once and branches into multiple logic gates
- Propositional calculus can be used to simply expressions, which create simpler and less expensive circuits

Adders

- A useful task that can be completed with circuits is adding numbers
- First, we create a half-adder that adds two bits
  - It has two output bits: a sum bit and a carry bit
  - The carry bit is essentially the "2-place" of the sum bit
- Truth table

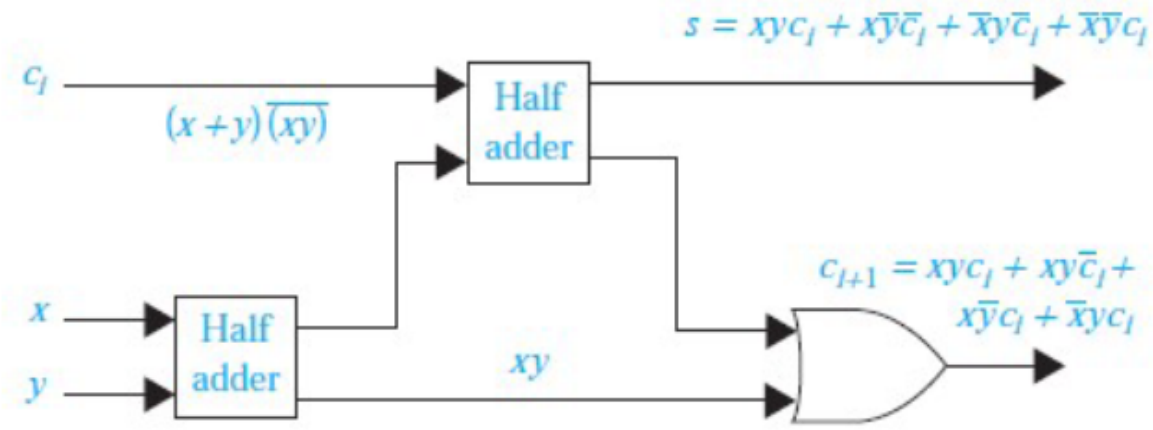
x	y	sum	carry
1	1	0	1
1	0	1	0
0	1	1	0
0	0	0	0

- We see that $sum = x\bar{y} + \bar{x}y \equiv (x + y)\bar{xy}$ and $carry = xy$
  - We can draw a circuit from this equation

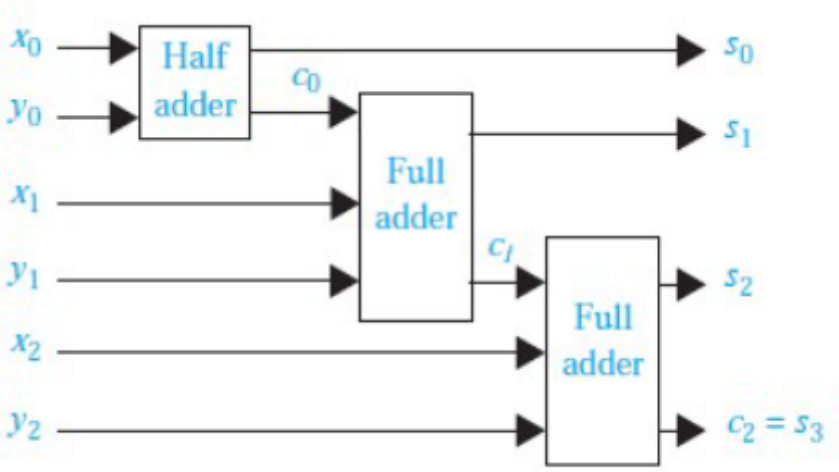
- We use half-adders to create a full adder with the following truth table:

Input			Output	
x	y	c_i	s	c_{i+1}
1	1	1	1	1
1	1	0	0	1
1	0	1	0	1
1	0	0	1	0
0	1	1	0	1
0	1	0	1	0
0	0	1	1	0
0	0	0	0	0

- We get $s = xyc_i + x\bar{y}c_i + \bar{x}ybarc_i + \bar{x}\bar{y}c_i$ and $c_{i+1} = xyc_i + xy\bar{c_i} + x\bar{y}c_i + \bar{x}yc_i$
- We get the following circuit



- We can keep chaining adders together in similar ways to get a three-bit adder, etc



Minimizing Circuits

- The less operators that are present in an logical expression, the less gates -> transistors need to be used to implement the circuit
- Thus, we can use the laws of propositional logic to minimize circuits and make them more efficient
- We can also use these techniques to simplify code, specifically code that uses nested if statements
  - We can determine the conditions for the code in each block to be run
  - Stuff like this is important for optimizing compilers

Lecture 6 - Formal Deducibility

Formal Deducibility

- So far, we have used *semantic methods* for proofs (i.e. truth tables, tautological consequence)
- We now want to find a set of formal rules: a **syntactic method**
  - This way, we can construct proofs mechanically
  - Formal here means that we will not pay attention to the semantics of what we write, just the syntax
- $A \vdash B$ means that B is formally deducible from A
  - Similar to $\models$, but proof method is very different

Conventions

- Σ refers to a set of formulas
- $\Sigma \vdash A$ means that A is formally deducible from Σ
- $\Sigma \cup \{A\}$ can be written as Σ, A
- $\Sigma \cup \Sigma'$ can be written as Σ, Σ' where Σ and Σ' are sets of formulas
- $A \dashv\vdash B \equiv A \vdash B \wedge B \vdash A$

11 rules of formal deduction

Name	Abbr.	Rule
Reflexivity	Ref	$A \vdash A$ is a theorem
Addition of premises	$+$	If $\Sigma \vdash A$ is a theorem, then $\Sigma, \Sigma' \vdash A$ is also a theorem
$\neg$ elimination	$\neg -$	If $\Sigma, \neg A \vdash B$ is a theorem and $\Sigma, \neg A \vdash \neg B$ is also a theorem, then $\Sigma \vdash A$ is a theorem
$\implies$ elimination	$\implies -$	If $\Sigma \vdash A \implies B$ is a theorem and $\Sigma \vdash A$ is a theorem, then $\Sigma \vdash B$ is a theorem
$\implies$ introduction	$\implies +$	If $\Sigma, A \vdash B$ is a theorem, then $\Sigma \vdash A \implies B$ is a theorem
$\wedge$ elimination	$\wedge -$	If $\Sigma \vdash A \wedge B$ is a theorem and $\Sigma \vdash A$ is a theorem, then so is $\Sigma \vdash B$

Name	Abbr.	Rule
$\wedge$ introduction	$\wedge+$	If $\Sigma \vdash A$ is a theorem and $\Sigma \vdash B$ is a theorem, then $\Sigma \vdash A \wedge B$ is a theorem
$\vee$ elimination	$\vee-$	If $\Sigma, A \vdash C$ is a theorem and $\Sigma, B \vdash C$ is a theorem, then $\Sigma, A \vee B \vdash C$ is a theorem
$\vee$ introduction	$\vee+$	If $\Sigma \vdash A$ is a theorem, then $\Sigma \vdash A \vee B$ and $\Sigma \vdash B \vee A$ are theorems
$\iff$ elimination	$\iff -$	If $\Sigma \vdash A \iff B$ and $\Sigma \vdash A$ are theorems, then so is $\Sigma \vdash B$ (same with A and B switched)
$\iff$ introduction	$\iff +$	If $\Sigma, A \vdash B$ and $\Sigma, B \vdash A$ are theorems, then so is $\Sigma \vdash A \iff B$

- These rules are really just templates (**schemes**) for creating families of rules

Proof example

- Prove the membership rule: If $A \in \Sigma$, then $\Sigma \vdash A$
  - Suppose $A \in \Sigma$ and $\Sigma' = \Sigma - \{A\}$. We have $A \vdash A$ by reference and so $A, \Sigma' \vdash A$ by addition of premises
- Here, we must act in individual steps and cite the rule that we use. This is formatted as a numbered list of steps with the rule written to the right
  - We might choose to cite previous steps on the right as well
- We have proven a new theorem: $\in$

Name	Abbr.	Rule
Membership rule	$\in$	If $A \in \Sigma$, then $\Sigma \vdash A$

- We can invoke **proved theorems** without justification since we could just as easily insert their proof in the proof we are writing
- A demonstrated $\Sigma \vdash A$ is called a **theorem** or a **scheme of formal deducibility**
- We can get a computer to check these proofs since it can check if we used the rules correctly

- We also have the following formal deduction rules (which must be proven to use)

1. $A \vdash \neg\neg A$ ("half of" double negation)
2. $A \vee B, \neg A \vdash B$ (disjunctive syllogism)
3. $\neg B, A \to B \vdash \neg A$ (modus tollens)
4. $A \to B \dashv\vdash \neg B \to \neg A$ (contrapositive)
5. If $A \vdash B$ then $\neg B \vdash \neg A$ ("flip-flop")
6. $\neg(A \vee B) \dashv\vdash \neg A \wedge \neg B$ (De Morgan)
7. $\neg(A \wedge B) \dashv\vdash \neg A \vee \neg B$ (De Morgan)
8. $A \to B \dashv\vdash \neg A \vee B$ (implication rule)
9. $\emptyset \vdash \neg(A \wedge \neg A)$ (non-contradiction)
10. $\emptyset \vdash A \vee \neg A$ (excluded middle)
11. $A \to B, \neg A \to B \vdash B$ (rule of cases)
12. $A, \neg A \vdash B$ (inconsistency rule)

Intuitive meanings of rules

$\neg$ elimination

- Expresses proof by contradiction since we derive a contradiction from deriving both B and $\neg B$ from $\neg A$
  - Therefore, A must be true (or rather, $\Sigma \vdash A$ a theorem)

$\vee$ elimination

- Recap: If $\Sigma, A \vdash C$ is a theorem and $\Sigma, B \vdash C$ is a theorem, then $\Sigma, A \vee B \vdash C$ is a theorem
- Expresses the idea of proof by cases
- Here, the premises of the cases are Σ, A and Σ, B
- If we reach the same conclusion in both cases (in this case, C), then we can conclude that $\Sigma, A \vee B \vdash C$

$\Longrightarrow$ introduction

- Recap: If $\Sigma, A \vdash B$ then $\Sigma \vdash A \implies B$
- Here, if we have $\Sigma \vdash A$, using Σ vs. Σ, A as premises will have the same effect
- Therefore, if $\Sigma, A \vdash B$, we must also have $\Sigma \vdash B$, and since we have A as well, $\Sigma \vdash A \implies B$ must also be true

Formal Deducibility complete definition

- A **formal system** is specified by a set of deduction rules
- A formula A is **formally deducible** from Σ (written $\Sigma \vdash A$) if and only if we can generated $\Sigma \vdash A$ by a finite number of applications of the rules of formal deduction
- This sequence of rule applications is called a **formal proof**
- To check a sequence of steps is actually a formal proof, we must check if the rules are applied correctly at each step and if the last term is what we wish to prove
- Although it may be difficult to find a proof, it is significant that proofs can be checked mechanically

Finding proofs

- We can check proofs, but how do we find them?
- One strategy: **working backwards**
  - For our last line, what rules can be used to generate that line?
  - We can find a statement that can be used to prove the last line using one rule application
  - Next, we must just prove that
  - This moves us closer and closer to our goal each time
- A line in a proof can be used several times
- A line can be
  1. On its own (ex. $A \vdash A$)
  2. Be a previously proved theorem
  3. Be based on previous line(s) (not necessarily the the one preceding it directly) by using
     - A rule of formal deduction
     - A proved theorem
- If the conclusion is an implication, use $\implies +$ to add premise of the implication to the set of premises Σ
  - I.e. $\Sigma, A \vdash B$
- If one of the premises is a disjunction, use $\vee -$ to split into cases and prove separately
- If a direct proof is difficult, try proving the contrapositive instead, then use the flip-flop theorem (which you must prove)

- If all else fails, try a proof by contradiction using $\neg -$
- We may wish to start with a set of premises that is not Σ
  - I.e. when doing a proof by contradiction
  - We may wish to "undo" these modifications at some point in the proof so that we can take advantage of theorems proven as part of the proof

Tautological consequence vs. deducibility

- Tautological consequence ($\Sigma \vDash A$) and formal deducibility ($\Sigma \vdash A$) are entirely different (but related) things
  - Tautological consequence: semantics
  - Formal deducibility: syntax
- Both of these are forms of metalanguage: they are not parts of formulas themselves
  - I.e. $\vDash$ and $\vdash$ are not symbols in $\mathcal{L}^p$
  - They should not be confused with $\implies$
    - $A \vDash B$ iff $A \implies B$ is a tautology
    - $A \vdash B$ iff $\emptyset \vdash A \implies B$

Proving statements about formal deduction

- Such statements can be proven recursively
  - Base case: reference rule ($A \vdash A$)
  - Recursive cases: other 10 rules (similar to $Form(\mathcal{L}^p)$ proof)

Some properties of formal deduction

Finiteness of premise set

- If $\Sigma \vdash A$, then there exists a finite $\Sigma^0 \subseteq \Sigma$ such that $\Sigma^0 \vdash A$
- This can be proven inductively/recursively
  - 10 cases, one for each recursive rule
- Intuition: since a proof for $\Sigma \vdash A$ has finitely many steps, we only need finitely many formulas from the premise set

Transitivity of deducibility

- Let $\Sigma, \Sigma' \subseteq Form(\mathcal{L}^p)$. If $\Sigma \vdash \Sigma'$ and $\Sigma' \vdash A$, then $\Sigma \vdash A$

Hypothetical Syllogism

Name	Abbr.	Rule
Hypothetical Syllogism	hyp	$A \implies B, B \implies C \vdash A \implies C$

Double Negation

- $\neg\neg A \vdash A$
- Proof:
  - $\neg\neg A, \neg A \vdash \neg A$
  - $\neg\neg A, \neg A \vdash \neg\neg A$
  - $\neg\neg A \vdash A$

Reductio ad absurdum

- If $\Sigma, A \vdash B$ and $\Sigma, A \vdash \neg B$, then $\Sigma \vdash \neg A$
- This holds when Σ is infinite as well, but the proof must invoke the finiteness of premises theorem
- This is sometimes denoted $\neg+$ since both it and $\neg-$ formalize the idea of proof by contradiction
  - $\neg-$ is stronger than $\neg+$
  - However, if $\neg-$ is replaced by $\neg+$ in the ruleset, $\neg-$ cannot be proven

Name	Abbr.	Rule
$\neg$ Introduction	$\neg+$	If $\Sigma, A \vdash B$ and $\Sigma, A \vdash \neg B$, then $\Sigma \vdash \neg A$

Syntactic Equivalence

- $A \vdash\dashv B$ denotes $A \vdash B \wedge B \vdash A$
- If we have $A \vdash\dashv A'$ and $B \vdash\dashv B'$, then
  - $\neg A \vdash\dashv \neg A'$
  - $A \wedge B \vdash\dashv A' \wedge B'$
  - etc.
- This is similar to tautological equivalence $\equiv$

Replaceability of syntactically equivalent formulas

- If $B \vdash\dashv C$, then we have $A \vdash\dashv A'$ when replacing any instances of B with C in A to get A'
- $A_1, A_2, \ldots A_n \vdash A$ iff $\emptyset \vdash A_1 \wedge \cdots \wedge A_n \implies A$
- $\emptyset \vdash A_1 \wedge \cdots \wedge A_n \implies A$ iff $\emptyset \vdash A_1 \implies (\ldots (A_n \implies A) \ldots)$
- Special case: $\Sigma = \emptyset$

- If $\emptyset \vdash A$, then A is **formally provable** (full stop)
- Example: the laws of non-contradiction $\neg(A \wedge \neg A)$ and the excluded middle $(A \wedge \neg A)$ are formally provable

Why prove?

- Tautological consequence corresponds to **informal deducibility** and involves semantics
- Formal deducibility is concerned with syntax
- We wish to a define a system where *we can formally prove everything that is semantically correct*

What makes a system of formal deduction good?

- It must not be able to formally prove incorrect statements (**soundness**)
- It should be able to formally prove every correct statement (**completeness**)

Soundness of a formal deduction system

- A system of formal deducibility, denoted $\vdash_*$ is defined by a list of formal deduction rules
- Soundness: "If $\Sigma \vdash_* A$ then $\Sigma \vDash A$" is true for any Σ and A
  - This means that what can be proven with $\vdash_*$ also holds in informal reasoning
  - I.e. we cannot prove incorrect statements (it is sound)
- Soundness theorem: If $\Sigma \vdash A$ then $\Sigma \vDash A$, where $\vDash$ refers to the formal deduction made with the 11 given rules
  - This is proven inductively by replacing $\vdash$ with $\vDash$ in each case (for each of the 11 rules), then proving the statement

Completeness of a formal deduction system

- Completeness: "If $\Sigma \vDash A$ then $\Sigma \vdash_* A$" is true for any Σ and A
  - I.e. whatever is correct can be formally proven
- Completeness theorem: if $\Sigma \vDash A$ then $\Sigma \vdash A$ where $\vdash$ means formal deduction based on the 11 given rules
- Proof steps
  1. If $A_1 \dots A_n \vDash A$ then $\emptyset \vDash (A_1 \implies \dots (A_n \implies A) \dots)$
  2. If $\emptyset \vDash A$ then $\emptyset \vdash A$ (every tautology has a formal proof)
  3. If $\emptyset \vdash (A_1 \implies \dots (A_n \implies A) \dots)$ then $A_1 \dots A_n \vdash A$

Connection between syntax and semantics

- The Soundness and Completeness Theorems associate the syntactic notion of formal deduction, based on the 11 rules, with the semantic notion of tautological consequence, and establish the equivalence between them.

Proving Invalidity and Logical Fallacies

- Formal deduction cannot be used to prove that an argument is invalid

Fallacy of denying the antecedent

- If $T \implies A$ and $\neg T$, then $\neg A$
  - A can still be true; if $\neg T$, we simply don't know anything about A
- This is the error behind assuming the converse of an implication to be true

Fallacy of affirming the consequent

- If $P \implies \neg Q$ and $\neg Q$, then P
  - We don't actually know anything about P
  - We have assumed the converse to be true, which it may not
- Summarized by the following joke:

> “Why are you standing on this street corner, waiving your hands?”
> “I am keeping away the elephants.”
> “But there aren’t any elephants here.”
> “You bet: that’s because I’m here.”

Consistency and Satisfiability

- A set of formulas Σ is **consistent** if there is no formula F such that $\Sigma \vdash F$ and $\Sigma \vdash \neg F$
- Otherwise, it is **inconsistent**
- Lemma: *A set Σ of formulas is satisfiable if and only if Σ is consistent*
  - Proof is by contradiction: assume that Σ is inconsistent (for $\rightarrow$) and and not satisfiable (for $\leftarrow$)

Lecture 7 - Resolution Proof Systems

Motivation for resolutions

- We have already learned about formal deduction; why do we need another proof system?
- Formal deduction appeals to *intuition*
  - This is good when people are the ones writing proofs because they can draw from their experience
  - However, we cannot write algorithms for finding proofs, so they cannot be generated mechanically
- On the other hand, the **resolution proof system** is less intuitive, but can be much more easily automated
- Resolution is used in industry

Resolution Rule

- Starting assumption: Out starting formulas A and B are both written as some disjunction of literals (ex. $A = p \vee \neg q \vee \neg r$)
- To apply the resolution rule to A and B, there must exist a propositional symbol that occurs in A and whose negation occurs in B (or vice versa)
  - So, $A = p \vee C$ and $B = \neg p \vee D$ for some p, C, D
- Then, the **resolvent** of A and B is $C \vee D$
  - *The leftovers after p are resolved with $\neg p$*
  - $\{A, B\}$ is satisfiable $\iff$ $C \vee D$ is satisfiable
- Explanation
  - If we are trying to determine if $\{C \vee p, D \vee \neg p\}$ is satisfiable, one of p and $\neg p$ will *always* be false, so whether it is satisfiable comes down to $C \vee D$, since only these remain

Determining if a set of clauses is satisfiable

- Recall that $\Sigma \vDash A$ if and only if $\Sigma \cup \{\neg A\}$ is *unsatisfiable*
- So, if resolution can determine the satisfiability of a set of formulas, then it can also answer questions about tautological consequence
  - I.e. argument validity

Two outcomes of a resolution proof

- We resolve everything we can, arriving at the empty clause $\{\}$
  - This *is not* satisfiable, so the argument must be *valid*
  - Why is this unsatisfiable?
    - In a disjunction, at least one member must be true to make the disjunction true
    - By definition, an empty disjunction $\{\}$ has no true items, so it cannot be true
- We resolve everything we can, arriving at the empty set $\emptyset$
  - This *is* satisfiable (by definition), so the argument must be *invalid*

Soundness of resolution formal deduction

- Theorem: the resolvent is a tautological implication of its parent clauses, which makes it a sound rule of formal deduction
- Proof: we essentially need to prove $\{p \vee A, \neg p \vee B\} \vDash A \vee B$ if we know that $\{p \vee A, \neg p \vee B\} \vdash A \vee B$
  - Cases: A and B are both empty, at least one is not empty
  - Proof is fairly trivial

Set of support strategy

- Motivation: if we don't know what to resolve when, we may keep going around in circles, unable to find a proof
- Strategy: **Set of support** is the set of formulas obtained, in some number of steps, *from the negated conclusion*
- Every resulting step must use at least on formula from the set of support
- Example:
  - Prove $\{(s \wedge h) \implies p, s, \neg p\} \vDash \neg h$
  - Convert first element into CNF: $\neg s \vee \neg h \vee p$
  - So, we can use the following set for a resolution: $\{\neg s \vee \neg h \vee p, s, \neg p\}$
  - We can start deriving the set of support. Every time we derive a new statement, it becomes part of the base of support
    - h from the negated conclusion
    - $\neg s \vee p$ from $\neg s \vee \neg h \vee p$ and h
    - $\neg s$ from $\neg p$ and $\neg s \vee p$
    - $\{\}$ from s and $\neg s$
  - Since we obtained $\{\}$, the set is unsatisfiable
  - In general, we would write out the premises and set of support in a numbered list and refer to items by their number instead
    1. $\neg s \vee \neg h \vee p$
    2. s

3. $\neg p$
4. h
5. $\neg s \vee p$
6. $\neg s$
7. $\{\}$

Davis-Putnam Procedure

Clauses as sets

- Every clause corresponds to a set of literals, i.e. those in the clause
  - $p \wedge q \wedge r$ corresponds to $\{p, q, r\}$, etc.
  - We can use a set because having duplicate terms in a clause will have no effect on it
- For these reasons, we can refer to clauses as sets, and use set notation in addition to logical notation
  - So, the resolvent of $C \cup \{p\}$ and $D \cup \{\neg p\}$ is $[(C \cup p) \cup (D \cup \neg p)]$ $p, \neg p$

Davis-Putnam Procedure

- Given a non-empty set of clauses using the variables $p_1 \ldots p_n$, the DPP repeats until there are no variables left
  1. Remove all clauses that contain both p and $\neg p$, since a disjunction involving these is always true and will thus never lead to a contradiction
  2. Choose a variable p_k that appears in one of the clauses
  3. Add all the possible resolvents using resolution on p to the set of clauses
  4. Discard any clauses that contain p (or $\neg p$)
  5. Discard duplicate clauses
- This is known as *eliminating* p
- Stopping conditions
  - If a step resolves $\{p\}$ and $\{\neg p\}$, then the empty clause $\{\}$ will be obtained, which ends the procedure and proves that the argument is **valid**
    - I.e. it shows that the argument with its negated conclusion is not satisfiable -> never corrected
  - If there is never a step that resolves $\{p\}$ and $\{\neg p\}$, then all the clauses will end up getting discarded and the empty set $\emptyset$ will be produced. This means the argument is **invalid**
    - I.e. no contradiction was found when the negated conclusion was included, so the argument is invalid

Example

- Apply the DPP to the set of clauses $\{\{\neg p, q\}, \{\neg p, \neg r, s\}, \{p\}, \{r\}, \{\neg s\}\}$
  - Eliminating p gives $\{\{q\}, \{\neg q, \neg r, s\}, \{r\}, \{\neg s\}\}$
  - Eliminating q gives $\{\{\neg r, s\}, \{r\}, \{\neg s\}\}$
  - Eliminating r gives $\{\{s\}, \{\neg s\}\}$
  - Eliminating s gives $\{\}$
- Since the output is an empty clause, the argument must be valid

Notational Conventions

- If the set of clauses is large or complex, we can give each clause an identifier $\mathcal{T}_i$ to keep track of it
- Then, when we produce the resolvents in the next step ($\mathcal{U}_i$), we list the identifiers that led to each resolvent instead of the clauses themselves

Soundness and Completeness of DPP

- Let $\mathcal{S}$ be a finite set of clauses
- $\mathcal{S}$ is not satisfiable $\iff$ the output of DPP on $\mathcal{S}$ is the empty clause $\{\}$
- Proof idea
  - Resolution propagates satisfiability forwards: if the parent clauses are satisfiable, then the resolvents will be as well
  - This also happens in reverse
- $\mathcal{S} \vdash \{\}$ by DPP $\iff$ $\mathcal{S}$ is not satisfiable
- Proof idea
  - Somewhere, there must be a resolution derivation from $\mathcal{S}$ to $\{\}$
  - Thus, by soundness, $\mathcal{S}$ must not be satisfiable
  - The other direction is provable by contradiction

Proving Invalidity with DPP

- Example: Affirming the consequent (assuming the converse is true)
  - If it's raining, then the streets are wet. The streets are current wet. Therefore, it must be raining.
- Set of clauses: $\{\{\neg r, w\}, \{w\}, \{\neg r\}\}$

Lecture 10 - First Order Logic

- Alternative names: *predicate logic, predicate calculus, elementary logic, restricted predicate calculus, restricted functional calculus, relational calculus, theory of quantification, theory of quantification with equality*
- In propositional logic, a proposition is either true or false
  - Because of this, there are some perfectly logical arguments that cannot be expressed in propositional logic

Example of argument inexpressible with propositional logic

1. All humans are mortal
2. Socrates is a human
3. Therefore, Socrates is mortal

- Here, each of these arguments would be expressed as a single propositional variable. It would look something like
  1. p
  2. q
  3. therefore, r
- This is obviously not allowed
  - All humans are mortal has to be written as an implication (humanity implies morality) for this to work with predicate logic
- We need a way to *associate individuals and their properties*
  - This is the objective of **first order logic**, which extends propositional logic

Elements of first order logic

- Usually used to explain mathematical theories; elements are geared towards this

Unique to FOL

- A domain of objects (individuals)
  - Ex. $\mathbb{N}$
- Designated individuals
  - Ex. 0
  - Remember, 0 and 1 are not technically part of predicate logic
- Relations

    - Ex. $=, \geq$
- Functions
    - Ex. $+$, $\times$, successor function

Also present in predicate logic

- Variables that are part of the domain (propositional variables)
- Logical connectives ($\neg$, $\wedge$, $\vee$, $\implies$, $\iff$)
- Quantifiers ($\forall$, $\exists$)
    - This is usually stated verbally, but can be symbolic as well
- Punctuation

Domain

- Consider the propositional statement *Joan is Paul's Mother*
    - It is ambiguous because we can only know if it is true with context, as well as because there are multiple people named Joan and Paul, etc.
- For this statement not to be ambiguous, we need to define a **domain**
- In this example, the domain may be any group of people that includes these two (and presumably doesn't include others with the same name)
- Often, the domain is mathematical in nature (I.e. a set)
- *The truth of a statement may depend on its domain*
    - Ex. The statement "there is a smallest number" is true of set domain $\{1, 2, 3, 4, 5\}$ but not of $\mathbb{R}$

Individuals

- **Individual**s are members of a domain
- To avoid trivial cases, it is assumed that a domain contains at least one individual
- Individuals are sometimes called **objects**
- To refer to a specific individual, identifiers must be used
    - These are called **individual symbols**
    - Ex. to refer to a natural number, the digits 1-9 must be used

Relations

- **Relations** make statements about individuals
    - These can also be called predicates, extending their definition
    - Ex. *Mary and Paul are siblings*
    - Ex. *The sum of 2 and 3 is 5*

- In each relation, there is a list of individuals, called the **argument list**, and something relating them (the relation)
  - From the first example
    - Argument list: *Mary and Paul*
    - Relation: *are siblings*

Notation

```
mother(Joan, Mary)
m(J, M)
```

- Each relation is given a name, which is followed by the argument list
- The argument list is enclosed in parentheses
- Single letters are often used for arguments and relations
- The order of the arguments is important
- In aggregate, each relation seems to look and behave like a function that takes parameters

Arity

- **Arity**: the number of elements in the argument list
- The arity of a relation is fixed
  - Therefore, two relations must be different if they have different arities
- A relation with arity n is an n-ary relation symbol
- A one-place (1-ary) relation is called a **property**
  - I.e. `Human(Socrates)` and `Mortal(Socrates)`

Variables

- Sometimes, we don't want to give properties to a particular individual, but a whole set of them
- To do this, we can use a **variable**
  - Ex. `Human(u)` denotes `u is human`
  - Ex. `Sum(a, b, c)` denotes `the sum of a and b is c`

Formulas

- A relation name followed by an argument list in parentheses is an **atomic formula**
- These take `True/False` values and can be combined using *logical connectives*
  - This is works the same as it does with propositions
  - So, we can write things like `Human(Socrates)` -> `Mortal(Socrates)`

- If all arguments in the relation are in the domain, an atomic formula must either be true or false

Expressing the values of relations

- Since the domains now consist of more than $\{0, 1\}$, we can no longer express the behavior of a formula for all of its domain using a truth table
- Instead, we must use a regular table/matrix
  - Each row/col is populated with each element in the domain (if not infinite)
  - The corresponding entry denotes whether `relation(row, col)` is 1 or 0
  - This works for a two-parameter relation; if the relation has n parameters, the corresponding table will be n-dimensional
  - Luckily, most mathematical relations do not have more than 2 arguments
- Example: relation table for `greater(u, v)`

u/v	1	2	3	4
1	0	0	0	0
2	1	0	0	0
3	1	1	0	0
4	1	1	1	0

Formulas and terms

- Like in propositional logic, we can give names to formulas:
  - $A = [Human(u) \implies Mortal(u)]$
- A variable can be used anywhere an individual can
- Together, variables and individuals are called **terms**

Quantifiers

- Quantifiers signify how often things are true
- Variables are **bound** by quantifiers
- These quantifiers depend on the domain
- **Universal Quantifier**: $A(u)$ is true for all values of u ($\forall u, A(u)$)
  - Names: "for all", "given any", "given an arbitrary", etc
  - Ex. "Everyone needs a break": $\forall p, break(p)$
- **Existential Quantifier**: $A(u)$ is true for at least one u ($\exists u, A(u)$)
  - Usually called "there exists", can also be called "for some"

- In first order logic, there is no direct expression for the quantifier "For no individuals in the domain"
  - However, both $\neg\exists x P(x)$ and $\forall x \neg P(x)$ are both correct (more on why later)
- Quantifiers are treated like unary connectives
  - They take higher precedence than binary connectives

Quantifiers with subsets

- Here, some subset of the domain has property $a(x)$, and some subset of that has $b(x)$
- $\forall x(a(x) \implies b(x))$ means all members in the domain $a(x)$ have property $b(x)$
  - NOT $\forall x(a(x) \land b(x))$ (this suggests *all* x are both $a(x)$ and $b(x)$)
  - All dogs are mammals: $\forall x(dog(x) \implies mammal(x))$
- $\exists x(a(x) \land b(x))$ means some $a(x)$ are $b(x)$
  - Some dogs are brown: $\exists x(dog(x) \land brown(x))$
- Only dogs bark: $\forall x(barks(x) \implies dog(x))$

Bound vs. Free Variables

- A variable is **bound** if it appears in a quantifier
  - Bound variables are within the local scope of the quantifier
- Otherwise, a variable is **free**

Socrates

- Return to the syllogism

#	Premise	FOL translation
1	All humans are mortal	$\forall x(Human(x) \implies Mortal(x))$
2	Socrates is human	$Human(Socrates)$
Conclusion	Socrates is mortal	$Mortal(Socrates)$

- Socrates' most famous quote: "I know that I know nothing"

Nested Quantifiers

- Nested quantifiers can exist
- $\forall$ and $\exists$ are not commutative: their order matters
  - However, $\forall x \forall y$ means the same as $\forall y \forall x$ (and same with $\exists$)
- Having nested quantifiers can be thought of like using nested loops

Negating Quantifiers

- The negation of an existential quantifier is a universal quantifier with the interior expression negated, and vice versa
  - I.e. $\neg\forall x P(x) \equiv \exists x \neg P(x)$ and $\neg\exists x P(x) \equiv \forall x \neg P(x)$
- This stems from De Morgan's Laws:
  - The universal quantifier is like conjunction:
    $\forall x R(x) = 1 \iff R(d_1) \wedge R(d_2) \wedge \cdots \wedge R(d_n) = 1$, where the domain is $D = \{d_1 \ldots d_n\}$
  - In the same way, the existential quantifier is like disjunction
  - Negating a conjunction yields the disjunction of the negations of the variables; so, the negation of a universal quantifier yields the existence of the negation of the interior expression
- $\forall$ is a generalized $\wedge$ and $\exists$ is a generalized $\vee$

Lecture 11 - Syntax of First order Logic

- In propositional logic, formulas are built recursively from atoms and the rules concerning connectives
- In first order logic, we need more specific formula rules
  - **Domain**: a specification of the basic objects
  - **Terms**: expression that refer to objects in the domain
  - **Atomic formulas**: use relations to combine symbols into simple true/false propositions
  - **Formulas**: Build recursively from atomic formulas using connectives $\{\neg, \wedge, \vee, \implies, \iff\}$ and quantifiers $\{\exists, \forall\}$

First Order Languages

- No single first order language exists. Instead, each case requires a combination of
  - **Logical Symbols**: Fixed syntactic use and semantic meaning (ex. $\neg, \wedge, \vee, \implies, \iff, \forall, \exists$)
  - **Non-logical symbols (parameters)**: Symbols with designated syntax but non-predefined meaning, i.e. math symbols (0, $\geq$, $\times$, successor function $s(x)$)

The Language $\mathcal{L}$ of first-order logic

- $\mathcal{L}$ consists of expressions using the following basic symbols
  - Logical Symbols
    - Connectives: $\neg, \wedge$, etc.
    - Free variable symbols
    - Bound variable symbols
    - Quantifiers: $\exists$, $\forall$
    - Punctuation: $(), \prime\prime, \prime$
  - Non-logical symbols
    - Constants (aka. individual symbols)
    - Relation symbols/predicate symbols (formulas in $\mathcal{L}^p$?)
    - Function symbols ($f(x)$, etc.)
- $\mathcal{L}$ is not a single language; we can choose which non-logical symbols we want
- Each relation symbol and function symbol has an arity
- $\mathcal{L}$ may or may not contain the equality symbol $\approx$
  - If it does, $\mathcal{L}$ is called a *first order language with equality*

Terms of $\mathcal{L}$

- **Term**: either an individual or a variable (i.e. anything that can be used in place of an individual).
- Functions taking terms as parameters are also terms
- Terms name an object in the domain; they act like nouns (individuals) and pronouns (variables) in the English Language

Example: First order language of number theory

- Equality relation: yes
- Relation symbols: $>$
- Constant symbols: 0
- Function symbols
  - Arity 1: s (successor; $s(0)$ is 1, $s(1)$ is 2, etc.)
  - Arity 2: $+$ and $\times$
- Example of a term: $+(u, 0), (s(s(0)))$, more commonly written as $u + 0$ and 2

Atomic formulas of $\mathcal{L}$

- Simples formula expressing a proposition
- Must be in one of the following two forms
  - $F(t, \ldots, t_n)$ where F is an n-ary relation symbol and $t_1, \ldots t_n \in Term(\mathcal{L})$
  - $\approx (t_1, t_2)$, where $t_1, t_2 \in Term(\mathcal{L})$ (not sure why this isn't a subcase of the first form)

Formulas of $\mathcal{L}$

- Built recursively, with atoms being the base case, using rules that describe when connectives and quantifiers may be used
- Formation rules
  - Every atom is a formula ($Atom(\mathcal{L}) \subset Form(\mathcal{L})$)
  - If A is a formula in $Form(\mathcal{L})$, then $\neg A$ is also a formula in $Form(\mathcal{L})$
  - If A and B are formulas in $Form(\mathcal{L})$, then $A \wedge B$, $A \vee B$, $A \implies B$, and $A \iff B$ are formulas in $Form(\mathcal{L})$
  - If $A(u)$ is a formula in $Form(\mathcal{L})$ with free variable u and x is a variable not present in $A(u)$, then $\forall x A(x)$ and $\exists x A(x)$ are formulas in $Form(\mathcal{L})$ ($A(x)$ denotes $A(u)$ where u is replaced with x)
- A formula (or term) is **closed** if it has no free variables; a closed formula is also known as a **sentence**
  - The set of sentences is denoted $Sent(\mathcal{L})$

- Example of a formula: $\forall x \forall y((x < y + 1) \implies \exists z((x < z) \wedge (z < y))$
- There are eight types of formulas in $Form(\mathcal{L})$: atoms, formulas joined with connectives (x5), and quantified formulas (x2)
- A parse tree can be made for a given formula

Lecture 12 - Semantics of First Order Logic

- Although $\mathcal{L}$ is purely syntactic, formulas in $Form(\mathcal{L})$ are meant to express statements with semantic meaning
- Like before, a truth valuation assigns truth values to proposition symbols and a truth value is determined
- A **valuation** of $\mathcal{L}$ consists of an **interpretation** of its non-logical symbols and **assignment** of values to its free variables
  - Must contain sufficient information to determine a truth value

Logical Symbols

- Connectives are interpreted as they are in propositional logic
- Logical symbols in $\mathcal{L}$: **connectives**, **quantifiers**, the **equality symbol** $\approx$, **variable symbols**, and **punctuation symbols**

Non-logical symbols

- A **valuation** is an **interpretation** plus an **assignment**
- **Interpretation** consists of:
  - A set of objects that (**domain**)
  - A specification for each individual symbol, relation symbol, and function symbol of the actual individuals, relations, and functions that are denoted
- **Assignment**: assigns a free variable to a value in the domain
- A valuation v of a symbol s is denoted s^v
- If $\mathcal{L}$ is not associated with a theory or structure (ex. number theory), the domain can be an arbitrary non-empty set and all non-logical symbols, relations, and functions are arbitrary

Describing relations as sets

- An n-ary relation R on the set D can be thought as a subset R of $D^n = D \times D \times \cdots \times D$
- Specifically, $R = \{(a_1, \ldots, a_n) \mid a_i \in D, \text{ and } R(a_1, \ldots a_n) = 1\}$
  - Ex. equality: $\{(x, y) \mid x, y \in D \text{ and } x = y\}$

Valuations: formal definition

- A **valuation** for the first order language $\mathcal{L}$ consists of
  - A domain D (must be non-empty)

        - The domain must be non-empty because everything is vacuously true following an empty domain
    - A function v with the following properties
        1. For each individual symbol a and free variable u, $a^v, u^v \in D$
        2. For each relation symbol F with arity n, F^v is an n-ary relation on D (i.e. $F^v \subseteq D^n$)
        3. For each m-ary function symbol f, we have $f^v : D^m \to D$
- Valuations are the *interpretation* of individual symbols, relation symbols, function symbols and the *assignment* of values to free variable symbols

Value of a term

- The value of a term t under valuation v over the domain D is denoted t^v and is defined recursively
    - If t is an individual symbol a or free variable u, then its value is $a^v \in D$ or $u^v \in D$
    - If t is an m-ary function $f(t_1, \dots t_m)$ where each t_i is a term, then $f(t_1 \dots t_m)^v = f^v({t_1}^v, \dots {t_m}^v)$
- If v is a valuation over D and t is a term, then $t^v \in D$

Qualified formulae

- For a valuation v, free variable u, and individual $d \in D$, $v(u/d)$ denotes the valuation that is the same as v, except u is replaced with d
    - I.e. $w^{v(u/d)}$ is d if $w = u$ and w^v otherwise
- $\forall x A(x)^v$ is 1 if $A(u)^{v(u/d)} = 1$ for every $d \in D$ (i.e. every value in the domain leads to 1) and 0 otherwise
- $\exists x A(x)^v$ is 1 if $A(u)^{v(u/d)} = 1$ for some $d \in D$ (i.e. at least one value in the domain leads to 1) and 0 otherwise

Value of a formula

- Recursive definition of valuation v with domain D
    - $R(t_1, \dots t_n)^v$ is 1 iff $(t_1^v, \dots t_n^v) \in R^v \subseteq D^n$
    - $(\neg A)^v$ is 1 if A^v is 0
    - $(B \wedge C)^v$ is 1 if both B^v and C^v are 1
    - etc. with the rest of the regular connectives
    - $\forall x A(x)^v$ and $\exists x A(x)^v$ as defined above

Satisfiability and universal validity

- A formula $A \in Form(\mathcal{L})$ is **satisfiable** iff there exists a valuation v such that $A^v = 1$

    - Ex. $A = f(g(a), g(u)) \approx g(b)$
        - One true valuation: f is addition, g is squaring, a is 3, u is 5, and b is 5
        - However, there are valuations do not lead to 0
    - Otherwise, it is **unsatisfiable**
- A formula $A \in Form(\mathcal{L})$ is **universally valid** iff $A^v = 1$ for every valuation v
    - Ex. $A = F(u) \vee \neg F(u)$
    - These are the counterparts of *tautologies* in $Form(\mathcal{L}^p)$
- The same definition can apply to a set of formulae $\Sigma \subseteq Form(\mathcal{L})$
- In general, there is no algorithm for deciding the universal validity or satisfiability of formulas in first-order logic (Church, 1936)

Comparison: higher-order logic

- First order logic: variables range over members of the domain
- Second order logic: subsets of the domain and relations on the domain can be used as variables
    - Ex. "Every non-empty subset of natural numbers has a smallest element"
- Higher-order logic: variables (and quantifiers) for sets of sets, sets of sets of sets, etc. are also allowed

History of Logic Timeline

- Aristotle: earliest study of formal logic
- George Boole: propositional logic
- Gottlob Frege: domains, variables, relations, functions, quantifiers
    - Intended to express all of mathematics
    - However, Bertrand Russel pointed out a paradox of Frege's system: the set $R = \{S \mid S \text{ is a set}, \text{and } S \notin S\}$ (the set of sets that do not contain themselves)
        - Is R a member of itself? This is a paradox
    - Restricting to first-order logic avoids Russel's paradox

Lecture 13 - Logical Consequence in First order logic

- The notation $\vDash$ is used for logical consequence as well
- We have $\Sigma \vDash A$ iff $\Sigma^v = 1 \implies A^v = 1$
- We have also use $\nvDash$ and $\equiv$ in the same way as propositional logic
  - $\equiv$ still denotes logical equivalence

Proving arguments

- Proof by contradiction: we assume the negation of the conclusion and show that the resulting argument is invalid
- We proceed in logical steps to find it
- Finding a counterexample usually involves deriving some $\exists$ formula
  - This can be achieved by negating a $\forall$

Refuting arguments

- This can be done by finding a single counterexample

Empty Premises (proving universal validity)

- We have $\emptyset \vDash A$ iff A is universally valid, since $\emptyset$ is vacuously true under any valuation
  - Therefore A must be true under any valuation for the implication to be correct
- Universal validity can be proven by contradiction
  - The opposite can be proven by counterexample
- It is not always possible to determine whether a formula is universally valid (Church)
  - Some formulas can (ex. $\forall x P(x) \lor \neg(\forall x P(x))$)

Replaceability and Duality in FOL

- **Replaceability**: Let $A \in Form(\mathcal{L})$ contain subformula $B \in Form(\mathcal{L})$. Let $B \equiv C$ and A' be the formula where B is replaced with C in A. Then $A \equiv A'$
  - Recursive: we have $\forall x A(x) \equiv \forall x A'(x)$, $\exists x A(x) \equiv \exists x A'(x)$, $\neg A \equiv \neg A'$, etc.

- **Duality**: Let $A \in Form(\mathcal{L})$. Derive $\Delta(A)$ from A by swapping $\wedge$ and $\vee$, $\exists$ and $\forall$, and each atom with its negation. Then $\neg A \equiv \Delta(A)$

Lecture 14 - Formal Deduction in First order logic

- For $\mathcal{L}^p$, we defined a calculus of reasoning: the 11 rules of formal deduction. This let us prove things semantically
- We would like to extend this system to handle first order logic
  - *We would like to be able to formally prove anything that is semantically correct*
- For this, we add 6 additional rules concerning quantifiers and the equality symbol $\approx$

Abbr.	Rule
$\forall -$	If $\Sigma \vdash \forall x A(x)$ is a theorem, then $\Sigma \vdash A(t)$ is a theorem where t is a term
$\forall +$	If $\Sigma \vdash A(u)$ is a theorem and u is not in Σ, then $\Sigma \vdash \forall x A(x)$ is a theorem
$\exists -$	If $\Sigma, A(u) \vdash B$ is a theorem and u is not in Σ or B, then $\Sigma, \exists x A(x) \vdash B$ is a theorem
$\exists +$	If $\Sigma \vdash A(t)$ is a theorem, then $\Sigma \vdash \exists x A(x)$ is a theorem (some ts are replaced with x)
$\approx -$	If $\Sigma \vdash A(t_1)$ and $\Sigma \vdash t_1 \approx t_2$ are theorems, then $\Sigma \vdash A(t_2)$ is a theorem (some replacement)
$\approx +$	$\emptyset \vdash u \approx u$ is a theorem

Explanations

$\forall +$

- Intuitive meaning: if any element of a set has a given property, then every element must have the property
  - "Any" here means an arbitrary element, not any particular one
  - u cannot be in Σ because the choice of u must be independent of the premises
  - If u appears in a premise as a free variable, it would be "fixed" because it would always refer to that same individual

$\exists -$

- Once again, u cannot be in Σ because the choice of u must be independent of the premises

Formal Deducibility

- Let $\Sigma \subseteq Form(\mathcal{L}) \ni A$. A is **formally deducible** from Σ in first-order logic $\iff \Sigma \vdash A$ can be generated using the 17 rules of formal deduction

Replaceability and Duality

- **Replaceability**: Let $A, B, C \in Form(\mathcal{L})$ with $B \vdash\dashv C$. Let A' be A with some (not necessarily all) instances of B replaced with C. Then $A' \vdash\dashv A$.
- **Duality**: Let $A \in Form(\mathcal{L})$. Derive $\Delta(A)$ from A by swapping $\wedge$ and $\vee$, $\exists$ and $\forall$, and each atom with its negation. Then $\neg A \vdash\dashv \Delta(A)$

Soundness and Completeness

- Let $\Sigma \subseteq Form(\mathcal{L})$ and $A \in Form(\mathcal{L})$. Then $\Sigma \vDash A$ iff $\Sigma \vdash A$
  - Soundness: $\Sigma \vdash A$ implies $\Sigma \vDash A$
  - Completeness: $\Sigma \vDash A$ implies $\Sigma \vdash A$

Proof strategies

1. Ignore the quantifiers at first and try to figure out the proof using propositional logic. Once we know the shape, our proof may look like:
   - Remove quantifiers using $\forall-$ and $\exists-$
   - Propositional logic proof
   - Introduce quantifiers using $\forall+$ and $\exists+$
2. If one of the premises in Σ is existentially quantified, $\exists$ can be removed by
   - Replacing the premise $\exists x A(x)$ with $A(u)$, resulting in Σ'
   - Continue the proof using Σ' as the premises
   - Use $\exists-$ to reintroduce $\exists$ at the end. u must not appear anymore in the premises or the conclusion

Lecture 15 - Resolution for FOL

Prenex normal form

- **Prenex normal form**: all quantifiers are at the start of the equation
  - i.e. in the form $Q_1x_1(Q_2x_2(\dots Q_nx_n(B)))$ where Q_i is a quantifier, x_i is a bound variable, and B is a quantifier-free expression
  - B is called the *matrix*, $Q_1x_1\dots$ is called the *prefix*
  - An expression with no quantifiers is trivially in prenex normal form

Algorithm for converting to PNF

- Every formula in $Form(\mathcal{L})$ is logically equivalent to a formula in PNF
- It can be found using the following steps:
  1. Eliminate all occurrences of $\implies$ and $\iff$
  2. "Apply negations" so that only atoms are negated. Usually requires De Morgan's laws, double negation, etc
  3. Standardize the variables apart wherever necessary
     - Meaning: make sure that separately bound variables all have different names (i.e. there are two different xs in $\forall x(A(x) \vee B(x)) \vee \exists xC(x)$)
     - This is fine due to the **replaceability of bound variable symbols**
  4. "Move" all the quantifiers to the front of the formula
     - This can be done if the equation being "moved over" doesn't depend on the variable bound by the quantifier

$\exists$-free PNF

- Must be in PNF and not contain any $\exists$ symbols
- Algorithm for generating:
  1. Convert to PNF
  2. For each $\exists$: For each $\forall$ (starting with the outermost one), replace the existentially bound variable with the appropriate skolem function

Skolem functions

- Consider the sentence $\forall x_1\forall x_2\dots\forall x_n\exists yA$
  - $\exists yA$ generates at least one individual for n-tuple $(a_1,\dots a_n)$ in the domain
  - I.e. y can be expressed as a function of $x_1,\dots x_n$, i.e. $f(x_1,\dots x_n)$

- f is called a Skolem function, and must be denoted by a new function symbol that does not occur in A
- The **skolemized version** of $\forall x_1 \forall x_2 \dots \forall x_n \exists y A$ is $\forall x_1 \forall x_2 \dots \forall x_n A'$, where each occurrence of y in A' is replaced with its skolem function $f(x_1, \dots x_n)$
  - It is possible to actually define this function
- The skolemized function is not (generally) logically equivalent to the original sentence, since the existential quantifier may imply more than one individual
- If we think of individual symbols as function with arity 0, the skolemized sentence is valid even if there are no universal quantifiers

Dropping $\forall$

- Let $F \in Sent(\mathcal{L})$ and F' denote the PNF of F
- F' is satisfiable $\iff$ there is an effective procedure for finding an $\exists$-free PNF of F' such that F is satisfiable
- Once all the $\exists$ have been removed, we can drop the universal quantifiers since sentences without quantifiers are implicitly assumed to be universally quantified
  - I.e. $\forall y \forall z P(x, y, z)$ becomes $P(x, y, z)$

Converting to clauses

- Given a sentence F in $\exists$-free PNF, a finite set of clauses C_F can be constructed such that (F is satisfiable $\iff$ C_F is satisfiable)
- Algorithm
  - Convert to $\exists$-free PNF if necessary
  - Put the matrix of F into CNF
  - Read off the clauses

Validity and Satisfiability of clause set

- Let Σ be a set of sentences and A be a sentence. The argument $\Sigma \vDash A$ is valid $\iff$ the set $C_{\Sigma, \neg A} = \bigcup_{F \in \Sigma} C_F \cup C_{\neg A}$ is *not* satisfiable
  - I.e. the set of clauses from the premises and the clauses from the negated conclusion
- So, if we wish to show the validity of an argument, we must show that $C_{\Sigma, \neg A}$ is not satisfiable.
  - We can use resolution to find this

Resolution for FOL

- Resolution works similarly for FOL compared to propositional logic

Unification

- **Instantiation**: An assignment of a quasi term t_i to a variable x_i.
  - A quasi-term is either an individual symbol, a variable symbol, or a function symbol applied to either of the above two
- **Unification**: Two FOL formulas **unify** if there are instantiations that make the formulas identical
  - The instantiation in question is called the **unifier**
  - This works because all variables are universally qualified
- Example: $Q(a, y, z)$ and $Q(x, b, c)$ can be unified using the instance $(y := a, z := b, x := c)$
- Not all expressions can be unified
  - Ex. $Q(a, b, y)$ and $Q(c, b, y)$ cannot be unified because we can't make individual a individual c since they occupy the same place

Resolution

- We use unification to create complimentary literals so that we can apply resolution
- As before, we can resolve $p \vee A(x)$ and $\neg p \vee B(x)$ to $A(x) \vee B(x)$
- We can discard resolutions that are universally true, since they will never lead to a contradiction
  - We can also discard duplicate literals
- A clause can be used as a parent multiple times (and can be instantiated multiple times)

Automated Theorem Proving

- A **theorem** is a logical argument with several **premises** and a **conclusion**
- To prove theorem automatically, we must transform the premises and negated conclusion into clauses:
  - Convert each formula to PNF
  - Replace $\exists$s with skolem functions
  - Drop the $\forall$s (by convention)
  - Convert the matrix to CNF and extract clauses
- Then, we must use unification to create complimentary literals to determine a resolvent
- If the empty clause $\{\}$ is reached, the set is not satisfiable and the theorem is valid
- Examples of programs for this: E (prover for first order logic with equality), SPASS, Vampire

Automatic Theorem Verification

- We can also verify that proofs are correct mechanically
  - Programs that do this are often called proof assistants

- Instead of creating new proofs, it just checks
- Examples: Isabelle (higher-order logic), Coq (interactive theorem prover)

Soundness and Completeness of Resolution

- A set $\mathcal{S}$ is not satisfiable iff there is a resolution derivation of the empty clause $\{\}$ from $\mathcal{S}$
  - **Soundness**: empty clause $\rightarrow$ not satisfiable
  - **Completeness**: not satisfiable $\rightarrow$ empty clause

Lecture 16 - Decidability and Computability

What is an algorithm?

- Finite sequence of well-defined, computer-implementable instructions to perform a computation
- It solves a problem if it produces a correct output for every input
- There exist problems that cannot be solved by a computer algorithm

Halting Problem

- We cannot write a program that takes a program P and its input I and return whether P halts (stops running) with input I.

Proof of the Halting Problem

- The proof is by contradiction
- Assume that there is an program $H(P, I)$ that solves that halting problem by outputting "halts" if the program halts and "loops forever" if it doesn't
- This program must be expressed in the form of a string of characters/bits, so it can itself be used as data
- So, we should be able to compute $H(P, P)$
- Construct a program $K(P)$ that halts if $H(P, P)$ prints "loops forever" and goes into an infinite loop if $H(P, P)$ outputs "halt"
  - I.e. has the opposite output of $H(P, P)$
- What happens if we call $K(K)$?
  - Case: $K(K)$ halts. So $H(K, K)$ must have outputted "halts" by definition. However, by the construction of $K(K)$, $K(K)$ must have looped forever (contradiction)
  - Case: $K(K)$ loops forever. So, $H(K, K)$ must output "loops forever". However, by the construction of $K(K)$, $K(K)$ must have halted (contradiction)
  - Thus, by contradiction, the program $H(P, I)$ cannot exist

Turing Machines

Informal description

- **Turing machine**: simple mathematical model of a computation
- Consists of

    - Finite control unit: has read-write head and can be in a finite amount of states
    - Two-way infinite tape, divided into cells. The cells can be read and written to, and can affect the state of the machine

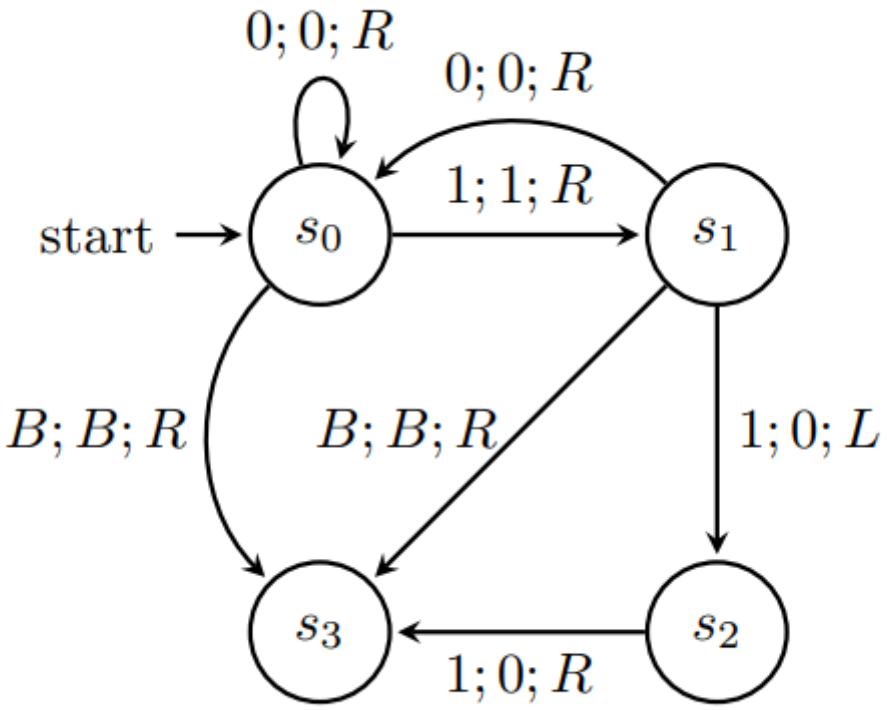
Formal description

- Turing machine: $T = (S, I, f, s_0)$
    - S: infinite set of states
    - I: input alphabet (finite set of symbols) that contains the blank symbol B
    - $s_o \in S$: the initial state
    - $f : S \times I \to S \times I \times \{L, R\}$: transition function
- The machine moves along the tape and performs an action at each step
    - This action is determined by the transition function f, which takes the current state and tape symbol into account
    - If we have current state s and current symbol x, then we have $f(s, x) = (s', x', d)$, where s' is the new state, x' is written to the current cell, and d determines the direction moved (d can be L or R)
    - This can be written as the 5-tuple (s, x, s', x', d), which is a *transition rule*
        - The function is kind of like a the set of transition rules
    - If the tuple (s, x) is undefined, then the machine *halts*
    - The last state before this happens is called the *final state*
- TMs are often defined by a set of transition rules or a transition diagram, where each state represents a node, the initial and final states are specified, and transition rules are arrows

Alternate Representation

- We can denote $T = (S, I, f, s_0)$ as $\alpha_1 s \alpha_2$, where $s \in S$ is the current state and $\alpha_1\alpha_2$ is the content of the tape (α_1 is the string before the head, α_2 is the string after it)

Graphical Representation



- States are represented by nodes
- Transition rules are represented by arrows (directed edges) labeled r, w, d
- A computation is a path in the graph
- The starting state is an incoming arrow
- The final (halting) state is a double circle node

Computation

- A **computation** consists of successive applications of the transition rules to the content of the tape
  - A single application is a *transition step*

Languages

- An **alphabet** Σ is a finite, non-empty set of symbols (ex. $\{0, 1\}$)
  - Σ^* denotes all the possible strings in that language (including the empty string λ)
- A **language** L over Σ is a subset of Σ^*
  - Ex. $L = \{w \in \{0,1\}^* \mid w \text{ is a string with an equal amount of 1s and 0s }\}$
- TMs can recognize and accept languages
- Let $V \subseteq I$. A TM $T = (S, I, f, s_0)$ accepts a string $x \in V^*$ iff it halts in a final state when x is written on the tape
  - If T does not halt or halts in a non-final state, then x is not accepted

    - T recognizes $L \subseteq V^*$ if $x \in L$ and x is recognized by T
- Symbols not in V may be present in I and thus Σ; they are often used as markers

Turing machines as functions

- We can think of turing machines as computing a function $T(x) = y$, where x is the input (initial) string and y is the string on the tape when the function halts
    - The domain of T is the set of strings for which T halts
    - If T does not halt with input x, $T(x)$ is undefined
- Using the alphabet $\{0, \ldots, 9\}$, we can compute functions $g : \mathbb{N} \to \mathbb{N}$
- A **total turing machine** always halts, no matter the input

Computing number-theoretic functions

- To compute functions $f : \mathbb{N}^k \to \mathbb{N}^m$, we must be able to represent arbitrary $n \in \mathbb{N}$ on tape
- We can use *unary*: n is represented by a string of $n + 1$ 1s
    - So, 0_{10} is 1_1 and 5_{10} is 111111_1
- For n-tuples, we can separate each member by an asterisk (marker character)

Variations on Turing machines

- Multi-tape: has the same computing power as a regular one, and makes creating Turing machines less tedious
- Can stop at a given step instead of moving left or right
- Two-dimensional tape (and directions)
- Multiple tape-heads
- Non-deterministic: different transition rules with the same starting state and alphabet symbol
    - Instead of defining a function to determine the next state, we use a relation (not necessarily one-to-one)
- Restricted alphabet
- Non-infinite tape length in one or both directions

Turing machines as a computational model

- Many other types of TMs exist, but they all have the same amount of computing power
- There exists a *universal turing machine* that can simulate all the computations of any other turing machine (an encoding of the TM and its input is given as the input)
    - This is also known as a *computer*

- **Church-Turing thesis**: *Any problem that can be solved with an algorithm can be solved using a turing machine*

Computability and Decidability

- **Decision problem**: yes or no question on an infinite set of inputs
  - Ex. Is a first-order logic formula satisfiable?
  - Can be thought of as a language L of input strings
- **Decidability**: a decision problem is *decidable* if there is a terminating algorithm that can solve it. Otherwise, it is *undecidable*
  - Ex. Whether an arbitrary $(x \in \mathbb{N}) \in S \subseteq \mathbb{N}$. This is undecidable for some S
    - If $|S|$ is finite, then S is decidable. Some infinite S are also decidable
  - Decidable problems
    - Determining whether $x \in \bar{S}$ given S
    - Determining whether a propositional formula A^p satisfiable, a tautology, or a contradiction
  - Undecidable problems
    - The halting problem
    - Determining whether a first-order logic formula is satisfiable or valid
    - Tiling problem
      - We can use a TM to simulate tilings and the process of finding tilings to simulate a TM
- **Computability**: A function that can be computed by a TM is computable

Proving Undecidability

- Undecidability is proven using **reduction**
- If we want to prove problem P undecidable, then we show that if Q is decidable, then P is decidable. Then, by contrapositive, if P is not decidable, then Q is not decidable
- This can be done that showing that P and Q are equivalent problems in some way (similar to bijection proofs) by creating an algorithm that turns an instance of the problem P into an instance of the problem Q
  - I.e. how solving problem Q can also solve problem P
  - This is called a **reduction** from P to Q
- The correct way to prove undecidability is to reduce the problem to an existing undecidable one
  - Ex. The halting problem can be reduced to the blank tape halting problem, so we know that the blank tape halting problem is undecidable
  - Ex. The state-entry problem can be reduced to the halting problem

        - State-entry problem: does TM T enter state q on input w
    - In these examples, we assume we have a halting problem decider and use it to build a whatever decider

Uncomputability

- A function can be computed by a total Turing machine $\iff$ the function is computable
- Uncomputable function: the *Busy Beaver* function $B(n)$
    - Let $B(n)$ be the maximum number of 1s that Turing machine with n states and alphabet $\{1, B\}$ may print on an initially blank tape
    - We know $B(2) = 4$, $B(3) = 6$, and $B(4) = 13$, but we don't know $B(n)$ for $n \geq 5$, although we know $B(5) \geq 4098$ and $B(6) \geq 10^{18267}$
- Every decision problem can be represented as a function (i.e. the one that outputs for 1 for "YES" and 0 otherwise)
    - Function is computable $\iff$ problem is decidable

Complexity

- Decidable problems can be ranked by "difficulty" to solve
    - We can measure this in terms of time and space
- **Computational complexity of an algorithm**: number of operations used by an algorithm as a function of input size
- **Computational complexity of an problem**: number of operations used by the most efficient algorithm that solves a problem
    - If the Church-Turing thesis is true, this can be defined explicitly by counting the number of transitions in the turing machine computation
    - Ex. constant, linear, quadratic $\subset$ polynomial, exponential, factorial, etc.

P vs. NP

- A decision problem is P (a polynomial-time problem) if it can be solved by a deterministic turing machine in polynomial time
    - Equivalent to statements solvable in polynomial time by a non-deterministic Turing machine
    - This is because a non-deterministic machine would , ,,,,
- The problem is classified as NP if it can be solved by a non-deterministic TM in polynomial time
    - This is equivalent to the set of decision problems where the problem instances with answer "YES" have proofs verifiable in polynomial time by a deterministic Turing machine

    - Reasoning: solving the problem non-deterministically would consist of
        1. A non-deterministic "guess" at the solution
        2. Verifying the guessed solution, which we know is possible in polynomial time
- Trivially, we have $P \subseteq NP$
- To prove a problem is NP, we must show that no polynomial time algorithm exists to solve it
- The satisfiability problem (whether a given propositional logic equation with n symbols is satisfiable) was the first to be proven NP-complete since essentially 2^n checks must be performed
    - Verifying an equation is much easier

$P = NP?$

- Can problems in NP be solved by polynomial time algorithms?
- It is generally believed that this is not the case, i.e. $P \neq NP$
    - I.e. there are problems that are harder to solve than to verify
- A problem is NP-complete if any other NP problem can be reduced to it
    - So, finding any polynomial time solution for an NP problem would show $P = NP$

Lecture 17 - Peano Arithmetic and Godel's Incompleteness Theorem

Properties of Equality

Abbr.	Theorem
$\approx -$	If $\Sigma \vdash A(t_1), t_1 \approx t_2$, then $\Sigma \vdash A'(t_2)$ where some occurences of t_1 are replaced by t_2
$\approx +$	$\emptyset \vdash u \approx u$

- These can be used to prove the **reflexivity** ($x = x$), **symmetry** ($x = y \implies y = x$), and **transitivity** ($x = y \land y = x \implies x = z$) of equality

First-order theories

- FOL can be used to describe specialized domains (*theories*) that contain a small number of relations and functions
- Each theory has **domain axioms** which are FOL statements we assume to be true
- **Theory**: set of domain axioms together with a system of formal deduction (from which all theorems in the theory can be proven)
  - Ex. Number theory, set theory, group theory, etc.
- For a theory T with axioms A_T, we use $\Sigma \vdash_{A_T} C$ to denote $\Sigma, A_T \vdash C$

Requirements for domain axioms A

- A should be decidable: there should exist a terminating algorithm that can determine whether a given formula is a domain axiom
- A should be consistent $\iff$ satisfiable
- A should be syntactically complete: For all $F \in A$, either F or $\neg F$ must be provable
  - This is not the same as semantic completeness

Example: Euclidian Geometry

1. A straight line exists between any two points
2. A straight line can be extended infinitely
3. A circle can be drawn with any given point as a center with any given radius
4. All right angles are equal

5. Parallel postulate: For any given point not on a given line, there exists one line passing through that point that is parallel to the given line
    - The parallel postulate is not provable from the first four axioms

Peano Arithmetic

- Peano's axioms form the basis of the Peano Arithmetic version of number theory
- Individual: 0
- Functions: *successor* $s(n)$, addition $+$, multiplication $\times$
- Relation: Equality $\approx$ (already a part of FOL)
- Axioms (denoted PA)

Number	Category	Axiom
PA1	Successor	$\forall x \neg(s(x) = 0)$
PA2	Successor	$\forall x \forall y((s(x) = s(y)) \implies (x = y))$
PA3	Addition	$\forall x(x + 0 = x)$
PA4	Addition	$\forall x \forall y(x + s(y) = s(x + y))$
PA5	Multiplication	$\forall x(x \times 0 = 0)$
PA6	Multiplication	$\forall x \forall y(x \times s(y) = x \times y + x)$
PA7	Induction	$(A(0) \land \forall x(A(x) \implies A(s(x)))) \implies \forall x A(x)$

Proofs in Peano Arithmetic

- We use induction to prove statements about all $n \in \mathbb{N}$
- Proof example: $\forall x(s(x) \neq x)$
    - Prove $A(0)$ (given as PA1)
    - Prove $\forall x(A(x) \implies A(s(x)))$
    - Prove $A(0) \land \forall x(A(x) \implies A(s(x)))$ from the first two using $(\land +)$
    - Obtain $\forall x A(x)$ by PA7 and $(\implies -)$

The big picture

- We can use FOL and the Peano axioms to construct other relations and theorems
    - Ex. $u \geq v$ is $\exists z(u + z = v)$
    - Ex. $Prime(u)$ is $(a < u)\neg(\exists z \exists y((u = y \times z) \land (a < y) \land (a < z)))$
    - Peano arithmetic can then be used to prove properties of the relations

- Any number theory theorem can be obtained using the Peano axioms and the 17 rules of formal deduction

Gödel's Incompleteness Theorem

- In any consistent formal theory T with a decidable set of axioms, that is capable of expressing elementary arithmetic (ex. Peano Arithmetic), there exists a statement/formula that can neither be proved nor disproved in the theory
  - Original proof: constructs a statement G_T that states G_T is unprovable in T
- Example proof: reduction to the halting problem
  - Write a TM that takes two inputs: a program P and an input I for P
  - It generates all strings s that use the Latin alphabet and math symbols in increasing length order
  - It then checks whether s is a formal proof of the statement P halts in input I or its negation
  - Since either "P halts on I" or "P does not halt on I" can be proven, our program terminates and gives a yes or no answer
  - This has just solved the halting problem
  - Since the halting problem is undecidable, our assumption was incorrect. Therefore, there exist statements expressible in T that can neither be proven or disproven using the axioms T
- This works with any theory T that can express a Turing Machine (Peano arithmetic is such a system)
- Hidden assumption: T must be consistent (otherwise we could prove both "P halts on I" and "P does not halt on I")

Lecture 18 - Program Verification

In these notes, we will use angle brackets to denote Hoare triples (ex. $\langle P \rangle C \langle Q \rangle$) because obsidian doesn't support the latex package with the proper notation :(

- **Program Correctness**: does a program satisfy its specification?
- Techniques for verifying program correctness
  - Inspection: code walkthroughs
  - Testing: black box and white box
    - Disadvantage: does not assure that no bugs exist, just that they probably don't
  - Formal verification: proving a program correct using formal deductions
    - State the specification of a problem in FOL and prove that it acts as such using formal deduction
    - Often used in safety-critical software (ex. traffic lights, etc.)

Steps of Formal Verification

1. Convert informal description of requirements R into a formula Φ_R in some symbolic logic
2. Write a program P that realizes Φ_R in some programming environment
3. Prove that the program P satisfies Φ_R
   - We will mostly consider this step

Programming Paradigms

- Imperative
  - Manipulating value of variables
  - **State**: consists of the vector containing the values of all the variables at a particular time in the execution of the program
  - Expressions are evaluated relative to the current state of the program
  - Statements change the state of the program
- Sequential: no concurrency
- Transformational: given inputs, the output is computed and the program terminates

Core programming language

- We will use a fake programming language, with C++ syntax and the following program constructs: integer and boolean expressions, assignment, sequences, if-then-else, while loops

- This is a *Turing equivalent* programming language: it can simulate any Turing machine

Hoare triples

- Have the following form:
  - $\langle P \rangle$: Precondition
  - C: Code (program)
  - $\langle Q \rangle$: postcondition
- We can use the relations $State(s)$, $Condit(P)$, $Code(C)$, $Satisfies(s, P)$, $Terminates(C, s)$ and function $result(C, s)$ to describe the pre and postconditions in FOL
- Meaning of the triple $\langle P \rangle C \langle Q \rangle$: if C is run in starting in a state that satisfies the logical formula P, then the resulting state after the execution of C will satisfy Q
- $\langle P \rangle C \langle Q \rangle$ is called a **Hoare triple**
  - P and Q are written in the first order logic of integers
- We can use Hoare triples to prove pieces of code

Formal specification

- A **specification** of a program C is a Hoare triple where C is the second component
- Often, we don't want to put any constraints on the initial state; the precondition can often be set to "true"

Partial Correctness

- A Hoare triple $\langle P \rangle C \langle Q \rangle$ is satisfied under *partial correctness* ($\vDash_{par} \langle P \rangle C \langle Q \rangle$) iff, for every state s that satisfies condition P, if the execution of C starting with state s terminates with state s', then s' satisfies condition Q
  - I.e. if the program terminates, the triple will be correct
- Partial correctness is weak: a program that never terminates is always vacuously "partially correct"
  - Ex. `while(true) { x = 0; }` satisfies all specifications under partial correctness

Total Correctness

- A Hoare triple $\langle P \rangle C \langle Q \rangle$ is satisfied under *total correctness* ($\vDash_{tot} \langle P \rangle C \langle Q \rangle$) iff, for every state s that satisfies condition P, the execution of C starting from state s terminates and the resulting state s' satisfies Q
- Total correctness = partial correctness + termination

```
// (x = 1)
y = x;
```

```
// (y = 1)
```

```
// (x >= 0)
y = 1;
z = 0;
while (z != x) {
        z = z + 1;
        y = y * z;
}
// (y = x!)
```

Expressing Correctness in FOL

- **Partial correctness** of $\langle P \rangle C \langle Q \rangle$: $\forall s \forall P \forall C \forall Q [State(s) \land Condit(P) \land Code(C) \land Condit(Q) \implies (Satisfies(s, P) \land Terminates(C, s) \implies Satisfies(result(C, s), Q))]$
- **Total correctness** of $\langle P \rangle C \langle Q \rangle$: $\forall s \forall P \forall C \forall Q [State(s) \land Condit(P) \land Code(C) \land Condit(Q) \implies (Satisfies(s, P) \implies Terminates(C, s) \land Satisfies(result(C, s), Q))]$

Proving Correctness

- Goal: proving total correctness
  - Execution: prove partial correctness, then termination. Total correctness follows from this

Proving partial correctness

- We define *inference rules* (similar to formal deduction rules) that can be applied to any state and condition
- A partial correctness proof is an *annotated program*
  - Each program statement has a pre and post condition, which form a Hoare triple. Each triple contains a justification (written to the side)
    - The conditions may require extra (logical) variables that do not appear in the program
  - Usually, these overlap, so the postcondition of a statement is the precondition of the next one
  - These are often assertions
- Proofs are often started bottom-upwards; we usually start with the postcondition
- Then, we need to provide FOL formal deductions for any implications that we used
- (More concrete) proof steps
  1. Annotate the program using inference rules
  2. Moving backwards, add an assertion/condition before each assignment

3. Prove any implications that result using FOL

Inference Rules

- We must think about what we would have to prove about the initial state to prove that Q holds in the final state

Name	Equation	Notes
Assignment	$\langle Q[E/x]\rangle x = E; \langle Q\rangle$	$Q[E/x]$ means we replace x with E. Q depends on x
Composition	If we have $\langle P\rangle C_1\langle Q\rangle$ and $\langle Q\rangle C_2\langle R\rangle$, then we have $\langle P\rangle C_1; C_2\langle R\rangle$	We can merge two adjacent Hoare triples
If-then-else	If we have $\langle P \wedge B\rangle C_1\langle Q\rangle$ and $\langle P \wedge \neg B\rangle C_1\langle Q\rangle$, then we have $\langle P\rangle \text{ if } (B), C_1, \text{ else } C_2\langle Q\rangle$	We can describe a similar version without the else block
Partial While	If we have $\langle I \wedge B\rangle C\langle I\rangle$, then we have $\langle I\rangle \text{ while } (B) C\langle I \wedge \neg B\rangle$	I is the invariant, so it stays the same before, during, and after termination. It expresses some meaningful relationship among the variables used in the loop. The loop condition B must be false after termination, since the loop finished

- Note that only the while loop can be responsible for a non-terminating program in the language we have constructed

Conditional Template

```
(|P|)
if ( B ) {
      (|P ∧ B|)      if-then-else
      C1
      (|Q|)          (justify depending on C1—a "subproof")
} else {
      (|P ∧ ¬B|)     if-then-else
      C2
      (|Q|)          (justify depending on C2—a "subproof")
}
(|Q|)                if-then-else [justifies this Q, given the previous two Q]
```

Partial While Template

```
(|P|)
(|I|)                Implied (a)
while ( B ) {
      (|I ∧ B|)      partial-while
      C
      (|I|)          ⟵ to be justified, based on C
      }
(|I ∧ ¬B|)           partial-while
(|Q|)                Implied (b)
```

Implications

- **Precondition Strengthening**: If $P \implies P'$ and we have $\langle P' \rangle C \langle Q \rangle$, then we also have $\langle P \rangle C \langle Q \rangle$
  - This means we can create a stronger precondition if we wish, as long as it implies the existing one (i.e. $\emptyset \vdash P \implies P'$)
  - Ex. we could use $\langle y = 6 \rangle$ as a precondition for $y + 1 = 7$, since $y = 6 \implies y + 1 = 7$ and $\langle y + 1 = 7 \rangle$ is a valid precondition
- **Postcondition Weakening**: If $\langle P \rangle C \langle Q' \rangle$ and we have $Q' \implies Q$, then we also have $\langle P \rangle C \langle Q \rangle$

- I.e. we can also weaken the postcondition in the same way

Total Correctness Problem

- Is a given Hoare triple $\langle P \rangle C \langle Q \rangle$ satisfied under total correctness
- This is an undecidable problem; we can reduce it to the blank tape halting problem
- The partial correctness problem is also undecidable