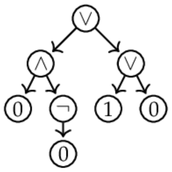
Lecture Notes
CS262 - Logic and Verification

Intro

1. **Arithmetic term** is either a variable $X, Y...$ or sum($\vee$), product($\wedge$) of other arithmetic terms. e.g. $(X \wedge Y) \vee Z$.
2. **Syntax** is the formal specification of a language, where no meaning is associated with symbols.
3. **Semantics** is the meaning of formal symbols in context.
4. **Proposition = Atomic formula = Statement** is an expression that has a truth value: e.g. "it's sunny today". **Propositional variable** x becomes a proposition once assigned a truth value ($x = T$).
5. **Propositional formula (PF)** comprises propositional variables and logical connectives. Its truth value depends on the assignment of those variables. Formulas representing same truth function are **logically equivalent**.
6. **Parse tree** for PFs can be inductively constructed with atomic formulas and connectives as nodes alike computation graphs. Subtrees are *subformulas.*
   Recursively define for formulas X, Y:
   - deg(A) = 0 if A is atomic formula
   - deg($\neg X$) = 1+deg(X)
   - deg($X \circ Y$) = 1+deg(X)+deg(Y)



7. **Theorem**: degree of a formula equals the number of inner nodes of the parse tree. Proof by induction
8. **Truth Table** specifies truth function between 2^n combinations of n variables to an output $f : \{T, F\}^n \to \{T, F\}$.
9. *Nullary/unary/binary* connective $\top/\neg/\wedge$ take 0/1/2 args.
10. **Valuation** v is mapping of PFs $\to \{T, F\}$ s.t. $v(\top) = T, v(\bot) = F$, $v(\neg X) = \neg v(X)$ and $v(X \circ Y) = v(X) \circ v(Y)$.
11. Formula evaluating to T under all/some/none valuations is called **tautology/satisfiable/contradicion**.
12. Formula X is a **consequence** of set S of formulas $S \vDash X$ if $\forall v(s_i \in S) = T \to v(X) = T$, or if all propositions in S are sufficient for X to be true ($\{p, p \to q\} \vDash q$). X is tautology iff $\varnothing \vDash X$, usually written as $\vDash X$ (unconditionally true).
13. Set of connectives is **complete** if can represent all 2^{2^n} truth func $\{T, F\}^n \to \{T, F\}$ using only such connectives.
14. Disjunctive/Conjunctive normal form **(DNF/CNF)** is a disj/conj of conj/disj of **literals** (var, $\neg$var, $\top$, $\bot$).
15. **Theorem**: every boolean function has a DNF and CNF.

Normal form algorithms

1. Let $X_1, ..X_n$ be sequence of propositional formulas. **Generalised disjunction** is *clause* $[X_1, ..X_n] := X_1 \vee ..X_n$. **Gen. conjunction** is *dual clause* $\langle X_1, ..X_n \rangle := X_1 \wedge ..X_n$.
2. **Neutral elements** of (dis/con)junction are valuations: *disj*: $v([]) = v(\bot) = F$ and *conj*: $v(\langle\rangle) = v(\top) = T$.
3. Group PFs of form $(X \circ Y)$ and $\neg(X \circ Y)$ with $\circ \in \{\wedge, \vee, \to, \leftarrow, \uparrow, \downarrow, \not\to, \not\leftarrow\}$ into conjunctive and disj. categories:

Conjunctive			*Disjunctive*		
α	α_1	α_2	β	β_1	β_2
$X \wedge Y$	X	Y	$\neg(X \wedge Y)$	$\neg X$	$\neg Y$
$\neg(X \vee Y)$	$\neg X$	$\neg Y$	$X \vee Y$	X	Y
$\neg(X \to Y)$	X	$\neg Y$	$X \to Y$	$\neg X$	Y
$\neg(X \leftarrow Y)$	$\neg X$	Y	$X \leftarrow Y$	X	$\neg Y$
$\neg(X \uparrow Y)$	X	Y	$X \uparrow Y$	$\neg X$	$\neg Y$
$X \downarrow Y$	$\neg X$	$\neg Y$	$\neg(X \downarrow Y)$	X	Y
$X \not\to Y$	X	$\neg Y$	$\neg(X \not\to Y)$	$\neg X$	Y
$X \not\leftarrow Y$	$\neg X$	Y	$\neg(X \not\leftarrow Y)$	X	$\neg Y$

   where $\forall v : v(\alpha) = v(\alpha_1) \wedge v(\alpha_2)$ and $v(\beta) = v(\beta_1) \vee v(\beta_2)$

4. **Expansion**: given PF X start $\langle [X] \rangle$ if CNF, $[\langle X \rangle]$ if DNF. Given current expansion D, select non-literal term N from some $D_i \in D$. α/β**-expansion** if N is α/β formula. Replace upper values of N with bottom values as follows:

$$\textbf{CNF}: \frac{\neg\top}{\bot} \; \frac{\neg\bot}{\top} \; \frac{\neg\neg Z}{Z} \; \frac{\beta}{\begin{matrix}\beta_1 \\ \beta_2\end{matrix}} \; \frac{\alpha}{\alpha_1 | \alpha_2} \quad \textbf{DNF}: \frac{\neg\top}{\bot} \; \frac{\neg\bot}{\top} \; \frac{\neg\neg Z}{Z} \; \frac{\beta}{\beta_1 | \beta_2} \; \frac{\alpha}{\begin{matrix}\alpha_1 \\ \alpha_2\end{matrix}}$$

   Proposition 1: this algorithm continuously produces a sequence of logically equivalent formulas.
5. **Theorem (Konig's lemma)**: finitely branching but infinite tree must have an infinite branch. Consider rooted tree, **branch** is sequence of nodes starting at root iteratively descending towards some child until none present (**finitely branching**). Tree/branch is **finite** if has finite num nodes, otherwise infinite.
6. Define **rank** of prop formula as $r([X_1, ..X_n]) = \sum_{i=1}^{n} r(X_i)$
   Recursion anchor $r(p) = r(\neg p) = 0$ for var $p, r(\top) = r(\bot) = 0; r(\neg\top) = r(\neg\bot) = 1$.
   Recursive step $r(\neg\neg Z) = r(Z) + 1; r(\alpha) = r(\alpha_1) + r(\alpha_2) + 1; r(\beta) = r(\beta_1) + r(\beta_2) + 1$.
7. **Proposition 2**: the algorithm terminates, regardless of which choices are made during the algorithm. **Proof**: assign curr conj of disj $\langle D_1, ..D_n \rangle$ a seq of n balls by placing a ball labelled $r(D_i)$ for each disj D_i into a box. At each step replace a ball by 2 if α-expand else 1, with lower rank. This game must end, so algorithm must terminate $\square$.

Proof Systems (PS)

1. **Semantic tableau** for DNF and **Resolution** for CNF are refutation systems taking X and aiming to arrive at contradiction **beginning at** $\neg X$. **Remember to negate!**
2. Both extend to first order logic (quantifiers), can be generalized to establish propositional consequences $S \vDash X$, not just tautologies $\vdash X$ and the rule application is non-deterministic in both.
3. **Theorem** tableau/resolution PS is **sound** ($\vdash_{t/r} X$ iff $\vDash X$)
4. **Theorem**: tableau/resolution PS is **complete** (tautology $\vDash X$ implies that strict tableau/resolution PS will terminate with a proof for it).
5. **Theorem**: For any set S of propositional formulas and any formula X, write $S \vDash X$ iff $S \vdash_t X$, $S \vdash_r X$ or $S \vdash_d X$

Semantic Tableau (DNF($\vee$))

1. **Semantic tableau** is a tree-like disjunction of conjunctive branches. Builds out every branch **all** of which need to reach contradiction, write $\vdash_t X$ if X has a tableau proof.
2. Tableau branch is **closed** if both formulas $X, \neg X$ or $\perp$ occur in the branch. If vars $x, \neg x$ appear, the branch is **atomically closed**. Tableau is (atomically) closed (proof succeeded) if **all** branches are (atomically) closed.
3. Tableau is **Strict** if no formula had an expansion rule applied to it twice on same branch. Represent tree as disj. of conj. Strictness removes expanded formula from the list, so **identical to DNF expansion**.
4. **S-introduction rule**: in $S \vdash_t X$ any formula $Y \in S$ can be added to end of any closed $\neg X$ tableau branch.

Propositional Resolution (CNF($\wedge$))

1. **Resolution** for CNF is just like Tableau, but only needs one contradiction. Identical to CNF expansion if strict. Write $\vdash_r X$ if X has a resolution proof.
2. **Strict** if every disjunction has at most 1 resolution expansion rule applied to it: no formula reuse, remove instead.
3. **Resolution rule**: for disj. D_1, D_2 s.t. $X \in D_1, \neg X \in D_2$: let $D = D_1 \backslash \{X\} \cup D_2 \backslash \{\neg X\}$ (remove and combine). If $\perp \in D$, delete all $\perp$'s occurrences, call D **trivial resolvent**

   Resolvent D is the result of **resolving** D_1 and D_2 **on** formula X. If X is atomic, then it's an **atomic** application of resolution rule.
4. **S-introduction rule**: in $S \vdash_r X$ for any formula $Y \in S$, can add the line $[Y]$ to closed $\neg X$ resolution expansion.

Natural Deduction

1. **Natural Deduction** has nested subordinate proofs (='lemmas') which draw conclusions from assumptions which are to be eventually exhausted.
2. **Implication rule**: If can derive Y from assumption X, then discharge X, conclude $X \to Y$ holds unconditionally.
3. Rule is **derived** if it doesn't strengthen proof system.

Derived Rules

Double negation

$$\frac{\neg\neg X}{X} \qquad \frac{X}{\neg\neg X}$$

Copy rule

$$\frac{X}{X}$$

Implication

$$\begin{array}{c}\boxed{\begin{array}{c}X\\ \vdots\\ Y\end{array}}\\ X \to Y\end{array}$$

Modus ponens

$$\frac{\begin{array}{c}X\\ X \to Y\end{array}}{Y}$$

Modus tollens

$$\frac{\begin{array}{c}\neg Y\\ X \to Y\end{array}}{\neg X}$$

Excluded middle

$$\frac{}{X \vee \neg X}$$

Implication Rules

Constant rules:

$$\frac{\perp}{X} \qquad \frac{}{\top}$$

Negation rules:

$$\frac{\begin{array}{c}X\\ \neg X\end{array}}{\perp} \qquad \begin{array}{c}\boxed{\begin{array}{c}X\\ \vdots\\ \perp\end{array}}\\ \neg X\end{array} \qquad \begin{array}{c}\boxed{\begin{array}{c}\neg X\\ \vdots\\ \perp\end{array}}\\ X\end{array}$$

Primary connective rules:

$$\alpha\mathbf{E} \quad \frac{\alpha}{\alpha_1} \quad \frac{\alpha}{\alpha_2} \qquad \alpha\mathbf{I} \quad \frac{\begin{array}{c}\alpha_1\\ \alpha_2\end{array}}{\alpha}$$

$$\beta\mathbf{E} \quad \frac{\begin{array}{c}\neg\beta_1\\ \beta\end{array}}{\beta_2} \quad \frac{\begin{array}{c}\neg\beta_2\\ \beta\end{array}}{\beta_1} \qquad \beta\mathbf{I} \quad \begin{array}{c}\boxed{\begin{array}{c}\neg\beta_1\\ \vdots\\ \beta_2\end{array}}\\ \beta\end{array} \quad \begin{array}{c}\boxed{\begin{array}{c}\neg\beta_2\\ \vdots\\ \beta_1\end{array}}\\ \beta\end{array}$$

4. Primary connective rules are: Introduction and Elimination. Last two negation rules determine if it's a contradiction proof (can start with $\neg X$ and aim to arrive at contradiction $\perp$). Second constant rule has no premises. Order of premises doesn't matter, but all must be active.
5. S-**introduction** rule for natural deduction: at any stage, any **premise** $S_i \in S$ may be used as a line. Write $S \vdash_d X$ if $\exists$ natural deduction derivation of X to S. E.g. $\{p \to q, q \to r\} \vdash_d p \to r$. Show $p \to q \to r$.

SATisfiability

1. **SAT Problem**: F Given a propositional formula in CNF, is there a satisfying assignment for it? NP-complete. Can solve k-SAT problem of L literals in $2^k \cdot L$.
2. Positive/negative **literal** is a variable / negated variable. **Clause** is disj of literals. k-SAT problem takes k-CNF formulas, where every clause has at most k literals.

3. **Theorem**: given SAT instance F $\exists$ polynomial time algorithm that produces $G(F)$ 3-CNF s.t. F is satisfiable iff $G(F)$ is satisfiable. Efficient 3-SAT $\Rightarrow$ efficient SAT.
   Proof: consider clause containing literals $[X, Y]$, replace $X \vee Y$ by new variable Z (1 literal fewer), express $X \vee Y \equiv Z$ by 3-CNF $F(X, Y, Z) : [...]$. Conjunctively connect it to current CNF: $F(X, Y, Z) \wedge \text{CNF}$. Repeat if $|clause| > 3$.$\square$

4. **Corollary**: efficient SAT $\Rightarrow$ efficient k-Colouring.
   Proof: $\forall v \in V$ and each colour k introduce variable $x_{v,k} = 1$ if v receives k else 0. Add constraint that $\forall v \in V$ receives exactly one colour, and $\forall(a, b) \in E : k_a \neq k_b$ (edge end vertices have diff colours). Num variables $= |V| \times k$, so runs in polynomial of $|V|, |E|$.

5. Can solve 2-SAT in linear time by exhaustively applying resolution rule. Each resolution produces clause of size ≤ 2, so at most $1 + 2n + 4\binom{n}{2} = 2n^2 + 1$ clauses can occur where n is num of variables. If empty clause $[]$ occurs, then non-satisfiable, else satisfiable.

6. **Directed graph** $D = D(F)$ can capture $u \vee v \equiv \neg u \rightarrow v \equiv \neg v \rightarrow u$ implication. Write $x \rightsquigarrow y$ if $\exists$ a directed path from x to y in $D(F)$ of SAT instance F.
   Vertex set $V(D) = V \cup \overline{V}$ (all vars $V = V(F)$, their negation). Have $2n$ vertices where $n = |V|$.
   Edge set $E(D){=}\{(\neg u, v), (\neg v, u) : [u{\vee}v] \in F\}{\cup}\{(\neg u, u) : [u] \in F\}$ - 2-clauses lead to two directed edges, a unit clause leads to one. Have $\leq 2m$ edges, where $m = |F|$.

7. **Lemma**: Given graph $G(F) = (V, E)$, F is not satisfiable iff $\exists x \in V$ s.t. $x \rightsquigarrow \neg x \rightsquigarrow x$ (**strongly connected**).
   Proof: let F' be CNF obtained from F by exhaustively applying resolution. Resolvent $[u, v]$ of $[x, y]$ and $[\neg x, v]$ (x and $\neg x$ cancel out) would add edges $\neg u \rightarrow v$ and $\neg v \rightarrow u$, but $\neg u \rightarrow x \rightarrow v$ and $\neg v \rightarrow \neg x \rightarrow u$ already present, so relation $\rightsquigarrow$ isn't altered. Then the following are equivalent: F not satisfiable $\Leftrightarrow$ $[] \in F'$ (by definition of resolution) $\Leftrightarrow$ $[x], [\neg x] \in F'$ (cancels out to $[]$) $\Leftrightarrow$ $x \rightarrow \neg x \rightarrow x \in D(F')$ (edge set of disj. from 6) $\Leftrightarrow$ $x \rightsquigarrow \neg x \rightsquigarrow x \in D(F)$.

8. Subset $G \subseteq F$ of 3-CNF over n vars is **independent** if no clauses share variables, e.g. $\langle[a, b], [\neg c, d]\rangle$. G is **maximal** if independency breaks upon adding any new clause.

9. **Lemma**: given maximal set G of independent 3-clauses in 3-CNF F have: $|G| \leq n \div 3$. For any truth assignment α in G, formula $F^{[\alpha]}$ obtained from assigning all vars in α to F and removing clauses with true literals and false literals from clauses, is 2-CNF. So, there are $7^{|G|} \leq 7^{n/3}$ satisfying assignments for G. **PROOF TBC**

10. **Theorem**: satisfiability of 3-CNF formula can be decided in $O(7^{n/3}\text{poly}(n)) = O(1.913^n)$ time.

SAT Solving

1. **Horn clause** contains ≤ 1 positive literal (non-negated). **Horn CNF** has only Horn clauses $[\neg x, \neg y, a]{\equiv}(x{\wedge}y) \rightarrow a$.

2. **Theorem**: the following algorithm decides satisfiability of Horn CNF in linear time.

Horn CNF satisfiability $O(n)$

```
F ← Horn CNF # a :- x,y. means [¬x,¬y,a]≡(x ∧ y) → a
while ([] ∉ F) { # empty clause [] isn’t satisfiable
 # ⟨⟩: any assignment, size ≥ 2: set all to false.
 if F=⟨⟩ or every F-clause has size≥2: return ”Yes”
 pick clause [u] ∈ F of size=1 # set clause to true
 remove clauses with u from F and ¬u from clauses}
```

3. **Complete methods** find satisfying assignment of proof that none exists (systematic solvers). **Incomplete methods** don't guarantee results, use stochastic local search.

4. $F|l \equiv$ remove clauses with l, and $\neg l$ from clauses, $l := T$.

Naïve Backtracking for SAT

```
def Back(F) → satisfying assignment or ⊥:
  if F = ⟨⟩: return ∅
  elif [] ∈ F: return ⊥
  else: # try assigning a literal to True or False
    Let l be literal in F; L := Back(F|l)
    if L ≠ ⊥: return L ∪ {l}; # try l := ⊤
      else: L := Back(F|¬l)
    if L ≠ ⊥: return L ∪ {¬l}; # if fail, l := ⊥
      else: return ⊥
```

5. **Unit clauses** $[l]$ force **unit clause propagation**: $l := T$. Also have **pure literals**, or l s.t. $\neg l$ doesn't appear in current formula, so can set $l := T$. Optimise using:

UnitPure(F)

```
def UnitPure(F) → partial assignment:
  L := ∅; F’ := F
  while F’ has unit-clause [l] or pure-literal l:
    L := L ∪ {l} # assign l to true
    F’ := F’|l # remove l-clauses and ¬l from clauses
```

Davis-Putnam-Logemann-Loveland

```
def DPLL(𝒢) → satisfying assignment or ⊥:
  U := UnitPure(𝒢); F := 𝒢 | U # optimisation
  if F = ⟨⟩: return ∅; elif [] ∈ F: return ⊥
  else: # try assigning a literal to True or False
    Let l be literal in F; L := DPLL(F|l)
    if L ≠ ⊥: return 𝒢 ∪ L ∪ {l}; # try l := ⊤
      else: L := DPLL(F|¬l)
    if L ≠ ⊥: return 𝒢 ∪ L ∪ {¬l}; # if fail, l:=⊥
      else: return ⊥
```

Usually optimised further using ***heristics***: ”Which literal l to choose in next recursion step?”

6. **Static heuristics**: linear ordering of variables fixed before start, fast to compute.
   Dynamic heuristics: order based on current formula F, typically from # occurrences of literals in F's clauses. e.g.
   • **Dynamic Largest Individual Sum (DLIS)**: choose literal which occurs most frequently.
   • **Max Occurrence in Clauses of Min Size (MOMS)**: literal occurring most frequently in clauses of min size.

7. Clauses only matter during search when going from:
   2 non-false literals to 1 (unit clause propagation), or
   1 non-false literal to 0 (conflict).

8. So, choose 2 **watched literals** in each clause. **Invariant**: watched literals are non-false (either true or not assigned) if clause is not satisfied.

9. Each literal l has a watch list $W(l)$ of clauses watching l. When l is falsified, visit every clause $C \in W(l)$:
   • ***some literal is true***, continue (don't track satisfaction of clauses explicitly).
   • ***all literals false***, return (backtrack).
   • ***all but 1*** $l' \in C$ ***are false***, assign $l' :=$ true (unit clause propagation), continue.
   • ***else***, add C to watch list of one of its remaining unassigned literals, remove from watch list $W(l)$.

10. Each disjunctive clause $[a_1, ..., a_n]$ wants to be satisfied - a single a_i set to True is enough. But watching all n literals is slow, choose 2 non-false (true/unassigned) **watch literals** per clause. Consider clause $A = [x, y, z]$; watch(A) $= \{x, y\}$. Now, if x gets assigned False somewhere else:
    - try to watch z
      z is ***True/unassigned***: watch$(A) := \{y, z\}$; continue
      z is ***False***: can't watch it;
      - check on y:
        y is ***True***; watch(A) stays $\{x, y\}$; continue
        y is ***unassigned***; $A = [F, y, F]$ is now a unit clause;
        unit propagate $y :=$ True; continue
        y is ***False***, then A is False; backtrack

11. **Benefits of watched literals**: no updates on watch lists upon backtracking, fewer clauses inspected when literal is set; once a literal is assigned false, it becomes unwatched in many clauses, so faster subsequent reassignment.

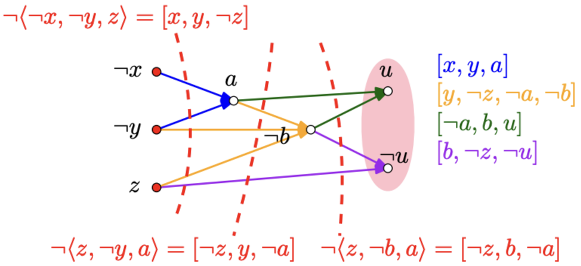
12. Backtracking doesn't need watch lists to update as it only un-assigns values, never falsifies a literal to maintain the invariant; use **lazy data structures**.

13. **Clause learning**: when a conflict is found, identify a minimal subset of the current assignment (conflict clause) that caused it, and add it as a new clause to prevent the solver from repeating the same mistake in other branches.

14. **Clause learning example**: assume reason for conflict in branch $\langle x, \neg y, z\rangle$ is $\langle x, \neg y\rangle$, so add $\neg(x, \neg y) = [\neg x, y]$ clause, preventing other branches from this conflict.

15. **Implication Graph** G is digraph associated with stages of the algorithm. Nodes are **decision literals** l, or literals set to true. For any clause $C = [l_1, ..l_k, l]$, where $\neg l_1, ..\neg l_k$ are nodes, add new node l and edges from $\neg_i \rightarrow l$ corresponding to C for all $i = 1, ..k$.

16. **Conflict literal** l if both l and $\neg l$ appear as nodes in the implication graph. **Conflict graph** $G' \subseteq G$ contains:
    • Exactly one conflict literal l (i.e., both l and $\neg l$ as nodes),
    • Only nodes that have a path to l or $\neg l$ in G,
    • For each node in G', only the incoming edges that come from a **single clause** (implication responsible for it).



17. Edge cut in G' with all decision literals on **reason side** R, and the conflict literals on the **conflict side** C. All edges across the cut are $R \rightsquigarrow C$.

18. **Conflict clause** takes disjunction of all literals l' between edges $l \rightarrow l'$ with $l \in R, l' \in C$. Use **learning schemes** to select one conflict clause to add.

19. **Lemma**: Any conflict clause can be inferred by resolution rule from existing clauses. **PROOF TBC**.
    Consequently, adding conflict clauses doesn't change satisfiability of input formula.

20. **Conflict-driven clause learning (CDCL) algorithm**:
    1. Select variable and assign true/false
    2. Apply unit clause propagation
    3. Build implication graph
    4. If conflict:
       • derive & add corresponding conflict clause to formula.
       • non-chronologically backtrack to decision level where first-assigned variable involved in conflict was assigned.
    5. Repeat from step 1 until all variable values are assigned.

21. **Monte Carlo method**: start with random truth assignment, while $\exists$ unsatisfying clauses, pick any one, flip a randomly chosen literal in it. Give up after $\leq N$ trials. Only works on satisfiable formulas, can't prove otherwise.

22. **Theorem**: for satisfiable 3-SAT formula with n variables, this algorithm will succeed with probability $\Omega((3/4)^n/n)$ after at most $\leq N$ rounds. So, repeat $Kn(4/3)^n$ times.

First-order Logic

1. Propositional logic has limited expressive power (only true/false possible), can't express things like: every natural number has a successor; $\forall x \ \exists \ x \leq$ prime $\leq 2x$ etc.

2. First-order language comprises:
   Propositional connectives $(\wedge, \vee, \rightarrow, \text{etc})$,
   Constants $(\top, \bot)$,
   Quantifiers $\forall, \exists$,
   Variables x, y, z,
   Relation symbols: $>, =, p(x)$ (is x prime) etc.,
   Function symbols: $\text{succ}(x)$ (successor of x), $x + y$ etc.
   Constant symbols: $1, 2, 3 \ldots$.

3. **First-order language** $L(R, F, C)$ has finite/countable sets: R of relation/predicate symbols, F of function symbols each with some number of arguments, and C of constant symbols (function symbols with 0 arguments).

4. **Family of terms** of $L(R, F, C)$ is the smallest set s.t.:
   1. every variable is a term,
   2. $\forall c \in C$ is a term,
   3. $\forall f \in F$ with n args $t_1, .., t_n$; - $f(t_1, ..t_n)$ is also a term.
   Terms allow us to iteratively use any constant and function symbols, but no relation symbols.

5. **Atomic formula** of $L(R, F, C)$ is a string $r(t_1, ..t_n)$ where $r \in R$ taking n arguments $t_1, ..t_n$. E.g. $\top, \bot$ are atomic.

6. **Family of formulas** of $L(R, F, C)$ is smallest set s.t.:
   1. every atomic formula is a formula of $L(R, F, C)$,
   2. if A is formula, then so is $\neg A$,
   3. if A, B formulas then $A \circ B$ is also a formula for any binary connective $\circ$,
   4. if A is a formula and x is a variable, then $(\forall x A)$ and $(\exists x A)$ are also formulas.

7. **Free-variable occurrences** in a formula are those not bound by quantifiers:
   1. All variable occurrences in atomic formulas are free.
   2. In $\neg A$, same free variables as in A.
   3. In $A \circ B$ (e.g., $\wedge$, $\vee$, $\rightarrow$), free vars are those in A, B.
   4. In $\forall x \, A$ or $\exists x \, A$, all free variables in A except for x.

   It's about **occurrences**, not vars: in $Q(x) \rightarrow \forall x R(x)$: x is both free in $Q(x)$ and **bound** in $\forall x R(x)$.

8. **Sentence** or closed formula of $L(R, F, C)$ is formula with no free-variable occurences.

9. **Model** $M = (D, I)$ for first-order language $L(R, F, C)$: where set D is a nonempty **domain** of M, and I is the **interpretation** (mapping) that associates:
   1. some member $c' \in D$ to $\forall c \in C$,
   2. some n-any func $f' : D^n \rightarrow D$ to $\forall f \in F$ with n args,
   3. some n-ary relation $R' \subseteq D^n$ to $\forall r \in R$ with n args.

10. **Assignment** in model $M = (D, I)$ is mapping A from set of vars to set D. Write x^A for image of x under A.

11. **Values of terms**: for model $M = (D, I)$, lang $L(R, F, C)$, assignment A, **associate** value $t^{I,A}$ to each lang term t s.t.:
    1. for $c \in C$, have $c^{I,A} = c^I$,
    2. for $x \in X$, have $x^{I,A} = x^A$,
    3. for $f \in F$, have $(f(t_1, ..t_n))^{I,A} = f^I(t_1^{I,A}, ...t_n^{I,A})$.
    for constants c, vars x and function symbols f with n args.

12. Associate **truth value** $\Phi^{I,A}$ to each formula of $L(R, F, C)$:
    1. $R(t_1, ..t_n)^{I,A}{=}T$ iff $(t_1^{I,A}, ..t_n^{I,A}) \in R'$; $\quad \top^{I,A}{=}T, \bot^{I,A}{=}F$
    2. $[\neg X]^{I,A} = \neg[X^{I,A}]$
    3. $[X \circ Y]^{I,A} = X^{I,A} \circ Y^{I,A}$
    4. $[\forall x \Phi]^{I,A}{=}T$ and $[\exists x \Phi]^{I,A}{=}T$ iff $\Phi^{I,A'} = T$ for every A' differing from A in at most the value assigned to x.
    for model $M = (D, I)$ and assignment A.

13. **1.** Formula Φ of lang $L(R, F, C)$ is **true in the model** $M = (D, I)$ if $\Phi^{I,A} = T$ for all assignments A.
    2. Φ is **valid** if $\Phi = T$ in all models for the language.
    3. Set S of formulas is **satisfiable** in M if there is an assignment A s.t. $\forall \Phi \in S : \Phi^{I,A} = T$.
    Can turn arbitrary formulas into sentences by universally quantifying away free variables.

First-order proof systems

1. For quantified formulas, γ-formulas act universally ($\forall$) and δ-formulas act existentially ($\exists$). Below, $\Phi\{x/t\}$ denotes formula obtained from Φ by substituting free occurrences of variable x by term t. Useful to expand δ before γ, as δ introduces variables, but γ chooses from existing ones.

γ	$\gamma(t)$	δ	$\delta(t)$
$\forall x \Phi$	$\Phi\{x/t\}$	$\exists \Phi$	$\Phi\{x/t\}$
$\neg \exists \Phi$	$\neg \Phi\{x/t\}$	$\neg \forall \Phi$	$\neg \Phi\{x/t\}$

2. Given $L(R, F, C)$ let **par** be countable set of constant symbols disjoint from C. Its elements are **parameters**, write $L^{\text{par}} \equiv L(R, F, C \cup \text{par})$.

3. In **δ-γ-expansion**, let γ take some closed term t of L^{par} (closed=no variables), and let δ-expansion introduce new parameter p that has not been used prior in the proof.

4. **Strictness**: if have γ on branch, should be allowed to add $\gamma(t_1)$ and later $\gamma(t_2)$ where t_1, t_2 are different closed terms.

5. **First-order Tableau** $\frac{\gamma}{\gamma(t)}, \frac{\delta}{\delta(p)}$. Still non-deterministic, unlike in propositional, can work forever never reaching closed tableau even if one exists.

6. **First-order Resolution** $\frac{\gamma}{\gamma(t)}, \frac{\delta}{\delta(p)}$. Else same as tableau.

7. **First-order Natural deduction**: $\gamma E \ \frac{\gamma}{\gamma(t)}, \frac{\delta}{\delta(p)}$.

Program verification $P \vdash \phi$

1. P is a program and ϕ is a property about that program established by deductive proof. Not decidable but partial automation is possible. Detects functional bugs in small programs performing identifiable tasks.

2. **Aims**: define small programming language, describe logical framework which allows logical properties to be derived from the program by deductive proof. Then for program P and important properly ϕ, set out a proof to show that P terminates establishing ϕ. The following are equivalent:

   `x=0; i=1; while (i≤n) {x=x+arr[i]; i=i+1}`

   $$\vdash x = \sum_{i=1}^{n} arr[i]$$

3. **Syntax domains** integer(E), boolean(B), commands(C):
   1. Assignment statement $x = E$.
   2. Composition (sequential): $C1; C2$: run $C1$ then $C2$.
   3. Conditional: `if` B `then` $\{C1\}$ `else` $\{C2\}$.
   4. Loop: `while` $B\{C\}$.

4. **Postcondition**: property that must hold upon program termination. First order logic formula referring to program variables, expressing desired conditions formally.

5. **Precondition**: property we want to program variables to satisfy before the program starts. Also a first order logic formula. Can be $\top$ (no condition in precondition).

6. **Hoare triples**: `(|Pre|) Prog (|Post|)` logical statement:
   1. Pre: precondition that we can assume holds
   2. Prog: program itself
   3. Post: postcondition we wish to establish.
   If the Hoare triple is valid, then any execution of `Prog` starting in a state where `Pre` holds will end in a state where `Post` holds

7. When need to refer to **original values**, often use 0 subscript e.g. x_0, y_0; can swap values of x, y with:
   $(|x = x_0 \wedge y = y_0|)\ t = x;\ x = y;\ y = t;\ (|x = y_0 \wedge y = x_0|)$

8. `Pre, Post` form **program specification**, but require proof.

9. **Weakest precondition** $wp(P, Post)$ for program P to establish postcondition `Post` is precondition implied by any other possible precondition, guaranteeing that `Post` holds.

   e.g. $wp(x = x+1, x > 3) = \underline{x > 2}$

   min requirement for P to successfully reach goal `Post`.

10. For **assignments**, formula will be true afterwards exactly when it holds beforehand with the new value:

    $$\boxed{wp(x = E, Post) = Post[x/E]}$$

    where $Post[x/E]$ is the condition obtained from `Post` by replacing x by E (**substitution**).

11. Weakest precondition for **composition**:

    $$\boxed{wp(P; Q, Post) = wp(P, wp(Q, Post))}$$

12. Weakest precondition for **conditional**:

    $$\begin{aligned} & wp(\text{if } B \text{ then } \{C_1\} \text{ else } \{C_2\}, Post) = \\ &= (B \to wp(C_1, Post)) \wedge (\neg B \to wp(C_2, Post)) \\ &= \boxed{(B \wedge wp(C_1, Post)) \vee (\neg B \wedge wp(C_2, Post))} \end{aligned}$$

13. Any program is a sequence of instructions $Prog = C_1; C_2; ..; C_n$, lay out proof for $Prog$ as shown on the right. Validity of each of the Hoare triples $(|\phi_{i-1}|)C_i(|\phi_i|)$ must be inferred from some rule, allowing us to deduce $(|\phi_0|)Prog(|\phi_n|)$.

    $$\begin{array}{ll} & (|\phi_0|) \\ C_1 & \\ & (|\phi_1|) \\ \cdots & \\ & (|\phi_{n-1}|) \\ C_n & \\ & (|\phi_n|) \end{array}$$

14. **Assignment rule** allows the Pre to just be $Post$ with syntactic substitution: $(|x+1 > x_0|)\ x = x+1\ (|x > x_0|)$, but can only deduce a triple if Pre is exactly the WP.

    $$\boxed{\frac{}{(|Post[x/E]|)x = E(|Post|)}\ \text{Assignment}}$$

15. Can ***strengthen*** Pre: e.g. $x = 5 \to x+1 > 1$, so can do:
    $(|x = 5|)\ \underline{x = x+1}\ (|\underline{x > 1}|) \to (|\underline{x+1 > 1}|)\ x = x+1\ (|x > 1|)$
    and ***weaken*** $Post$: e.g. $x > 1 \to x \neq 0$, so can do:
    $(|x{+}1 > 1|)\ x = x{+}1\ (|x > 1|) \to (|x{+}1 > 1|)\ x = x{+}1\ (|x \neq 0|)$

16. **Implied rule**: combines strengthening and weakening and means that "upper" formula implies the "lower" one.

    $$\boxed{\frac{Pre \to P \quad (|P|)Prog(|Q|) \quad Q \to Post}{(|Pre|)Prog(|Post|)}\ \text{Implied}}$$

17. **Composition rule**:

    $$\boxed{\frac{(|Pre|)Prog_1(|Mid|) \quad (|Mid|)Prog_2(|Post|)}{(|Pre|)Prog_1; Prog_2(|Post|)}\ \text{Composition}}$$

18. When constructing the proof, we "push" $Post$ upwards to see what is needed for command to achieve desired result.

19. **Conditional rule**: `if B {C₁} else {C₂}`. Push $Post$ backwards through respective code, in other words, calculate $wp(C_1, Post)$ and $wp(C_2, Post)$.

    $$\boxed{\frac{(|Pre \wedge B|)C_1(|Post|) \quad (|Pre \wedge \neg B|)C_2(|Post|)}{(|Pre|) \text{ if } B\{C_1\} \text{ else } \{C_2\}(|Post|)}\ \text{Conditional}}$$

20. **Loop rule**: `while B {C}` where L is **loop invariant**, holds before and after each iteration (even last one). Need to find a good loop invariant.

    $$\boxed{\frac{(|B \wedge L|)C(|L|)}{(|L|) \text{ while } B\{C\}(|\neg B \wedge L|)}\ \text{Loop}}$$

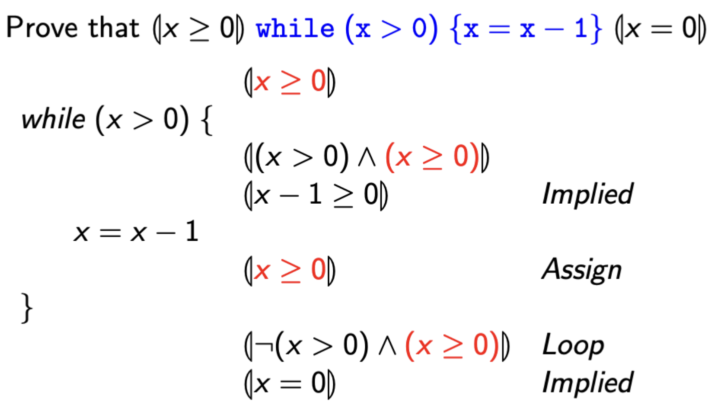
Other notation & Examples

1. **Equational reasoning**: Laws of Boolean algebra can be used to simplify complex formulas.

2. **CNF expansion** of $\neg(p \wedge \neg\bot) \vee \neg(\top \uparrow q)$:
   $\langle[\neg(p \wedge \neg\bot) \vee \neg(\top \uparrow q)]\rangle \equiv$ conj ”$\langle\rangle$” of disj ”[]”
   $\langle[\neg(p \wedge \neg\bot)], [\neg(\top \uparrow q)]\rangle \equiv$ ”$\vee$” is disj, so ”β_1, β_2”
   $\langle[\neg p, \neg\neg\bot, \top, q]\rangle \equiv$
   $\langle[\neg p, \bot, q]\rangle \equiv$
   $\underline{\langle[\neg p, q]\rangle}$ ☐

3. **DNF exp.** of $(x \rightarrow y) \rightarrow (y \downarrow \neg z) \equiv$
   $[\langle(x \rightarrow y) \rightarrow (y \downarrow \neg z)\rangle] \equiv$ disj ”[]” of conj ”$\langle\rangle$”
   $[\langle\neg(x \rightarrow y) \vee (y \downarrow \neg z)\rangle] \equiv$ expand ”$\rightarrow$”
   $[\langle\neg(x \rightarrow y)\rangle, \langle(y \downarrow \neg z)\rangle] \equiv$ ”$\vee$” is disj, so ”β_1, β_2”
   $[\langle\neg(\neg x \vee y)\rangle, \langle(\neg y \wedge \neg\neg z)\rangle] \equiv$ expand inner ”$\rightarrow$” and ”$\downarrow$”
   $[\langle(\neg\neg x \wedge \neg y)\rangle, \langle(\neg y \wedge \neg\neg z)\rangle] \equiv$ distribute ”$\neg$”
   $[\langle x \wedge \neg y\rangle, \langle\neg y \wedge z\rangle] \equiv$ cancel ”$\neg\neg$”
   $\underline{[\langle x, \neg y\rangle, \langle\neg y, z\rangle]}$ ☐ $\wedge$ is conj, so ”α_1, α_2”

4. **Resolution S-introduction** $\{p \rightarrow q, q \rightarrow r\} \vDash \neg(\neg r \wedge p)$:
   *Use (**introduce**) LHS formulas when need to resolve*
   0. $[\neg\neg(\neg r \wedge p)]$ negation of main formula
   1. $[\neg r \wedge p]$ cancel out $\neg\neg$
   2. $[\neg r]$ from 1
   3. $[p]$ from 1
   4. $[p \rightarrow q]$ s-introduciton from left side of statement
   5. $[\neg p, q]$ expand ”$\rightarrow$” from 4
   6. $[q]$ resolution on p from 3 and $\neg p$ from 5
   7. $[q \rightarrow r]$ s-introduction
   8. $[\neg q, r]$ expand ”$\rightarrow$” from 7
   9. $[r]$ resolution on q from 6 and $\neg q$ from 8
   10. $[]\bot$ resolution on $\neg r$ from 2 and r from 9.



Prove that $(\!|x \geq 0|\!)$ `while (x > 0) {x = x - 1}` $(\!|x = 0|\!)$

```
                                  $(|x \geq 0|)$
while $(x > 0)$ {
                                  $(|(x > 0) \wedge (x \geq 0)|)$
                                  $(|x - 1 \geq 0|)$               Implied
      $x = x - 1$
                                  $(|x \geq 0|)$                   Assign
}
                                  $(|\neg(x > 0) \wedge (x \geq 0)|)$   Loop
                                  $(|x = 0|)$                      Implied
```

5.

Prolog

1. Unknown variables must be capitalised or start with ”_”. Comma ”,” is same as $\wedge$, semicolon ”;” is same as $\vee$. :- is reverse implication $\leftarrow$ or ”if” e.g. $a(X)$:-$b(X)$ means that a is true if b is true.

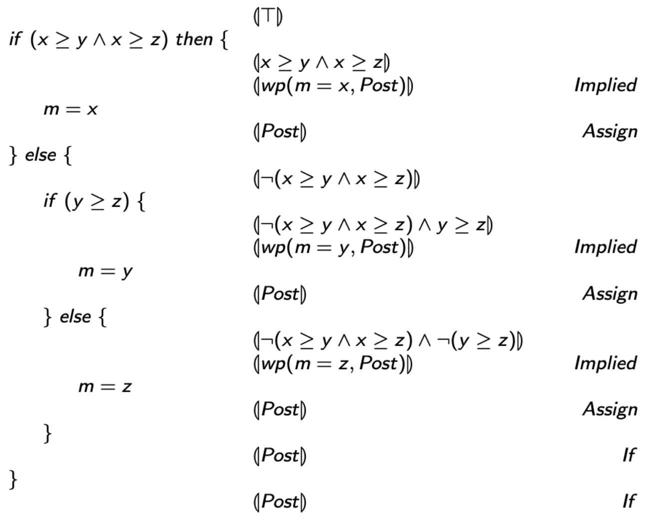
Basic Prolog Syntax

```
indian(curry).         % assert single variable fact
likes(john, sushi).    % asserts a fact
?- likes(john, sushi). % query/question a fact
likes(john, X) :- likes(mary, X) % bind facts (←)
```

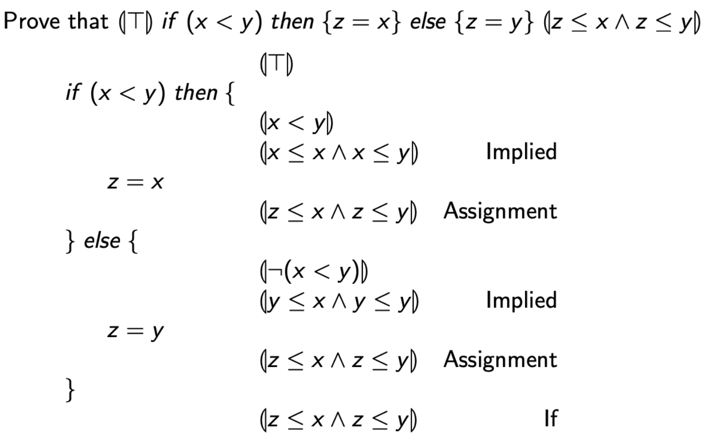
Lists

```
[a, b, c] % a list
[a, b, [c, d]] % list in lists
[Head | Tail] % access first element and the rest
[H1 | [H2|Tail]] = [H1, H2|Tail] % iterate list
member (E1, List) % test membership
nth1 (Idx, List, E1) % get nth element
_ % placeholder for arbitrary expression
! % backtracking cut operator
```

2. Verification Conditional proof:
   $(|\top|)\ P\ (|m \geq x \wedge m \geq y \wedge m \geq z \wedge (m = x \vee m = y \vee m = z)|)$



```
                                        $(|\top|)$
if $(x \geq y \wedge x \geq z)$ then {
                                        $(|x \geq y \wedge x \geq z|)$
                                        $(|wp(m = x, Post)|)$                       Implied
    $m = x$
                                        $(|Post|)$                                  Assign
} else {
                                        $(|\neg(x \geq y \wedge x \geq z)|)$
    if $(y \geq z)$ {
                                        $(|\neg(x \geq y \wedge x \geq z) \wedge y \geq z|)$
                                        $(|wp(m = y, Post)|)$                       Implied
        $m = y$
                                        $(|Post|)$                                  Assign
    } else {
                                        $(|\neg(x \geq y \wedge x \geq z) \wedge \neg(y \geq z)|)$
                                        $(|wp(m = z, Post)|)$                       Implied
        $m = z$
                                        $(|Post|)$                                  Assign
    }
                                        $(|Post|)$                                  If
}
                                        $(|Post|)$                                  If
```



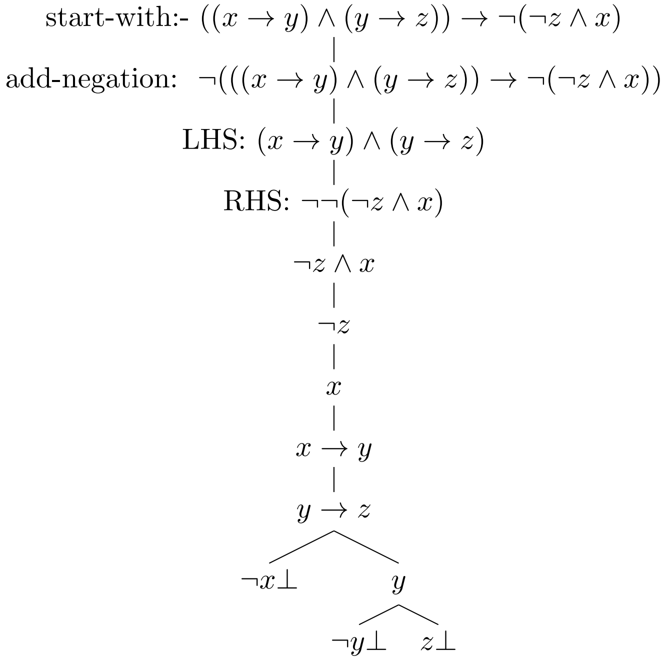
Prove that $(|\top|)$ *if* $(x < y)$ *then* $\{z = x\}$ *else* $\{z = y\}$ $(|z \leq x \wedge z \leq y|)$

```
                              $(|\top|)$
    if $(x < y)$ then {
                              $(|x < y|)$
                              $(|x \leq x \wedge x \leq y|)$        Implied
        $z = x$
                              $(|z \leq x \wedge z \leq y|)$    Assignment
    } else {
                              $(|\neg(x < y)|)$
                              $(|y \leq x \wedge y \leq y|)$        Implied
        $z = y$
                              $(|z \leq x \wedge z \leq y|)$    Assignment
    }
                              $(|z \leq x \wedge z \leq y|)$               If
```

3.

Propositional Logic Proof Systems

1. **Tableau Proof**: DNF, OR: branch; AND: add line.



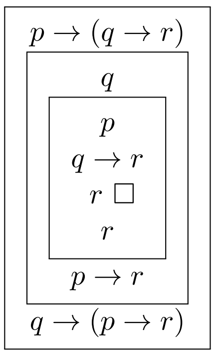
2. **Resolution Proof**: CNF; OR: add ","; AND: add line.

start with: $((x \to y) \wedge (y \to z)) \to \neg(\neg z \wedge x)$

0) $[\neg(((x \to y) \wedge (y \to z)) \to \neg(\neg z \wedge x))]$	negate, add "[]"
1) $[(x \to y) \wedge (y \to z)]$	LHS
2) $[\neg z \wedge x]$	RHS
3) $[x \to y]$	by 1
4) $[y \to z]$	by 1
5) $[\neg z]$	by 2
6) $[x]$	by 2
7) $[y]$	resolution on 3,6
8) $[z]$	resolution on 4, 7
9) $[]\bot$	resolution on 5,8

3. **Natural Deduction Proof**:

Imp = implication (→); MP = Modus-Ponens rule

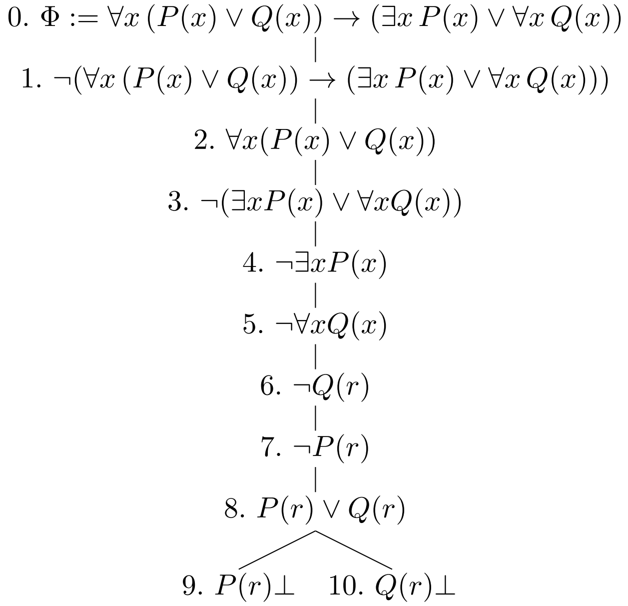


Imp LHS_1
Imp LHS_2
Imp LHS_3
MP
MP
Imp RHS_3
Imp RHS_2
Imp RHS_1

$(p \to (q \to r)) \to (q \to (p \to r))$ Formula to prove

First-order Proof Systems

1. **First Order Logic Tableau Proof**:



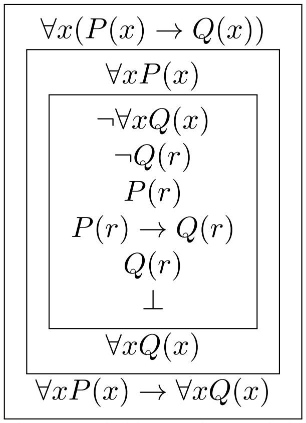
1) add negation	7) γ-expand 4, $\{x/r\}$
2,3) α-expand 1	8) γ-expand 2, $\{x/r\}$
4,5) α-expand 2	9,10) β-expand 8
6) δ-expand 5, $\{x/r\}$	

2. **First Order Logic Resolution Proof**:

0) $\Phi := \forall x(P(x) \vee Q(x)) \to (\exists x P(x) \vee \forall x Q(x))$

1) $[\neg(\forall x(P(x) \vee Q(x)) \to (\exists x P(x) \vee \forall x Q(x)))]$	negate
2) $[\forall x(P(x) \vee Q(x))]$	α-expand 1
3) $[\neg(\exists P(x) \vee \forall x Q(x))]$	α-expand 1
4) $[\neg\exists P(x)]$	α-expand 3
5) $[\neg\forall x Q(x)]$	α-expand 3
6) $[\neg Q(r)]$	δ-expand 5, $\{x/r\}$
7) $[P(r) \vee Q(r)]$	γ-expand 2, $\{x/r\}$
8) $[P(r), Q(r)]$	β-expand 7
9) $[P(r)]$	resolve 6,8
10) $[\neg P(r)]$	γ-expand 4, $\{x/r\}$
11) $[]\bot$	resolve 9,10.

3. **First Order Logic Natural Deduction Proof**:



FTSOC
δ-exp $\{x/r\}$
γ-exp $\{x/r\}$
γ-exp $\{x/r\}$
MP
Neg rule
Neg rule
Imp rule

$\Phi := \forall x(P(x) \to Q(x)) \to (\forall x P(x) \to \forall x Q(x))$

General Logic

1. $\neg$ negation (NOT)
2. $\wedge$ conjunction (AND)
3. $\vee$ disjunction (OR)
4. $\oplus$ exclusive or (XOR)
5. $\uparrow$ (NAND)
6. $\downarrow$ (NOR)
7. $\in$ membership
8. $\rightarrow$ implication
9. $\equiv$ or $\leftrightarrow$ equivalence
10. $\top$ "top" always returns T
11. $\perp$ "bottom" always returns F
12. $\exists$ existential quantifier ("exists")
13. $\forall$ universal quantifier ("forall")

$A \rightarrow B$: A is sufficient for B, B is necessary for A
$A \leftrightarrow B$: A/B is necessary and sufficient for B/A

Theorems & Rules

1. $A \rightarrow B \equiv \neg B \rightarrow \neg A \equiv \neg A \vee B$
2. $A \leftrightarrow B \equiv (A \wedge B) \vee (\neg A \wedge \neg B)$
3. $A \wedge B \equiv \neg(\neg A \vee \neg B)$
4. $A \vee B \equiv \neg(\neg A \wedge \neg B)$

Property	**Statement**
Associativity	$(x \vee y) \vee z \equiv x \vee (y \vee z)$ $(x \wedge y) \wedge z \equiv x \wedge (y \wedge z)$
Commutativity	$x \vee y \equiv y \vee x$ $x \wedge y \equiv y \wedge x$
Identity Laws	$x \vee F \equiv x$ $x \wedge T \equiv x$
Idempotence	$x \vee x \equiv x$ $x \wedge x \equiv x$
De Morgan's Laws	$\neg(x \vee y) \equiv \neg x \wedge \neg y$ $\neg(x \wedge y) \equiv \neg x \vee \neg y$
Excluded Middle	$x \vee \neg x \equiv T$ $x \wedge \neg x \equiv F$
Doub. Neg.	$\neg\neg x \equiv x$
Annihilation	$x \wedge F \equiv F$ $x \vee T \equiv T$
Absorption	$x \vee (x \wedge y) \equiv x$ $x \wedge (x \vee y) \equiv x$
Distributivity	$x \vee (y \wedge z) \equiv (x \vee y) \wedge (x \vee z)$ $x \wedge (y \vee z) \equiv (x \wedge y) \vee (x \wedge z)$

Predicates

1. Predicate is a function producing truth value.
2. Proposition is a thing with attached truth value.
3. Atomic proposition: F, T, [1+1=2].
4. Compound: atomic prop.'s connected by operators.
5. Totology is a composition that is always true
6. $\exists!$ - there exists exactly 1

Express finite set predicates using ($\wedge$) and ($\vee$)
1. $\forall x \in S : P(x) \equiv P(a_1) \wedge ... \wedge P(a_n)$
2. $\exists x \in S : P(x) \equiv P(a_1) \vee ... \vee P(a_n)$

De Morgan's laws on predicates
3. $\neg \forall x : P(x) \equiv \exists x : \neg P(x)$
4. $\neg \exists x : P(x) \equiv \forall x : \neg P(x)$

When Q contains x as a free variable
5. $(\forall x : P(x)) \wedge (\exists x : Q(x)) \equiv \forall x : (P(x) \wedge Q(x))$
6. $(\exists x : P(x)) \vee (\exists x : Q(x)) \equiv \exists x : (P(x) \vee Q(x))$

When Q doesn't contain x as a free variable
7. $(\forall x : P(x)) \wedge Q \equiv \forall x : (P(x) \wedge Q)$
8. $(\exists x : P(x)) \vee Q \equiv \exists x : (P(x) \vee Q)$
9. $(\forall x : P(x)) \vee Q \equiv \forall x : (P(x) \vee Q)$
10. $(\exists x : P(x)) \wedge Q \equiv \exists x : (P(x) \wedge Q)$
11. $(\forall x : P(x)) \rightarrow Q \equiv \forall x : (P(x) \rightarrow Q)$
12. $(\exists x : P(x)) \rightarrow Q \equiv \exists x : (P(x) \rightarrow Q)$
13. $Q \rightarrow (\forall x : P(x)) \equiv \forall x : (Q \rightarrow P(x))$
14. $Q \rightarrow (\exists x : P(x)) \equiv \exists x : (Q \rightarrow P(x))$
15. $(\forall x : P(x)) \equiv Q \equiv \forall x : (P(x) \equiv Q)$
16. $(\exists x : P(x)) \equiv Q \equiv \exists x : (P(x) \equiv Q)$

Other rules:
17. $\neg \forall x . P(x) \equiv \exists x . \neg P(x)$
18. $\neg \exists x . P(x) \equiv \forall x . \neg P(x)$