Artificial Intelligence I

Ibrahim Nasser[*]

Last updated on January 19, 2026

Contents

Disclaimer

This document is an independent summary by Ibrahim Nasser as a personal learning and documentation effort for the course AI 1: Artificial Intelligence 1 taught by Prof. Dr. Michael Kohlhase. It is not an official course document, and no guarantee is made for its correctness, completeness, or accuracy. The material here is meant as an auxiliary resource for understanding the subject and should not be used as the sole basis for exam preparation. The official lecture notes and course materials remain the authoritative reference.

This summary is a living document, i.e. I may continue to revise, extend, and improve it over time as I refine my understanding or receive feedback.

If you find any mistakes, unclear explanations, or have suggestions for improvement, please contact me.

1 Foundations

1.1 Mathematical Language

Def 1.1. Natural Language: A natural language is any form of spoken or signed means of communication that has evolved naturally in humans through use and repetition without conscious planning or premeditation.

Def 1.2. Domain of Discourse: The set of entities, objects, or concepts that a language, discussion, or formal system refers to. It specifies the scope within which terms, statements, and reasoning apply.

Def 1.3. Jargon (Terminology): A jargon (or terminology) is a set of specialized words or phrases (called technical terms or just terms) relating to concepts from a particular domain of discourse.

Def 1.4. Technical Language: A technical language is a natural language extended by a terminology and (possibly) special idioms, discourse markers, and notations.

Def 1.5. Stylized Language: Mathematicians use a stylized language that uses formulae to represent mathematical objects, uses math idioms for special situations (e.g. "iff", "hence", "let... be..., then..."), and classifies statements by role (e.g., definition, Lemma, Theorem, Proof, Example).

Def 1.6. Mathematical Vernacular: Specialized technical language used by mathematicians to express definitions, proofs, and reasoning. It combines natural language, standard mathematical idioms, and symbolic expressions into a coherent style for clear mathematical communication.

Def 1.7. Formulae: A mathematical formula can be a **mathematical statement** (clause) which can be true or false (e.g. $x > 5, 3 + 5 = 7$), or a **mathematical object** (e.g. $3, n, x^2 + y^2 + z^2, \int_1^0 x^{3/2} dx$)

Def 1.8. Sequence: mathematical vernacular uses the concept of sequences instead of lists. Sequences are usually finite, but can be infinite as well.

Def 1.9. Ellipses: mathematical vernacular uses ellipses $(\cdots)$ as a constructor for sequences. The meaning of an ellipsis is usually considered "obvious" and left for interpretation by the reader.

ellipses allow to write down large objects easily, examples:

- $1, \cdots, n \rightsquigarrow$ the sequence of natural numbers between 1 and n in order.
- $1, 4, 9, 16, \cdots \rightsquigarrow$ the sequence of squares in orders.
- $e_1, \cdots, e_n \rightsquigarrow$ a sequence of objects e_i for $1 \leq i \leq n$

Def 1.10. Varieties of Mathematical Statements: Mathematical statements come in different epistemic varieties:

- **Definitions**: statements that introduce new global identifier for important objects
- **Assertions**: statements that state properties of mathematical objects
- **Examples**: statements that exhibit a witness for some property
- **Axioms**: statements that characterize the objects of a certain domain of discourse or theory. An axiom (postulate) is a statement about mathematical objects that we *assume* to be true.

Def 1.11. Pragmatic Categories of Mathematical Assertions: Mathematical assertions are pragmatically classified into categories:

- **proofs** are arguments that justify the truth of statements beyond any doubt. Proofs are not really statements, but we sometimes treat them together.
- **proposition** is a statement which is interesting in its own right.
- **lemma** is an easily proved statement which is helpful for proving other propositions and theorems, but is usually not particularly interesting in its own right. A lemma is an intermediate theorem that serves as part of a proof of a larger theorem.
- **theorem** is a more important statement than a proposition which says something definitive on the subject, and often takes more effort to prove than a proposition or lemma. A theorem is a statement about mathematical object that we *know* to be true.
- **corollary** is a quick consequence of a proposition or theorem that was proven rececently. It is a theorem that follows directly from another theorem.

- **conjecture** is a statement that is thought to be provable, but has not been yet. A conjecture that is proven is a theorem

Def 1.12. Declaration: In mathematical vernacular we call a clause that introduce new identifiers together with some properties a declaration. For example Let $\epsilon, \delta > 0 \cdots$

Def 1.13. Scope: The scope of an identifier is the part of an expression where the reference is valid; that is, where the identifier can be used to refer. In the other parts, the identifier may refer to a different entity, or to nothing at all (unbound)

Def 1.14. Definiendum: The definiendum is the symbol, expression, or term being introduced by a definition. It is the newly named object or concept.

Def 1.15. Definiens: The definiens is the symbol, expression, or construction that specifies the meaning of the defineiendum. It explains what the defineiendum denotes.

Def 1.16. Simple Definition: A **simple definition** introduces a new name (the defineiendum) for an existing object or concept (the definiens). The defineiendum must be new and must not occur in the definiens. We write simple definitions using := or =:, for example $\phi := 2$.

Def 1.17. Pattern Definition: A **pattern definition** introduces a new operator or relation by applying it to distinct pattern variables $v_1, \ldots, v_n$. The definiens is an expression in these variables. Applying the operator to arguments $a_1, \ldots, a_n$ means substituting v_i by a_i in the definiens. We use := for operator definitions and :⇔ for relation definitions. Example:

$$A \cap B := \{x \mid x \in A \wedge x \in B\}.$$

Def 1.18. Implicit Definition: An **implicit definition** (definition by description) specifies a new name n (the defineiendum) by giving a clause A such that we can prove unique existence $\exists^1 n. A$. If such a proof exists, n is **well-defined**. Example: the empty set Φ is implicitly defined by $\forall x. x \notin \Phi$. Another example: the exponential function is the unique $f : \mathbb{R} \to \mathbb{R}$ satisfying $f' = f$ and $f(0) = 1$.

Def 1.19. Example: An example E is a mathematical statement that consists of:

- symbol p for the **exemplandum** (plural: exemplanda): the property to be exemplified,

- the **exemplans** (plural: exemplantia): an expression A denoting a mathematical object that acts as witness object for the property p, and

- (optionally) a **justification** of E, i.e. a proof π that $p(A)$ holds in the current context.

Def 1.20. Counter Example: An example for the complement of the property p where π is a proof that $p(A)$ does not hold.

Def 1.21. Running Example: An example that is recalled time and again during an argument, applying the newly discovered knowledge to it, presumably to show how things work.

> **Note**
>
> An example (or even several) for a mathematical statement B do not constitute a proof for B. On the other hand, one counter example for a mathematical statement B does show that $\neg B$ is true ($\exists x. \neg A \equiv \neg(\forall x. A)$).

Def 1.22. Proof by Contradiction: A **proof by contradiction** assumes the negation $\neg A$ of the claim A and derives a contradiction from that assumption. Since $\neg A$ cannot be true, we infer $\neg \neg A$ and therefore A.

Def 1.23. Proof by Local Hypothesis: A **proof by local hypothesis** is a proof of an implication $A \to B$ where we temporarily assume A and derive B from that assumption. After the subproof is closed, the assumption A may no longer be used.

Def 1.24. Chaining: **Chaining** applies a general implication $A \to B$ to a specific instance A' obtained from A by substituting variables with concrete values. This allows us to conclude the corresponding instance B' obtained from B by the same substitution.

Def 1.25. Without Loss of Generality (WLOG): An inference principle used in proofs when a statement $Q(x)$ must be shown for all cases of x, but the different cases are equivalent by symmetry or trivial reduction. Formally, if the domain of x can be partitioned into cases $A_1, \ldots, A_k$ and it can be shown that:

- proving $Q(x)$ under one representative case A_i suffices, because

- for every other case A_j there is either a trivial proof of $Q(x)$ or a direct reduction to the representative case,

then we may assume A_i holds "without loss of generality". This allows us to simplify the argument by only proving the representative case.

> **Note**
>
> **WLOG Example.** if we want to prove the property $Q(p)$ for all primes p. We know that every prime is either 2 (the only even prime) or an odd prime. If the case $p = 2$ is easy, and the general odd prime case is the only real work, then we can say:
>
> WLOG, assume p is odd
>
> This does not exclude the even case; it means we have already checked that the even case adds no new difficulty.

Def 1.26. Greek Alphabet: In mathematical vernacular, Greek letters are used extensively to convey precise meanings. Table 1 lists the Greek letters you are expected to recognize and write fluently.

> **Note**
>
> In addition, table 2 shows the curly Roman and Fraktur letters, which are also standard in mathematical writing.

α	A	alpha	β	B	beta	γ	Γ	gamma
δ	Δ	delta	ϵ	E	epsilon	ζ	Z	zeta
η	H	eta	θ, ϑ	Θ	theta	ι	I	iota
κ	K	kappa	λ	Λ	lambda	μ	M	mu
ν	N	nu	ξ	Ξ	xi	o	O	omicron
π, ϖ	Π	pi	ρ	P	rho	σ	Σ	sigma
τ	T	tau	υ	Υ	upsilon	φ, ϕ	Φ	phi
χ	X	chi	ψ	Ψ	psi	ω	Ω	omega

Table 1: Greek Alphabet

Roman	$\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}, \mathcal{F}, \mathcal{G}, \mathcal{H}, \mathcal{I}, \mathcal{J}, \mathcal{K}, \mathcal{L}, \mathcal{M}, \mathcal{N}, \mathcal{O}, \mathcal{P}, \mathcal{Q}, \mathcal{R}, \mathcal{S}, \mathcal{T}, \mathcal{U}, \mathcal{V}, \mathcal{W}, \mathcal{X}, \mathcal{Y}, \mathcal{Z}$
Fraktur	$\mathfrak{A}, \mathfrak{B}, \mathfrak{C}, \mathfrak{D}, \mathfrak{E}, \mathfrak{F}, \mathfrak{G}, \mathfrak{H}, \mathfrak{I}, \mathfrak{J}, \mathfrak{K}, \mathfrak{L}, \mathfrak{M}, \mathfrak{N}, \mathfrak{O}, \mathfrak{P}, \mathfrak{Q}, \mathfrak{R}, \mathfrak{S}, \mathfrak{T}, \mathfrak{U}, \mathfrak{V}, \mathfrak{W}, \mathfrak{X}, \mathfrak{Y}, \mathfrak{Z}$

Table 2: Roman and Fraktur Letters

1.2 Sets

Def 1.27. Set: A collection of different things (called **elements** or **members** of the set). We can represent sets by listing the elements within curly brackets, e.g. $\{a, b, c\}$ or by describing the elements via a property, e.g. $\{x \mid x \mod 2 = 0\}$ (the set of even numbers). We can state element-hood ($a \in S$) or not ($b \notin S$)

Def 1.28. Set Equality: $A \equiv B :\equiv \forall x. x \in A \Leftrightarrow x \in B$

Def 1.29. Subset: $A \subseteq B :\equiv \forall x. x \in A \Rightarrow x \in B$

Def 1.30. Proper Subset: $A \subset B :\equiv (A \subseteq B) \wedge (A \not\equiv B)$

Def 1.31. Super Set: $A \supseteq B :\equiv \forall x. x \in B \Rightarrow x \in A$

Def 1.32. Proper Superset: $A \supset B :\equiv (A \supseteq B) \wedge (A \not\equiv B)$

Def 1.33. Set Union: $A \cup B := \{x \mid x \in A \vee x \in B\}$

Def 1.34. Set Intersection: $A \cap B := \{x \mid x \in A \wedge x \in B\}$

Def 1.35. Disjoint: Two sets A, B are **disjoint** iff $A \cap B = \Phi$

Def 1.36. Set Difference: $A \setminus B := \{x \mid x \in A \wedge x \notin B\}$

Def 1.37. Power Set: $\mathcal{P}(A) := \{S \mid S \subseteq A\}$ (the set of all subsets)

Def 1.38. Cartesian Product: $A \times B := \{(a, b) \mid a \in A \wedge b \in B\}, \quad (a, b)$ is a pair

Def 1.39. Family: A set of sets

Def 1.40. Topology: A family of open sets

Def 1.41. Indexing Function: Let I and X be sets. A function $f : I \to X$ is called an indexing function. The set I is

called the index set. For each $i \in I$, we define $x_i := f(i)$, here i is called the index (or parameter) of x_i.

Def 1.42. Indexed Family: Given an indexing function $f : I \to X$, the set of values $f(I) = \{f(i) : i \in I\}$ is called an indexed family. It is often written as $(x_i)_{i \in I}$, $\langle x_i \rangle_{i \in I}$, or $\{x_i\}_{i \in I}$.

Def 1.43. Union over a Family: Let $(X_i)_{i \in I}$ be an indexed family, then $\bigcup_{i \in I} X_i := \{x \mid \exists i \in I.x \in X_i\}$

Def 1.44. Intersection over a Family: Let $(X_i)_{i \in I}$ be an indexed family, then $\bigcap_{i \in I} X_i := \{x \mid \forall i \in I.x \in X_i\}$

Def 1.45. n-fold Cartesian Product: $\prod_{i=1}^{n} X_i := X_1 \times \cdots \times X_n := \{\langle x_1, \cdots, x_n \rangle \mid \forall i.1 \leq i \leq n \Rightarrow x_i \in X_i\}$ where $\langle x_1, \cdots, x_n \rangle$ is called an n-tuple.

Def 1.46. n-dim Cartesian Space: $X^n := \{\langle x_1, \cdots, x_n \rangle \mid 1 \leq i \leq n \Rightarrow x_i \in X\}$ where $\langle x_1, \cdots, x_n \rangle$ is called a vector.

Def 1.47. Size of a Set: The size $\Gamma(A)$ (or $|A|$) of a set A is the number of elements in A.

Def 1.48. Pairwise Disjoint: A family of sets is called **pairwise disjoint** or **mutually disjoint**, if any two of those sets are disjoint.

Def 1.49. Multiset: A **multiset** (or **bag**) is a generalization of a set where elements can appear more than once. Formally, a multiset $\mathcal{M}$ over a domain D is a pair (D, m), where $m : D \to \mathbb{N}$ is a function mapping each element $x \in D$ to its **multiplicity** (the number of occurrences).

Def 1.50. Multiset Membership: $x \in \mathcal{M} :\equiv m(x) > 0$

Def 1.51. Multiset Size: The size of a multiset $\mathcal{M} = (D, m)$, denoted $\Gamma(\mathcal{M})$, is the sum of the multiplicities of all its elements: $\Gamma(\mathcal{M}) = \sum_{x \in D} m(x)$.

Def 1.52. Multiset Sum (Union): The sum of two multisets $\mathcal{M}_1 = (D, m_1)$ and $\mathcal{M}_2 = (D, m_2)$, denoted $\mathcal{M}_1 \uplus \mathcal{M}_2$, is a multiset where the multiplicity of each element is the sum of its multiplicities: $m_\uplus(x) = m_1(x) + m_2(x)$.

Def 1.53. Multiset Intersection: The intersection of two multisets $\mathcal{M}_1 = (D, m_1)$ and $\mathcal{M}_2 = (D, m_2)$, denoted $\mathcal{M}_1 \cap \mathcal{M}_2$, is a multiset where the multiplicity of each element is the minimum of its multiplicities in the two multisets: $m_\cap(x) = \min(m_1(x), m_2(x))$.

Def 1.54. Multiset Equality: $\mathcal{M}_1 \equiv \mathcal{M}_2 :\equiv \forall x \in D.m_1(x) = m_2(x)$

1.3 Relations

Def 1.55. Relation: $R \subseteq A \times B$ is a (binary) relation between A and B

Def 1.56. Relation on: a relation $R \subseteq A \times B$ where $A = B$ is called a *relation on A*

Def 1.57. Total: A relation $R \subseteq A \times B$ is called *total* (*left total*) iff $\forall x \in A.\exists y \in B.(x, y) \in R$

Def 1.58. Converse Relation: $R^{-1} \subseteq B \times A := \{(y, x) \mid (x, y) \in R\}$ is the converse relation of R

Def 1.59. Relation Composition: The composition of $R \subseteq A \times B$ and $S \subseteq B \times C$ is defined as $R \circ S := \{(a, c) \in A \times C \mid \exists b \in B.(a, b) \in R \land (b, c) \in S\}$

A relation on A: $R \subseteq A \times A$ is called:

- **reflexive** on A, iff $\forall a \in A.(a, a) \in R$

- **irreflexive** on A, iff $\forall a \in A.(a, a) \notin R$

- **symmetric** on A, iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \in R$

- **asymmetric** on A, iff $\forall a, b \in A.(a, b) \in R \Rightarrow (b, a) \notin R$

- **antisymmetric** on A, iff $\forall a, b \in A.(a, b) \in R \land (b, a) \in R \Rightarrow a = b$

- **transitive** on A, iff $\forall a, b, c \in A.(a, b) \in R \land (b, c) \in R \Rightarrow (a, c) \in R$

- **equivalence relation** on A, iff R is reflexive symmetric, and transitive

Def 1.60. Equality Relation: The equality relation is an equivalence relation on any set.

Def 1.61. Divides Relation: The *divides* relation on the integers is defined as $\mid \subseteq \mathbb{Z} \times \mathbb{Z}$, where $\mid = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid \exists k \in \mathbb{Z} : b = a \cdot k\}$. We write $a \mid b$ to denote $(a, b) \in \mid$ (read as *"a divides b"*).

Def 1.62. Congruence Modulo: For a fixed $n \in \mathbb{N}$ with $n \geq 1$, the *congruence modulo n* relation on the integers is defined as

$$\equiv_n \subseteq \mathbb{Z} \times \mathbb{Z}, \qquad \equiv_n = \{(a, b) \in \mathbb{Z} \times \mathbb{Z} \mid n \mid (a - b)\}.$$

We write $a \equiv b \pmod{n}$ to denote $(a, b) \in \equiv_n$ (read as *"a is congruent to b modulo n"*).

Def 1.63. Parital Ordering ($\preceq$): A relation $R \subseteq A \times A$ is called a (non-strict) **partial ordering** on A, iff R is reflexive, antisymmetric, and transitive on A.

Def 1.64. Strict Partial Ordering ($\prec$): A relation $R \subseteq A \times A$ is called a **strict partial ordering** on A, iff R is irreflexive, and transitive on A.

Def 1.65. Comparable: In a non-strict partial ordering, two elements a, b are comparable if either $a \preceq b$ or $b \preceq a$. If neither holds, they are incomparable.

Def 1.66. Linear Order: A partial ordering is called linear (or total) on A, iff all elements in A are comparable, i.e. if $(x, y) \in R$ or $(y, x) \in R$ for all $x, y \in A$. For example, the $\leq$ relation is a linear order on $\mathbb{N}$. However, the "divides" ($|$) relation is a non-strict partial ordering on $\mathbb{N}$ that is not linear (e.g., neither $2 \mid 3$ nor $3 \mid 2$).

Def 1.67. Partially Ordered Set (Poset): A **Partially Ordered Set** (or **poset**) is a pair $\langle G, \preceq \rangle$ where G is a set and $\preceq$ is a non-strict partial ordering on G.

Def 1.68. Least Element: Let $\langle G, \preceq \rangle$ be a poset and $T \subseteq G$. An element $t \in T$ is the **least** (or **smallest**, **minimal**, or $\preceq$**-minimal**) element of T iff $t \preceq t'$ for all $t' \in T$.

Def 1.69. Greatest Element: Let $\langle G, \preceq \rangle$ be a poset and $T \subseteq G$. An element $t \in T$ is the **greatest** (or **largest**, **maximal**, or $\preceq$**-maximal**) element of T iff $t' \preceq t$ for all $t' \in T$.

1.4 Functions

Def 1.70. Partial function: A relation $R \subseteq X \times Y$ is a *partial function* from X to Y ($f : X \rightharpoonup Y$) iff $\forall x \in X, \forall y_1, y_2 \in Y.\big((x, y_1) \in R \wedge (x, y_2) \in R \Rightarrow y_1 = y_2\big)$ (i.e. there is at most one $y \in Y$ with $(x, y) \in f$)

Def 1.71. dom, codom: We call X the domain ($\operatorname{dom}(f)$), and Y the codomain ($\operatorname{codom}(f)$)

Def 1.72. Function Application: Instead of writing $(x, y) \in f$ we write $f(x) = y$

Def 1.73. Undefined: we call a partial function $f : X \rightharpoonup Y$ undefined at $x \in X$, iff $\forall y \in Y.(x, y) \notin f$ ($f(x) = \bot$).

Def 1.74. Function: A partial function $f : X \rightharpoonup Y$ is called a *function* (or *total function*) from X to Y iff it is a total relation. ($\forall x \in X. \exists^1 y \in Y : (x, y) \in f$)

> **Note**
>
> A partial function guarantees uniqueness, but not existence. So each x has at most one y (some might have none). A total function guarantees both uniqueness and existence (at most and at least $\rightarrow$ exactly one)

A function $f : X \rightarrow Y$ is called

- **injective** iff $\forall x_1, x_2 \in X.f(x_1) = f(x_2) \Rightarrow x_1 = x_2$
- **surjective** iff $\forall y \in Y.\exists x \in X.f(x) = y$
- **bijective** iff f is injective and surjective

Def 1.75. Function Composition: If $f : A \rightarrow B$ and $g : B \rightarrow C$ are functions, then we call $g \circ f : A \rightarrow C; x \longmapsto g(f(x))$ the composition of g and f (read g after f).

Def 1.76. Image and Preimage: Let $f : X \rightarrow Y$ be a function, $X' \subseteq X$ and $Y' \subseteq Y$, then we call

- $f(X') := \{y \in Y \mid \exists x \in X'.(x, y) \in f\}$ the **image** of X' under f,
- $\operatorname{Im}(f) := f(X)$ the **image** of f, and
- $f^{-1}(Y') := \{x \in X \mid \exists y \in Y'.(x, y) \in f\}$ the **preimage** of Y' under f.

Def 1.77. Cardinality: We say that a set A is **finite** and has **cardinality** $\Gamma(A) \in \mathbb{N}$, iff there is a bijective function $f : A \rightarrow \{n \in \mathbb{N} \mid n < \Gamma(A)\}$.

Def 1.78. Countably Infinite: We say that a set A is countably infinite, iff there is a bijective function $f : A \rightarrow \mathbb{N}$.

Def 1.79. Countable: A set A is called countable, iff it is finite or countably infinite.

Def 1.80. Curried Function Type: Let X, Y, Z be sets. An *uncurried* function is written as

$$f : X \times Y \rightarrow Z, \quad f(x, y) = E(x, y).$$

The same function can equivalently be written in *curried* form as

$$f : X \to Y \to Z, \quad f(x)(y) = E(x, y).$$

Thus in the curried notation the function takes one argument at a time: for $x \in X$, the value $f(x)$ is itself a function $Y \to Z$, and applying it to $y \in Y$ yields $f(x)(y) \in Z$.

Def 1.81. Invertible: Let A and B be sets and $f : A \to B$ a function, then f is called invertible, iff there is a function $g : B \to A$, such that $f \circ g = \mathrm{Id}_B$. g is called the inverse function (or just inverse) of f and is written as f^{-1}.

1.5 Mathematical Structures

Def 1.82. Formulae: A mathematical formula can be a **mathematical statement** (clause) which can be true or false (e.g. $x > 5, 3 + 5 = 7$), or a **mathematical object** (e.g. $3, n, x^2 + y^2 + z^2, \int_1^0 x^{3/2} dx$)

Def 1.83. Mathematical Structure: A mathematical structure combines multiple mathematical objects (the components) into a new object. The components usually have names by which they can be referenced. Given a definition of a mathematical structure S, we say that any object that conforms to that is an instance of S.

Def 1.84. Group: A group is a mathematical structure $\langle G, \circ, e, .^{-1} \rangle$ that consists of:

- a base set G of objects,

- an operation $\circ : G \times G \to G$, such that $\forall a, b \in G.a \circ b \in G$ and $\forall a, b, c \in G.(a \circ b) \circ c = a \circ (b \circ c)$,

- a unit $e \in G$, such that $\forall a \in G.e \circ a = a \circ e = a$, and

- the inverse function $.^{-1} : G \to G$, such that $\forall a \in G.a \circ a^{-1} = a^{-1} \circ a = e$

An example of a group is the set $G = \mathbb{Z}$, with the operation $+ : \mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}$; then $e = 0$ and $.^{-1}$ is $\lambda x \in \mathbb{Z}. - x$. We write it as $\langle \mathbb{Z}, +, 0, - \rangle$.

Question: can we do the same for multiplication where $e = 1$? If we think about it, we would be stuck on finding the $.^{-1}$ component of the group. What can we always multiply with any integer and get $e = 1$ back? Take for example 1, we can multiply it with $\frac{1}{1}$ (great!). Take -1, we can multiply it with $\frac{1}{-1}$ (also great!). But take any other integer, e.g. 2, we must multiply it with $\frac{1}{2}$ to get 1 back, however, $\frac{1}{2} \notin \mathbb{Z}$, it is however $\in \mathbb{Q}$! The multiplicative group would only make sense iff $G = \mathbb{Q} \setminus \{0\}$. We write that as $\langle \mathbb{Q} \setminus \{0\}, *, 1, a \mapsto \frac{1}{a} \rangle$.

But we need some multiplication structure that works for $\mathbb{Z}$!

Def 1.85. Monoid: A monoid is a mathematical structure $\langle M, \circ, e \rangle$ that consists of:

- A base set M of objects,

- An operation $\circ : M \times M \to M$, such that for all $a, b \in M$, we have $a \circ b \in M$ (closure),

- Associativity: for all $a, b, c \in M$, $(a \circ b) \circ c = a \circ (b \circ c)$, and

- A unit element $e \in M$, such that for all $a \in M.e \circ a = a \circ e = a$.

Hence, the mathematical structure $\langle \mathbb{Z}, *, 1 \rangle$ is a monoid (a group missing an inverse function).

We saw that $\langle \mathbb{Z}, +, 0, - \rangle$ is a group under addition. And $\langle \mathbb{Z}, *, 1 \rangle$ is a monoid under multiplication. Now, we want to do everyday arithmetics with integers; we want expressions where we have additions and multiplication (e.g. $a * (b + c)$) which evaluates to $(a * b) + (a * c)$.

For that, we need a mathematical structure where both addition and multiplication live together.

Def 1.86. Ring: A ring is a mathematical structure $\langle R, +, 0, -, *, 1 \rangle$ consisting of:

- A base set R.

- An addition operation $+ : R \times R \to R$ such that $\langle R, +, 0, - \rangle$ is a group (called abelian group).

- A multiplication operation $* : R \times R \to R$ such that $\langle R, *, 1 \rangle$ is a monoid.

- Distributivity: for all $a, b, c \in R, a * (b + c) = (a * b) + (a * c), (a + b) * c = (a * c) + (b * c)$

Hence, mixing components from the group $\langle \mathbb{Z}, +, 0, - \rangle$ and the monoid $\langle \mathbb{Z}, *, 1 \rangle$ leaves us with a ring $\langle \mathbb{Z}, +, 0, -, *, 1 \rangle$

Def 1.87. Magma: A magma is a mathematical structure $\langle M, \circ \rangle$ consisting of:

- A base set M, and

- A binary operation $\circ : M \times M \to M$.

Example: $\langle \mathbb{Z}, - \rangle$ (integers under subtraction). $\langle \mathbb{N}, + \rangle$ (natural numbres under addition)

Def 1.88. Semigroup: A semigroup (magma with associativity) is a mathematical structure $\langle S, \circ \rangle$ consisting of:

- A base set S,
- A binary opertation $\circ : S \times S \to S$, and
- Associativity: for all $a, b, c \in S, (a \circ b) \circ c = a \circ (b \circ c)$

Example: $\langle \mathbb{Z}, + \rangle$ (integers under addition).

1.6 Formal Languages and Grammars

Def 1.89. Alphabet: An alphabet is a finite set; we call each element $a \in A$ a **character**, and an n-tuple $s \in A^n$ a **string** (of *length n over A*). We often write a string $\langle c_1, \cdots, c_n \rangle$ as "$c_1 \cdots c_n$", for instance `"abc"` for $\langle a, b, c \rangle$

Def 1.90. Empty String: Let A be an alphabet, then $A^0 = \{\langle \rangle\}$, where $\langle \rangle$ is the unique 0-tuple. We consider $\langle \rangle$ as the string of length 0 and call it the **empty string** and denote it with ϵ.

Def 1.91. String Length: Given a string s, we denote its length with $|s|$.

Def 1.92. String Concatenation: The concatenation $\text{conc}(s, t)$ of two strings $s = \langle s_1, \cdots, s_n \rangle \in A^n$ and $t = \langle t_1, \cdots, t_m \rangle \in A^m$ is defined as $s + t$ or simply st

Def 1.93. Kleene Plus/Star: Let A be an alphabet, then we define the sets:

- $A^+ := \bigcup_{i \in \mathbb{N}^+} A^i$ (*nonempty strings*), and
- $A^* := A^+ \cup \{\epsilon\}$ (*strings*).

Def 1.94. Formal Language: A set $L \subseteq A^*$ is called a *formal language* over A. For example, If $A = \{a, b\}$ then:

$$A^* = \bigcup_{n=0}^{\infty} A^n$$

where $A^0 = \epsilon$, $A^1 = \{a, b\}$, $A^2 = \{a, b, ab, ba\}$, and so on ...

$$A^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \cdots\}$$
$$A^+ = \{a, b, aa, ab, ba, bb, aaa, aab, aba, abb, \cdots\}$$

A formal language might be $L = \{a, b, aab, bbb, \cdots\}$

Def 1.95. Repeating Chars: We use $c^{[n]}$ for the string that consists of the character c repeated n times. **Examples:**

- Let $A = \{a, b\}$, $a^{[5]} = $ `"aaaaa"`.
- The set $M := \left\{ ba^{[n]} \mid n \in \mathbb{N} \right\}$ of strings that start with character b followd by an arbitrary numbers of a's is a formal language over $A = \{a, b\}$

> **Note**
>
> A formal language might be infinite and even undecidable even if the alphabet A is finite. For example, let $A = \{a, b\}$ then the formal language $L = \{a, ab, bba\}$ is finite but the formal language $L = \{ a^{[n]} \mid n \in \mathbb{N} \}$ is infinite.

Since we cannot list all strings in a formal language L because there are infinitely many, we need a **finite description** that can generate all strings in L. We need **Grammars**.

Def 1.96. Phrase Structure Grammar: A phrase structure grammar (*type 0 grammar, unrestricted grammar, grammar*) is a tuple $\langle N, \Sigma, P, S \rangle$ where

- N is a finite set of **nonterminal symbols**,
- Σ is a finite set of **terminal symbols**. (members of $N \cup \Sigma$ are called symbols).
- P is a finite set of **production rules**: pairs $p := h \to b$ (also written as $h \Rightarrow b$), where $h \in (\Sigma \cup N)^* N (\Sigma \cup N)^*$ and $b \in (\Sigma \cup N)^*$. The string h is called the **head** of p and b the **body**.

- $S \in N$ is a distinguished symbol called the **start symbol** (also **sentence symbol**).

The sets N and Σ are assumed to be disjoint. Any word $w \in \Sigma^*$ is called a **terminal word**.

> **Note**
>
> Production rules map strings with at least one nonterminal to arbitrary other strings

Notation: If we have n production rule $h \to b_i$ sharing a head, we often write $h \to b_1 \mid \cdots \mid b_n$ instead.

Example: A simple grammar for english sentences:

$$
\begin{aligned}
\text{S} &::= \text{NP Vi} \\
\text{NP} &::= \text{Article N} \\
\text{Article} &::= \text{the} \mid \text{a} \mid \text{an} \\
\text{N} &::= \text{dog} \mid \text{teacher} \mid \cdots \\
\text{Vi} &::= \text{sleeps} \mid \text{smells} \mid \cdots
\end{aligned}
$$

Def 1.97. Lexical: A production rule whose head is a single nonterminal and whose body consists of a single terminal is called **lexical** or a **lexical insertion rule**.

Def 1.98. Lexicon of a Grammar: The subset of lexical rules of a grammar G is called the **lexicon** of G

Def 1.99. Vocabulary: The set of body symbols are called the vocabulary (or the alphabet).

Def 1.100. Lexical Categories of a Grammar: The nonterminals that appear in the heads of lexical rules are called lexical categories of the grammar.

Def 1.101. Structural Rules: The non lexicon production rules are called structural.

Def 1.102. Phrasal categories: The nonterminals in the heads of production rules that expand into other nonterminals are called phrasal (syntactic) categories.

Def 1.103. G-Derivation: Given a phrase structure grammar $G := \{N, \Sigma, P, S\}$, we say G **derives** $t \in (\Sigma \cup N)^*$ from $s \in (\Sigma \cup N)^*$ in one step, iff there is a production rule $p \in P$ with $p = h \to b$ and there are $u, v \in (\Sigma \cup N)^*$, such that $s = uhv$ and $t = ubv$. We write $s \to_G^p t$ (or $s \to_G t$ if p is clear from the context) and use $\to_G^*$ for the reflexive transitive closure of $\to_G$. We call $s \to_G^* t$ a G-derivation of t from s.

Def 1.104. Sentential Form: Given a phrase structure grammar $G := \{N, \Sigma, P, S\}$, we say that $s \in (\Sigma \cup N)^*$ is a **sentential form** of G, iff $S \to_G^* s$

Def 1.105. Sentence: A sentential form that does not contain nonterminals is called a sentence of G, we also say that G **accepts** s. We say that G **rejects** s, iff it is not a sentence of G.

Def 1.106. Language Generation: The language $L(G)$ of G is the set of its sentences. We say that $L(G)$ is **generated** by G.

Def 1.107. Equivalent Grammars: We call two grammars **equivalent**, iff they have the same languages.

Def 1.108. Universal Grammar: A grammar G is said to be **universal** if $L(G) = \Sigma^*$

Def 1.109. Syntactic Analysis: Syntactic analysis (parsing / syntax analysis) is the process of analyzing a string of symbols, either in a formal or a natural language by means of a grammar.

> **Note**
>
> The shape of the grammar determines the size of its language

Def 1.110. Context-Sensitive: We call a grammar context-sensitive (or type 1), if the bodies of production rules have no less symbols than the heads.

Def 1.111. Context-Free: We call a grammar context-free (or type 2), iff the heads have exactly one symbol.

Def 1.112. Regular: We call a grammar regular (or type 3, or REG), if additionally the bodies are empty or consist of a nonterminal, optionally followed by a terminal symbol.

1.7 Graphs and Trees

Def 1.113. Undirected Graph: An undirected graph is a pair $\langle V, E \rangle$ such that

- V is a set of **vertices (nodes)**, and
- $E \subseteq \{\, \{v, \bar{v}\} \mid v, \bar{v} \in V \wedge v \neq \bar{v} \,\}$ is the set of its **undirected edges**.

Def 1.114. Directed Graph: A directed graph (digraph) is a pair $\langle V, E \rangle$ such that

- V is a set of **vertices (nodes)**, and
- $E \subseteq V \times V$ is the set of its **directed edges**.

Def 1.115. Indegree: Given a directed graph $\langle V, E \rangle$, the indegree $\mathrm{indeg}(v)$ of a vertex $v \in V$ is defined as $\mathrm{indeg}(v) = \Gamma(\{\, w \mid (w, v) \in E \,\})$.

Def 1.116. Outdegree: Given a directed graph $\langle V, E \rangle$, the outdegree $\mathrm{outdeg}(v)$ (branching factor) of a vertex $v \in V$ is defined as $\mathrm{indeg}(v) = \Gamma(\{\, w \mid (v, w) \in E \,\})$.

> **Note**
>
> directed graphs are nothing else than relations.

Def 1.117. Initial vs Terminal Node: Let $G = \langle V, E \rangle$ be a directed graph, then we call a node $v \in V$

- **initial** (source of G), iff there is no $w \in V$ such that $(w, v) \in E$ (no predecessor)
- **terminal** (sink of G), iff there is no $w \in V$ such that $(v, w) \in E$ (no successor)

Def 1.118. Graph Isomorphism: Iff we can find a bijection $\psi : V \to \bar{V}$ between two graphs $G = \langle V, E \rangle$ and $\bar{G} = \langle \bar{V}, \bar{E} \rangle$ then we call them isomorphic. The bijection $\psi : V \to \bar{V}$ is defined as $(a, b) \in E \Leftrightarrow (\psi(a), \psi(b)) \in \bar{E}$ for directed graph. And for undirected graph it is defined as $\{a, b\} \in E \Leftrightarrow \{\psi(a), \psi(b)\} \in \bar{E}$

Def 1.119. Equivalent Graphs: Two graphs G and $\bar{G}$ are **equivalent** iff there is an isomorphism ψ between G and $\bar{G}$.

Def 1.120. Labeled Graph: A labeled graph G is a quadruple $\langle V, E, L, l \rangle$ where $\langle V, E \rangle$ is a graph and $l : V \cup E \rightharpoonup L$ is a partial function into a set L of labels.

Def 1.121. Paths in Graphs: Given a graph $G := \langle V, E \rangle$ we call a $n + 1$-tuple $p = \langle v_0, \cdots, v_n \rangle \in V^{n+1}$ a **path** in G iff $(v_{i-1}, v_i) \in E$ for all $1 \leq i \leq n$ and $n > 0$.

- We say that v_i are nodes on p and that v_0 and v_n are **linked** by p.
- v_0 and v_n are called the **start** and **end** of p (write $\mathrm{start}(p)$ and $\mathrm{end}(p)$), the other v_i are called **inner nodes** of p.
- n is called the **length** of p (write $\mathrm{len}(p)$).
- We denote the set of paths in G with $\Pi(G)$.

Def 1.122. Cyclic Graphs: Given a directed graph $G = \langle V, E \rangle$, a path p is called **cyclic** iff $\mathrm{start}(p) = \mathrm{end}(p)$. A cycle $\langle v_0, \cdots, v_n \rangle$ is called **simple**, iff $v_i \neq v_j$ for $1 \leq i, j \leq n$ with $i \neq j$ (all inner nodes are distinct).

Def 1.123. DAG: A directed graph with no cycles is called **directed acyclic graph (DAG)**.

Def 1.124. Node Depth: Let $G = \langle V, E \rangle$ be a directed graph, then the depth $\mathrm{dp}(v)$ of a vertex $v \in V$ is defined to be 0, iff v is a source of G and the supremum $\sup(\{\, \mathrm{len}(p) \mid \mathrm{indeg}(\mathrm{start}(p)) = 0 \wedge \mathrm{end}(p) = v \,\})$ otherwise, i.e. the length of the longest path from a source of G to v (can be infinite).

Def 1.125. Graph Depth: Given a directed graph $G = \langle V, E \rangle$, the **depth** $(\mathrm{dp}(G))$ of G is defined as the supremum $\sup(\{\, \mathrm{len}(p) \mid p \in \Pi(G) \,\})$, i.e. the maximal path length in G.

Def 1.126. Tree: A tree is a DAG $G = \langle V, E \rangle$ such that

- There is exactly one initial node $v_r \in V$ (called the **root**)
- All nodes but the root have indegree of 1.

We call v the **parent** of w, iff $(v, w) \in E$ (w is a child of v). We call a node v a **leaf** of G, iff it is terminal, i.e. if it does not have children.

> **Note**
>
> For any node $v \in V$ except the root v_r, there is exactly one path $p \in \Pi(G)$ with $\text{start}(p) = v_r$ and $\text{end}(p) = v$.

2   Introduction

Def 2.1. Artificial intelligence (AI): The capability of computational systems to perform tasks typically associated with human intelligence, such as learning, reasoning, problem-solving, perception, and decision-making

Def 2.2. Symbolic AI: A subfield of Artificial Intelligence (AI) based on the assumption that many aspects of intelligence can be achieved by the manipulation of symbols, combining them into meaningful structures (expressions) and manipulating them (using processes) to produce new expressions.

Def 2.3. Statisitical AI: Remedies the two shortcomings of symbolic AI approaches: that all concepts represented by symbols are crisply defined, and that all aspects of the world are knowable/representable in principle. Statistical AI adopts sophisticated mathematical models of uncertainty and uses them to create more accurate world models and reason about them.

Def 2.4. Subsymbolic AI (a.k.a connectionism/neural AI): A subfield of AI that posits that intelligence is inherently tied to brains, where information is represented by a simple sequence pulses that are processed in parallel via simple calculations realized by neurons, and thus concentrates on neural computing.

Def 2.5. Embodied AI: Posits that intelligence cannot be achieved by reasoning about the state of the world (symbolically, statistically, or sub-symbolically), but must be embodied i.e. situated in the world, equipped with a "body" that can interact with it via sensors and actuators. Here, the main method for realizing intelligent behavior is by learning from the world.

Def 2.6. Reasoning: The process of producing valid arguments and predictive world models. There are three forms of reasoning:

- **deductive reasoning** to produce new knowledge from existing knowledge. *Example:* All humans are mortal. Socrates is a human. Therefore, Socrates is mortal.

- **inductive reasoning** to produce knowledge from perception. *Example:* The sun has risen every morning in recorded history. Therefore, the sun will rise tomorrow.

- **abductive reasoning** to produce explanations for observations and given knowledge. *Example:* The grass is wet this morning. If it rained last night, that would explain it. Therefore, it probably rained.

Def 2.7. Inference: The act or process of reaching a **conclusion** about something from known facts or evidence (jointly called **premises**)

Def 2.8. Formal Logic: The science of deductively valid inferences, meaning arguments whose conclusions necessarily follow from their premises by virtue of their form (their structure), regardless of the topic.

Def 2.9. Agent: An agent is a structure $A := \langle \mathcal{P}, \mathcal{A}, f \rangle$ where:

- $\mathcal{P}$ is a set of percepts

- $\mathcal{A}$ is a set of actions

- f is a function $f : \mathcal{P}^* \to \mathcal{A}$ that maps from percepts to actions.

In other words, an agent is anything that perceives its environment via sensors and acts on it with actuators.

Def 2.10. Performance Measure: A function that evaluates a sequence of environments.

Def 2.11. Rational: An agent is called **rational**, if it chooses whichever action maximizes the expected value of the performance measure given the percept sequence to date.

Def 2.12. PEAS: To design rational agents, we must specify the PEAS components: Performance Measure, Environment, Actuators, and Sensors.

Def 2.13. Environment Types: For an agent a we classify the environment e of a by its **type**, which is one of the following. We call e

1. *fully observable*, iff the a's sensors give it access to the complete state of the environment at any point in time; otherwise, we call it *partially observable*.

2. *deterministic*, iff the next state of the environment is completely determined by the current state and a's actions; otherwise, *stochastic*.

3. *episodic*, iff a's experience is divided into atomic episodes, where it perceives and then performs a single action, and the next episode does not depend on previous ones. Otherwise, *sequential*.

4. *dynamic*, iff the environment can change without an action performed by a; otherwise, *static*. If the environment does not change but a's performance measure does, we call e *semidynamic*.

5. *discrete*, iff the sets of e's state and a's actions are couuntable; otherwise, *continuous*.

6. *single-agent*, iff only a acts on e; otherwise *multi-agent*.

Def 2.14. Reflex: An agent $\langle \mathcal{P}, \mathcal{A}, f \rangle$ is called a **reflex agent**, iff it only takes the last percept into account when choosing an action, i.e. $f(p_1, \cdots, p_k) = f(p_k) \; \forall p_1 \cdots p_k \in \mathcal{P}$

Def 2.15. Model-Based Agent: A model-based agent $\langle \mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0, S, a \rangle$ is an agent $\langle \mathcal{P}, \mathcal{A}, f \rangle$ whose actions depend on:

- a **world model**: a set $\mathcal{S}$ of possible *states*, and a start state $s_0 \in \mathcal{S}$.

- a **transition model** $\mathcal{T}$ that predicts a new state $\mathcal{T}(s, a)$ from a state s and an action a.

- a **sensor model** S that given a state s and a percept p determines a new state $S(s, p)$.

- an **action function** $a : \mathcal{S} \to \mathcal{A}$ that given a state $s \in \mathcal{S}$ selects the next action $a \in \mathcal{A}$.

> **Note**
>
> If the agent is in state s then it took action a and now perceives p, then the agent state will become $s' = S(p, \mathcal{T}(s, a))$ and accordingly take action $a' = a(s')$.

Def 2.16. Goal Based Agent: A goal based agent $\langle \mathcal{P}, \mathcal{A}, \mathcal{S}, \mathcal{T}, s_0, S, a, \mathcal{G} \rangle$ is a model based agent with an explicit set of goals. It consists of:

- a set of internal states $\mathcal{S}$ and an initial state $s_0 \in \mathcal{S}$,

- a transition model $\mathcal{T} : \mathcal{S} \times \mathcal{A} \to \mathcal{S}$,

- a state update function $S : \mathcal{P} \times \mathcal{S} \to \mathcal{S}$,

- a set of goals $\mathcal{G}$,

- a goal conditioned action function $a : \mathcal{S} \times \mathcal{G} \to \mathcal{A}$ selecting an action given a state and the goals.

Def 2.17. Utility-Based Agent: A utility-based agent uses a world model along with a utility function that models its preferences among the states of that world. It chooses the action that maximizes the expected utility.

Def 2.18. Learning Agent: A learning agent is an agent that can improve its own behavior through experience. It is composed of four main components:

- the performance element, which selects actions based on the agent's current percepts and represents its existing knowledge or behavior,

- the learning element, which improves the performance element over time by analyzing feedback and identifying how to make better decisions,

- the critic, which evaluates the agent's performance according to a given performance standard and provides feedback to the learning element,

- the problem generator, which suggests new and informative actions or experiences that help the agent explore and learn more effectively.

Def 2.19. State Representation: We call a state representation **atomic**, iff it has no internal structure (black box). However, iff each state is characterized by attributes and their values then the representation is **factored**. A **structured** state representation is when include representations of objects, their properties and relationships.

3   General Problem Solving

Def 3.1. Search Problem: A search problem $\Pi := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$ consists of a set $\mathcal{S}$ of states, a set $\mathcal{A}$ of actions, and a transition model $\mathcal{T} : \mathcal{A} \times \mathcal{S} \to \mathcal{P}(\mathcal{S})$ that assigns to any action $a \in \mathcal{A}$ and state $s \in \mathcal{S}$ a set of successor states. Certain states in $\mathcal{S}$ are designated as goal (terminal) states ($\mathcal{G} \subseteq \mathcal{S}$ with $\mathcal{G} \neq \phi$) and initial states $\mathcal{I} \subseteq \mathcal{S}$.

Def 3.2. Action Application: We say that an action $a \in \mathcal{A}$ is applicable in state $s \in \mathcal{S}$, iff $\mathcal{T}(a, s) \neq \phi$ and that any $s' \in \mathcal{T}(a, s)$ is a result of applying action a to state s. We call $\mathcal{T}_a : \mathcal{S} \to \mathcal{P}(\mathcal{S})$ with $\mathcal{T}_a(s) := \mathcal{T}(a, s)$ the result relation for a and $\mathcal{T}_\mathcal{A} := \bigcup_{a \in \mathcal{A}} \mathcal{T}_a$ the result relation of Π.

Def 3.3. State Space: The graph $\langle \mathcal{S}, \mathcal{T}_\mathcal{A} \rangle$ is called the state space induced by Π.

Def 3.4. Solution: A solution for Π consists of a sequence $a_1, \cdots, a_n$ of actions such that for all $1 < i \leq n$:

- a_i is applicable to state s_{i-1} where $s_o \in \mathcal{I}$, and
- $s_i \in \mathcal{T}_{a_i}(s_{i-1})$, and $s_n \in \mathcal{G}$

Def 3.5. Cost Function: Often we add a cost function $c : \mathcal{A} \to \mathbb{R}_0^+$ that associates a step cost $c(a)$ to an action $a \in \mathcal{A}$. The cost of a solution is the sum of the step costs of its actions.

> **Note**
>
> - For deterministic environments, we have $|\mathcal{T}(a, s)| \leq 1$
> - For fully observable ones, we have $\mathcal{I} = \{s_0\}$

Def 3.6. Successor Function/State: In a search problem, $\mathcal{T}_a$ induces a partial function $S_a : \mathcal{S} \rightharpoonup \mathcal{S}$ whose natural domain is the set of states where a is applicable: $S_a(s) := s'$ if $\mathcal{T}_a = \{s'\}$ and undefined at s otherwise. We call S_a the **successor function** for a and $S_a(s)$ the **successor state** of s.

> **Note**
>
> - A search problem is called a **single-state problem** iff it is fully observable, deterministic, static, and discrete.
> - A search problem is called a **multi-state problem** iff it is partially observable.
> - A search problem is called a **contingency problem** iff the environment is non deterministic and the state space is unknown.

Def 3.7. Tree Search: Given a search problem $\Pi := \langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$, the **tree search algorithm** consists of the simulated exploration of the state space $\langle \mathcal{S}, \mathcal{T}_\mathcal{A} \rangle$ in a search tree formed by successively expanding already explored states.

Def 3.8. Path Cost: We define the path cost of a node n in a search tree T to be the sum of the step costs on the path from n to the root of T.

The general structure of Tree Search algorithms is as follows:

```
TREE_SEARCH(start, INSERT):
    fringe ← empty list
    append start to fringe
    while fringe is not empty:
        node ← remove first element of fringe
        if is_goal(node):
            return node
        children ← expand(node)
        INSERT(fringe, children)
    return failure
```

Def 3.9. Strategy: A strategy is a function that picks a node from the fringe of a search tree

> **Note**
>
> Every algorithm has its own strategy, hence the implemention of the INSERT differs based on that strategy.

Def 3.10. Properties of Strategies:

- completeness: does it always find a solution if one exists?
- optimality: does it always find the optimal solution (least cost)?

- time complexity: number of nodes the algorithm explores (expands)

- space complexity: maximum number of nodes in memory

> **Note**
>
> Time and space complexity measured in terms of:
> - b: maximum branching factor of the search tree
> - d: minimal graph depth of a solution in the search tree
> - m: maximum graph depth of the search tree

Def 3.11. Breadth-First Search (BFS): The BFS strategy treats the fringe as a FIFO queue, i.e. successors go in at the end (back) of the fringe.

BFS Properties Analysis

- Time Complexity:

  assume we have a binary search tree ($b = 2$) and we found solution in depth $= 3$. In the 0th level we explored $2^0 = 1$ nodes. In the 1st level, $2^1 = 2$ nodes. In the 2nd level, $2^2 = 4$ nodes. And finally $2^3 = 8$ nodes in the third level. Number of nodes the algorithm explored will be $2^0 + 2^1 + 2^2 + 2^3$.

  - Generally, for arbitrary b, d, we have $1 + b + b^2 + \cdots + b^d$, which means, worst time complexity is $\mathcal{O}(b^d)$ (exponential in d).

- The space complexity is also $\mathcal{O}(b^d)$.

- BFS is complete if: (i) b is finite, and (ii) the state space is finite or has a solution.

- BFS is optimal if we have a uniform constant cost for all actions.

Def 3.12. Uniform-Cost Search (UCS): UCS is the strategy where the fringe is ordered by increasing path cost. (*expands least cost first*)

> **Note**
>
> UCS it is equivalent to BFS if all costs are equal

UCS Properties Analysis

- Time and space complexity $\approx \mathcal{O}(b^d)$

- UCS is complete if:

  - b is finite

  - all actions costs are $\geq \epsilon > 0$

  - state space is finite or has a solution.

Def 3.13. Depth-First Search (DFS): DFS is the strategy where the fringe is organized as a LIFO stack, i.e. successors go in at the front of the fringe.

Def 3.14. Backtracking: Every node that is pushed to the stack is called a **backtrack point**. Popping a non-goal node from the stack and continuing the search with the new top element is called **backtracking**. For this reason, the DFS algorithm is also referred to as a **backtracking search**.

DFS Properties Analysis

- Time complexity is $\mathcal{O}(b^m)$ (exponential in the maximum depth)

- Space complexity is linear. While BFS keeps track of all nodes at the current level in memoty, DFS needs to only keep the current path (from root to m), and also needs to keep the remaining unexplored sibling root node for each node along that path (to backtrack). This means, for each level, we store the siblings, i.e. b nodes. We explore until we hit the max depth m and in each level we store b. Hence we have a linear space $\mathcal{O}(b \cdot m)$

- DFS is complete if the search tree is finite and acyclic.

- DFS is not optimal.

Def 3.15. Depth-Limited Search (DLS): DLS is a DFS with a depth limit l. We treat all nodes at depth l as if they have no children.

DLS Properties Analysis

- Time complexity $\rightsquigarrow \mathcal{O}(b^l)$

- Space complexity $\rightsquigarrow \mathcal{O}(b \cdot l)$

- Not complete (solution maybe beyond chosen l)

- Not optimal

Def 3.16. Iterative Deepening Search (IDS): IDS is a DLS with an ever-increasing depth limit. We call the difference between successive depth limits the step size.

> **Note**
> - IDS solves the problem of choosing a good l in DLS.
> - IDS tries all values until either a solution is found (depth is returned to DLS, we call that the cutoff value) or we return failure.
> - IDS combines benefits of DFS and BFS.

IDS Properties Analysis

It is similar to DFS in memory requirements (assuimg finite and acyclic search tree).

- $\mathcal{O}(b \cdot d)$ when there is a solution

- $\mathcal{O}(b \cdot m)$ when there is no solution

It is similar to BFS in terms of optimality, completeness, and time complexity. Optimal for problems where all actions have the same cost. Complete if b is finite and the state space is finite or has a solution.

With regard to time complexity, consider IDS finds a solution at depth d, then:

- nodes at the very bottom (d) are visited only once in the final iteration.

- nodes at $d-1$ are visited twice, once in the search with depth $d-1$ and once in the search with depth d.

- nodes at $d-2$ are visited three times

- and so son, until root node ($d = 0$) which is visited d times.

Generally, we have $(d)b^1 + (d-1)b^2 + (d-2)b^3 + \cdots + b^d$. However, the majority of nodes are at the max depth where there are b^d nodes. Those are only visited once and the extra cost of visiting shallower nodes does not significantly increase the overall count. Therefore, time complexity is bounded by b^d ($\mathcal{O}(b^d)$) because b^d dominates the total nodes count in a large search space.

Def 3.17. Graph Search: A graph search algorithm is a variant of a tree search algorithm that prunes nodes whose state has already been considrered (duplicate pruning), essentially using a DAG data strucutre.

The general structure of Graph Search algorithms is as follows:

```
GRAPH_SEARCH(start, INSERT):
    fringe ← empty list
    visited ← empty set
    append start to fringe
    add state(start) to visited
    while fringe is not empty:
        node ← remove first element of fringe
        if is_goal(node):
            return node
        children ← expand(node)
        for each child in children:
            if state(child) not in visited:
                add state(child) to visited
                INSERT(fringe, child)
    return failure
```

Def 3.18. Search Alrogrithm: We speak of a **search algorithm** when we do not want to distinguish whether it is a tree or a graph search algorithm - *difference considered as an implementation detail.*

Def 3.19. Informed: A search algorithm is called **informed**, iff it uses some form of external information to guide the search.

Def 3.20. Evaluation Function: An evaluation function assigns a desirability value to each node of the search tree. It is not part of the search problem, but must be added externally.

Def 3.21. Best-First Search: In best-first search, the fringe is a queue sorted in decreasing order of desirability.

Def 3.22. Heuristic: A heuristic is an evaluation function h on states that estimates the cost form n to the nearest goal state. We speak of **heuristic search** if the search algorithm uses a heuristic in some way.

Def 3.23. Greedy Search: A best-first search strategy where the fringe is organized as a queue sorted based on h value.

Greedy Search Properties

- Time and space complexity are both exponential in m (max depth) $\rightsquigarrow \mathcal{O}(b^m)$

- It is not optimal. But it is complete if the state space is finite.

Def 3.24. Heuristic Function: A heuristic funciton for search problem Π is a function $h : \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$ so that $h(s) = 0 \quad \forall s \in \mathcal{G}$.

Def 3.25. Goal Distance Function: The function $h^* : \mathcal{S} \to \mathbb{R}_0^+ \cup \{\infty\}$, where $h^*(s)$ is the cost of a cheapest path from s to a goal state, or ∞ if no such path exists. (*the perfect heuristic which we do not know and cannot compute*)

Def 3.26. Admissible: For a search problem Π with states $\mathcal{S}$ and actions $\mathcal{A}$. We say that a heuristic h for Π is **admissible** if

$$h(s) \leq h^*(s) \quad \forall s \in \mathcal{S}$$

> **Note**
>
> Admissible heuristics never overestimates; our guess should be always optimistic or exact, never too high

Def 3.27. Consistent: For a search problem Π with states $\mathcal{S}$ and actions $\mathcal{A}$. We say that a heuristic h for Π is **consistent** if

$$h(s) - h(s') \leq c(a) \quad \forall s \in \mathcal{S}, a \in \mathcal{A}, s' \in \mathcal{T}(s, a)$$

> **Note**
>
> A consistent heuristic never drops faster than the real cost you pay when moving through the state space. Formally, for every transition $s \xrightarrow{a} s'$,
>
> $$h(s) \leq c(a) + h(s')$$
>
> This means the heuristic respects a triangle inequality with respect to the actual step costs. Consistency ensures that the estimated total cost along a path never decreases.

Def 3.28. A* Evaluation Function: Given a path cost function g and a heuristic h, the **A* evaluation function** is

$$f(n) = g(n) + h(n).$$

It estimates the total cost of a cheapest path from the start state to a goal via n.

Def 3.29. A* Search: A* search is the best-first search strategy that uses the A* evaluation function $f = g + h$ to order the fringe.

Theorem 3.1. A* Search with admissible heuristic is optimal

Def 3.30. Dominance: Let h_1 and h_2 be two admissible heuristic, we say that h_2 **dominates** h_1 if $h_2(s) \geq h_1(s) \quad \forall s \in \mathcal{S}$.

Theorem 3.2. If h_2 dominates h_1, then h_2 is better for search than h_1

Proof. If h_2 dominates h_1, then h_2 is closer to h^* than h_1 □

Def 3.31. Adversarial Search: An adversarial search problem is a search problem $\langle \mathcal{S}, \mathcal{A}, \mathcal{T}, \mathcal{I}, \mathcal{G} \rangle$, where:

- $\mathcal{S} = \mathcal{S}^{\mathrm{Max}} \uplus \mathcal{S}^{\mathrm{Min}} \uplus \mathcal{G}$

- $\mathcal{A} = \mathcal{A}^{\mathrm{Max}} \uplus \mathcal{A}^{\mathrm{Min}}$

- For $a \in \mathcal{A}^{\mathrm{Max}}$, if $s \xrightarrow{a} s'$, then $s \in \mathcal{S}^{\mathrm{Max}}$ and $s' \in (\mathcal{S}^{\mathrm{Min}} \cup \mathcal{G})$

- For $a \in \mathcal{A}^{\mathrm{Min}}$, if $s \xrightarrow{a} s'$, then $s \in \mathcal{S}^{\mathrm{Min}}$ and $s' \in (\mathcal{S}^{\mathrm{Max}} \cup \mathcal{G})$

together with a **game utility function** $u : \mathcal{G} \to \mathbb{R}$

Def 3.32. Strategy: Let Θ be an adversarial search problem, and let $X \in \{\mathrm{Max}, \mathrm{Min}\}$. A **strategy** for X is a function $\sigma^X : \mathcal{S}^X \to \mathcal{A}^X$ so that $a \in \mathcal{A}^X$ is applicable to $s \in \mathcal{S}^X$ whenever $\sigma^X(s) = a$.

Def 3.33. Optimal Strategy: A strategy is called **optimal** if it yields the best possible utility for X assuming perfect opponent play.

> **Note**
>
> Assumptions for Adversarial Games:
> - Two players acting in strictly alternating turns
> - Fully observable and deterministic game states
> - Finite state space so the game tree is finite
> - Terminal states have a real-valued utility
> - The game is zero-sum: Max maximizes the utility, Min minimizes the same utility

Def 3.34. Minimax: Let Θ be an adversarial search problem with utility function $u : \mathcal{G} \to \mathbb{R}$. The **minimax value** is the function $\hat{u} : \mathcal{S} \to \mathbb{R}$ defined by:

$$\hat{u}(s) = \begin{cases} u(s) & \text{if } s \in \mathcal{G}, \\ \max\limits_{s' \in \mathcal{T}(a,s)} \hat{u}(s') & \text{if } s \in \mathcal{S}^{\mathrm{Max}}, \\ \min\limits_{s' \in \mathcal{T}(a,s)} \hat{u}(s') & \text{if } s \in \mathcal{S}^{\mathrm{Min}}, \end{cases}$$

where a ranges over all actions applicable to s. The **minimax decision** at a root state r is any action whose successor state achieves $\hat{u}(r)$.

Def 3.35. Minimax Algorithm: The minimax algorithm is given by the following recursive function whose arguments are a node (state) and whether it is Max's turn or Min's.

```
function MINIMAX(node, maximizing):
    if TERMINAL(node):
        return UTILITY(node)

    if maximizing:
        best = -infinity
        for each child in SUCCESSORS(node):
            val = MINIMAX(child, false)
            if val > best:
                best = val
        return best

    else:
        best = +infinity
        for each child in SUCCESSORS(node):
            val = MINIMAX(child, true)
            if val < best:
                best = val
        return best
```

Def 3.36. Evaluation Function: In practice, full-depth minimax is infeasible because the search tree is too large. We therefore impose a fixed *depth limit / horizon d* and stop the search at all states $\{ s \in \mathcal{S} \mid \mathrm{dp}(s) = d \}$, called *cut-off states.*

An evaluation function is a function $f : \mathcal{S} \to \mathbb{R}$ that assigns a numerical estimate of the true minimax value $\hat{u}(s)$ to any nonterminal cut-off state s. If a cut-off state is terminal, its utility $u(s)$ is used instead of $f(s)$.

Def 3.37. Alpha Value: For each node n in a minimax search tree, the alpha value $\alpha(n)$ is the highest Max-node utility that search has encountered on its path from the root to n.

Def 3.38. Beta Value: For each node n in a minimax search tree, the beta value $\beta(n)$ is the highest Min-node utility that search has encountered on its path from the root to n.

Def 3.39. Alpha-Beta Pruning: Alpha-beta pruning is a variant of minimax that avoids exploring branches that cannot change the minimax value.

- At a Max-node n, if a successor s is found with utility $\hat{u}(s) \geq \beta(n)$, then no remaining successors of n can affect its value and they are pruned.

- At a Min-node n, if a successor s is found with utility $\hat{u}(s) \leq \alpha(n)$, then no remaining successors of n can affect its value and they are pruned.

The algorithm returns the same minimax value as full search while expanding fewer nodes.

```
function ALPHABETA(node, alpha, beta, maximizing):
    if TERMINAL(node):
        return UTILITY(node)

    if maximizing:
        best = -infinity
        for each child in SUCCESSORS(node):
            val = ALPHABETA(child, alpha, beta, false)
            if val > best:
                best = val
            if best > alpha:
                alpha = best
            if beta <= alpha:
                break
        return best

    else:
        best = +infinity
        for each child in SUCCESSORS(node):
            val = ALPHABETA(child, alpha, beta, true)
            if val < best:
                best = val
            if best < beta:
                beta = best
            if beta <= alpha:
                break
        return best
```

4 Constraint Satisfaction Problems

Def 4.1. Constraint Satisfaction Problem (CSP): A triple $\gamma := \langle V, D, C \rangle$ where

- V is a finite set of variables,
- D is an V-indexed family of domains: $(D_v)_{v \in V}$
- C is the set of constraints. For a subset $\{v_1, \cdots, v_k\} \subseteq V$, a constraint $C_{\{v_1, \cdots, v_k\}} \subset D_{v_1} \times \cdots \times D_{v_k}$

Def 4.2. Variable Assignment: We call a partial function:

$$\varphi : V \rightharpoonup \bigcup_{v \in V} D_v$$

a **variable assignment** if $\varphi(v) \in D_v \quad \forall v \in \text{dom}(\varphi)$

Def 4.3. Satisfying Assignment: a variable assignment φ **satisfies** a constraint $C_{\{v_1, \cdots, v_k\}}$ iff $(\varphi(v_1), \cdots, \varphi(v_k)) \in C_{\{v_1, \cdots, v_k\}}$.

Def 4.4. Consistent Assignment: a variable assignment φ is called **consistent** iff it satisfies all constraints in C.

Def 4.5. Legal Assignment: A value $d \in D_v$ is called **legal** for a variable $v \in V$ iff $v \mapsto d$ is a consistent assignemnt; otherwise, **illegal**.

Def 4.6. Conflicted: A variable with an illegal value under assignment φ is called **conflicted**.

Def 4.7. CSP Solution: A variable assignment that is total (i.e. function) and consistent is a **solution** for the CSP.

Def 4.8. Satisfiable: A CSP γ is called **satisfiable** iff it has a solution (a total variable assignemnt φ that satisfies all constraints).

Def 4.9. Discrete: We call a CSP **discrete** iff all of the variables have countable domains; otherwise, **continuous**.

Def 4.10. Boolean CSP: A discrete CSP is called **boolean** iff $|D_v| = 2 \quad \forall v \in V$.

> **Note**
>
> Discrete CSPs with domain size d and n variables has a search space of size d^n, so a naive solve (brute-force) is worst case $\mathcal{O}(d^n)$. In general, deciding solvability of a finite-domain CSP is NP-complete.

Def 4.11. Constraint Order (Arity): Let $\gamma = \langle V, D, C \rangle$ be a CSP. For a constraint

$$C_{\{v_1, \ldots, v_k\}} \subseteq D_{v_1} \times \cdots \times D_{v_k},$$

its **order** (or *arity*) is

$$\text{ord}\big(C_{\{v_1, \ldots, v_k\}}\big) := k.$$

The order of γ is

$$\max_{C_{\{v_1, \ldots, v_k\}} \in C} k.$$

A constraint of order 1 is *unary*, order 2 is *binary*, and any constraint with order > 2 is *higher order*.

Def 4.12. Binary: A **binary CSP** is a CSP where each constraint is unary or binary.

Def 4.13. Constraint Graph: A binary CSP forms a graph called the **constraint graph** whose nodes are variables, and whose edges represent the constraints.

Def 4.14. Constraint Network: A constraint network is a CSP $\gamma := \langle V, D, C \rangle$ of order 2, hence, we write a unary constraint as C_v (representing $C_{\{v\}}$) and a binary constraint as C_{uv} (representing $C_{\{u,v\}}$). Note that $C_{uv} = C_{vu}$.

> **Note**
>
> The constraint graph where all constraints are binary is the undirected graph:
>
> $$\big\langle V, \big\{ (u,v) \in V^2 \mid C_{uv} \neq D_u \times D_v \big\} \big\rangle$$

CSP as Search

We can induce a search problem $\Pi_\gamma := \langle \mathcal{S}_\gamma, \mathcal{A}_\gamma, \mathcal{T}_\gamma, \mathcal{I}_\gamma, \mathcal{G}_\gamma \rangle$ from the constraint network $\gamma := \langle V, D, C \rangle$.

We define the set of states $\mathcal{S}_\gamma$ as the set of all partial assignments:

$$\mathcal{S}_\gamma := \left\{ \varphi : V \rightharpoonup \bigcup_{v \in V} D_v \mid \varphi(v) \in D_v \quad \forall v \in \mathrm{dom}(\varphi) \right\}$$

We define goal states $\mathcal{G}_\gamma$ as the set of total and consistent assignments

$$\mathcal{G}_\gamma := \left\{ \varphi : V \rightarrow \bigcup_{v \in V} D_v \mid (\varphi(v) \in D_v \quad \wedge \quad (\varphi(u), \varphi(v)) \in C_{uv}) \quad \forall u, v \in V \right\}$$

The initial state is when we have no assignments:

$$\mathcal{I}_\gamma := \left\{ \varphi : V \rightharpoonup \bigcup_{v \in V} D_v \mid \mathrm{dom}(\varphi) = \emptyset \right\}$$

Actions represent assignments as pairs:
$$\mathcal{A}_\gamma := \{ (v, d) \mid v \in V \wedge d \in D_v \}$$

Transition model, given a current state s and an action $a = (v, d)$ gives us the successor state s'.

$$\mathcal{T}_\gamma : \mathcal{A}_\gamma \times \mathcal{S}_\gamma \rightarrow \mathcal{P}(\mathcal{S}_\gamma)$$

Concretely: $\mathcal{T}_\gamma((v, d), s) := \{ s' \}$ where s' is the successor state (we are updating the variable assignment) which will evaluate as follows:

$$s'(w) := \begin{cases} d, & \text{if } w = v \\ s(w), & \text{if } w \in \mathrm{dom}(s) \text{ and } w \neq v \end{cases} \quad \text{and} \quad \mathrm{dom}(s') := \mathrm{dom}(s) \cup \{v\}.$$

Def 4.15. Backtracking Search: Backtracking search is the basic uninformed algorithm for solving CSPs. It is DFS with two improvements, namely:

1. One variable at a time
   - variable assignments are commutative, fixing the order saves us time!
   - i.e. starting from `state(WA=red)` → `state(WA=red, NT=green)` is the same as starting from `state(NT=green)` → `state(NT=green, WA=red)` [1]
   - We only need to consider assignments to a single variable at each step
2. Check constraints as you go
   - Consider only values which do not conflict previous assignments
   - That is not a goal test! We can think about it as incremental goal test.

The pseudocode for backtracking search is shown in Algorithm 1

Algorithm 1 Backtracking Search

 1: **function** BACKTRACKING-SEARCH(csp)
 2:    **return** RECURSIVE-BACKTRACKING({}, csp)
 3: **end function**
 4: **function** RECURSIVE-BACKTRACKING(assignment, csp)
 5:    **if** assignment is complete **then**
 6:       **return** assignment
 7:    **end if**
 8:    var ← SELECT-UNASSIGNED-VARIABLE(Variables[csp], assignment, csp)
 9:    **for** each value in ORDER-DOMAIN-VALUES(var, assignment, csp) **do**
10:       **if** value is consistent with assignment given Constraints[csp] **then**
11:          add {var = value} to assignment
12:          result ← RECURSIVE-BACKTRACKING(assignment, csp)
13:          **if** result ≠ failure **then**
14:             **return** result
15:          **end if**
16:          remove {var = value} from assignment
17:       **end if**
18:    **end for**
19:    **return** failure
20: **end function**

> **Note**
>
> Note that the exact behaviors of the function: `SELECT-UNASSIGNED-VARIABLE` and `ORDER-DOMAIN-VALUES` are not specified yet

Def 4.16. Forward Checking: Forward checking is a filtering (inference) technique that improves the general uninformed backtracking seach. It propagates information about illegal values. Whenever a variable $v \in V$ is assigned by $d \in D_v$ ($\varphi(v) = d$), we delete all values that are inconsistent with $\varphi(v)$ from every D_u for all variables u connected with v by a constraint.
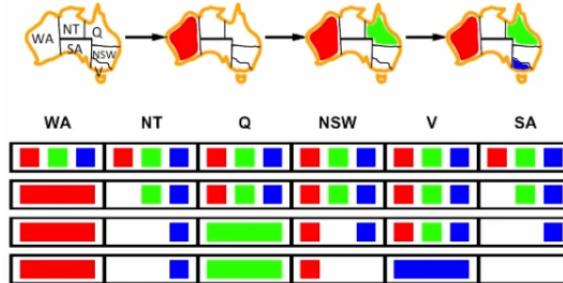
> **Note**
>
> The idea is to keep track of domains for unassigned variables and cross off bad options. In forward checking we cross off values that violate a constraint when added to the existing assignment (look ahead).

forward checking propagates information from assigned to unassigned variables, but doesn't provide early detection for all failures. For example, (remember Australia map):

- Assume we chose to assign `WA=Red`
- Because `NT, SA` are neighbors, we remove `Red` from their domains

---

[1] I am referring to the known Australia map example mentioned in the lecture notes

- Assume we pick next `Q=Green`

- Because `NT, SA, NSW` are neighbors, we remove `Green` from their domains

- At this point, forward checking thinks we are in good shape, but we are already doomed! *why?* Because `NT, SA` are neighbors and they both are left with only `Blue` in their domains!

- Assuming, next we choose `V=Green`, we cross off `Green` from `NSW, SA`, now `SA` is left with no colors, only then, we know we failed, so we backtrack.
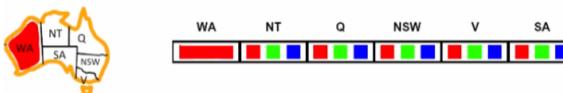

Def 4.17. Arc Consistency: Let $\gamma := \langle V, D, C \rangle$ be a constraint network, a variable $u \in V$ is **arc consistent** relative to another variable $v \in V$ if either $C_{uv} \notin C$, or for every value $d \in D_u$ there exist a value $d' \in D_v$ such that $(d, d') \in C_{uv}$. The constraint network γ is **arc consistent** if every variable $u \in V$ is **arc consistent** relative to every other variable $v \in V$.
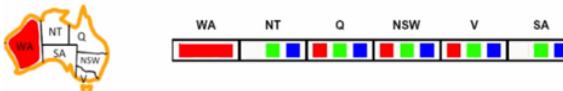
> **Note**
>
> Note that when we are checking some arc $x \rightarrow y$ and find it inconsistent, we remove values from D_x to make it consistent. When doing so, we need to check every other arc $z \rightarrow x$ (where the *tail* of the arc we checked is the *head* of other arcs) even if it was checked before (because now we have fewer options, it is not necessarily consistent anymore!)

Example:
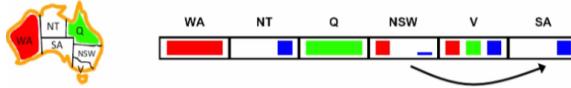
- Checking `NT -> WA` ⤳ removing `Red` from the domain of `NT`

- Checking `SA -> WA` ⤳ removing `Red` from the domain of `SA`

- All arcs are consistent now
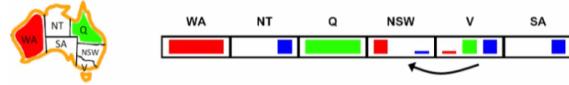
Assume we pick next `Q=Green`, and therefore remove `Green` from `NT, NSW, SA`:
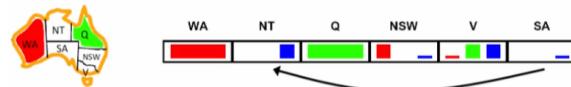
- Looking at `V -> NSW` ⤳ consistent

- Looking at `SA -> NSW` ⤳ consistent

- Looking at `NSW -> SA` ⤳ not consistent! If we choose `NSW = Blue`, nothing I can choose in `SA` that would not violate the constraint!

- We make `NSW -> SA` consistent by removing `Blue` from the domain of `NSW`

- Now, we check again the arc where `NSW` was the head.

- Looking at `V -> NSW` ⇝ not consistent

- Looking at `SA -> NSW` ⇝ consistent

- Continue checking arcs. Assume looking at `SA -> NT` ⇝ not consistent

- Now `SA` domain is empty, we know we failed, so we backtrack!

- Notice that we detect failure earlier than forward checking!

The pseudocode for the arc consistency algorithm (called `AC-3`) is shown in Algorithm 2

Algorithm 2 AC-3

1: **function** AC-3(csp)
2:     inputs: csp, a binary CSP with variables $\{X_1, X_2, \ldots, X_n\}$
3:     local variables: *queue*, a queue of arcs, initially all the arcs in *csp*
4:     **while** queue is not empty **do**
5:         $(X_i, X_j) \leftarrow$ REMOVE-FIRST(queue)
6:         **if** REMOVE-INCONSISTENT-VALUES((X_i, X_j)) **then**
7:             **for** each X_k in Neighbors[X_i] **do**
8:                 add (X_k, X_i) to queue
9:             **end for**
10:         **end if**
11:     **end while**
12:     **return** csp
13: **end function**
14: **function** REMOVE-INCONSISTENT-VALUES((X_i, X_j))
15:     removed $\leftarrow$ false
16:     **for** each x in Domain[X_i] **do**
17:         **if** no value y in Domain[X_j] allows (x, y) to satisfy the constraint $X_i \leftrightarrow X_j$ **then**
18:             delete x from Domain[X_i]
19:             removed $\leftarrow$ true
20:         **end if**
21:     **end for**
22:     **return** removed
23: **end function**

- In the worst case, we have a fully connected constraint network of n variables, hence we will have a total of $n(n-1)$ arcs.

- For one arc $x \rightarrow y$, if the number of values each variable can take is d, to make this arc consistent, we have one variable that has d possibilities (*the tail*) and the *head* too (another d possibilities), and we want to make sure for everything in the *tail* there is some value in the *head* that we can assign. That requires $d \cdot d$ checks in the worst case $= d^2$.

- Moreover, it is not enough to check each arc once. We `enqueue` an already `dequeued` arc whenever we eliminate a value from the domain of the arc's *head*. In the worst case, we might eliminate all the values of a variable, i.e. we put an arc back d times.
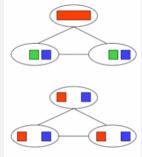
- Therefore, the total run time for `AC-3` is $\mathcal{O}((n(n-1)) \cdot d^2 \cdot d) = \mathcal{O}(n^2 d^3)$

> **Note**
>
> Notice that the run time $\approx$ polynomial and yet detecting all possible future problems is NP-Hard because of backtracking. Arc consistency does not guarantee finding a solution. After enforcing arc consistency, we might have:
> - one solution left
> - multiple solutions left
> - no solutions left (and not know it!)
>
> Moreover, the algorithm still runs inside backtracking search.
>
>

Def 4.18. Minimum Remaining Values: The **MRV** heuristic for backtracking search (*a.k.a Most Constrained Variable First*) selects the unassigned variable with the fewest legal values given the current partial assignment φ. Formally, it chooses the variable v that minimizes $\left| \{ d \in D_v \mid \varphi \cup \{v \mapsto d\} \text{ is consistent} \} \right|$.

> **Note**
>
> MRV $\mapsto$ we want to fail early! Hence, MRV ordering is also called fail-fast ordering.

Def 4.19. Least Constraining Value: Given a variable v, the LCV heuristic chooses the least constraining value for v, i.e. the one that rules out the fewest values in the remaining variables. For the current partial assignment φ and a chosen variable v, we pick a value $d \in D_v$ that minimizes $|d' \in D_u \mid u \notin \mathrm{dom}(\varphi), C_{uv} \in C, \text{ and } (d', d) \notin C_{uv}|$

Def 4.20. Independent Subproblems: Assume we have independent connected components of a constraint graph. We treat each component as a separate CSP since it has no constraints outside its variables.
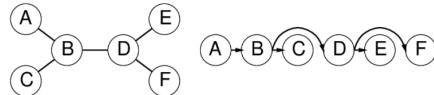
Remember, a naive solver (brute-force) solves a discrete CSPs with domain size d and n variables in $\mathcal{O}(d^n)$ (exponential in n). If that same problem has independent subproblems each with c variables, then we have $\frac{n}{c}$ subproblems where each one in the worst case is $\mathcal{O}(d^c)$. So the total time will be $\mathcal{O}(\frac{n}{c} d^c)$, that is, linear in n and exponential in c. That is very fast for small c.

Def 4.21. Decomposition: The process of decomposing a constraint network into components.

Def 4.22. Tree-Structured CSP: We call a CSP **tree-structured** iff its constraint graph is acyclic.
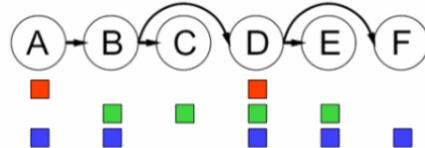
Theorem 4.1. A Tree-Structured CSP can be solved in $\mathcal{O}(nd^2)$

To solve a Tree-Structured CSP, we choose a variable as root and order the variables from root to leaves such that every node's parent precedes it in the ordering:
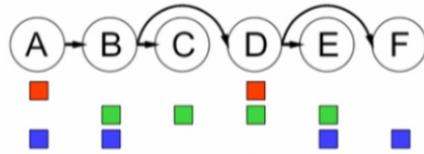
- Backward pass: For $i = n$ and down to 2, apply `Remove-Inconsistent-Values((Parent(X_i), X_i))`

- Forward pass: For $i = 1$ and up to n, assign X_i consistently with `Parent(X_i)`

For example, consider the following domains:
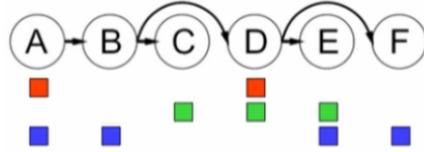
Backward Pass:
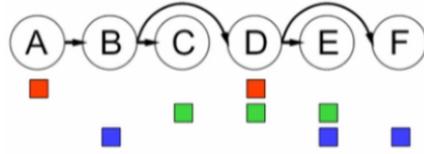
- Start with F, checking $D \to F \rightsquigarrow$ not consistent:

- Next is E, checking $D \to E \rightsquigarrow$ consistent.
- Next is D, checking $B \to D \rightsquigarrow$ consistent.
- Next is C, checking $B \to C \rightsquigarrow$ not consistent:

- Next is B, checking $A \to B \rightsquigarrow$ not consistent:

Forward Pass:

$\varphi(A) = \texttt{Red}$, $\varphi(B) = \texttt{Blue}$, $\varphi(C) = \texttt{Green}$, $\varphi(D) = \texttt{Green}$, $\varphi(E) = \texttt{Blue}$, $\varphi(F) = \texttt{Blue}$

> **Note**
>
> After the backward pass on a tree-structured CSP, every root-to-leaf arc becomes consistent, and no arc ever needs to be enqueued again. The key reason is the direction in which consistency is enforced. Starting from the leaves and moving upward, each arc $x \to y$ is processed only after all of x's children have been handled. When enforcing consistency on $x \to y$, we only remove values from the domain of the tail x. Since all arcs of the form $v \to x$ were already processed earlier, there will be no later step that removes values from the head y of any previously processed arc. In contrast to general AC algorithms, where pruning the head forces re-checking incoming arcs, the tree structure guarantees that once an arc's head has been finalized, it is never modified again. This is why no arc needs to be re-enqueued: domain reductions always flow upward toward the root, never back down.

> **Note**
>
> If root-to-leaf arcs are consistent, the forward pass will not backtrack and will go straight to the solution!

Def 4.23. Conditioning: Instantiate a variable, prune its neighbors' domains

Def 4.24. Cutset conditioning: Instantiate in all ways a set of variables such that the remaining constraint graph is a tree.

> **Note**
>
> A cutset of size c gives runtime $\mathcal{O}(d^c(n-c)d^2)$:
> - Because we are left with a tree of size $n - c$, we have $\mathcal{O}((n-c)d^2)$
> - Because we have to look at all possible assignments of the cutset, we have $\mathcal{O}(d^c)$
>
> and that is very fast for small c!

5   Formal Systems

Def 5.1. Logical System: A **logical system** (or simply a **logic**) is a triple $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$, where

- $\mathcal{L}$ is a set of propositions

- $\mathcal{M}$ is a set of **models**,

- $\vDash$ is a relation $\vDash \subseteq \mathcal{M} \times \mathcal{L}$ called the **satisfaction relation**. We read $\mathcal{M} \vDash A$ as $\mathcal{M}$ **satisfies** A and correspondingly $\mathcal{M} \nvDash A$ as $\mathcal{M}$ **falsifies** A

Let $\langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, $M \in \mathcal{M}$ a model and $A \in \mathcal{L}$ a proposition. Then we say that A is

- **satisfied** by M iff $M \vDash A$

- **satisfiable** iff A is satisfied by some model

- **unsatisfiable** iff A is not satisfiable

- **falsified** by M iff $M \nvDash A$

- **valid** or **unfalsifiable** (write $\vDash A$) iff A is satisfied by **every** model $M \in \mathcal{M}$

- **invalid** or **falsifiable** (write $\nvDash A$) iff A is not valid

Def 5.2. Derivation Relation ($\vdash$):   Let $\mathcal{L}$ be a formal language. For any set of propositions $\mathcal{H} \subseteq \mathcal{L}$ (*called the context or hypotheses*) and any proposition $A \in \mathcal{L}$, we write $\mathcal{H} \vdash A$ to say that A is derivable from $\mathcal{H}$. We call a relation $\vdash \subseteq \mathcal{P}(\mathcal{L}) \times \mathcal{L}$ a **derivation relation** for $\mathcal{L}$, iff

- $\mathcal{H} \vdash A$, if $A \in \mathcal{H}$ (proof reflexive),

- $\mathcal{H} \vdash A$ and $(\mathcal{H}' \cup \{A\}) \vdash B$ imply $(\mathcal{H} \cup \mathcal{H}') \vdash B$ (proof transitive),

- $\mathcal{H} \vdash A$ and $\mathcal{H} \subseteq \mathcal{H}'$ imply $\mathcal{H}' \vdash A$ (monotonic admits weakening).

Def 5.3. Inference Rules: Let $\mathcal{L}$ be a formal language, then an **inference rule** over $\mathcal{L}$ is decidable $n+1$-ary relation on $\mathcal{L}$. Inference rules are traditionally written as:

$$\frac{A_1 \quad \dots \quad A_n}{C} \, \mathcal{N}$$

where $A_1, \cdots, A_n$ and C are schemata for words in $\mathcal{L}$ and $\mathcal{N}$ is a name. The A_i are called **assumptions** of $\mathcal{N}$, and C is called its **conclusion**.

Any $n+1$-tuple $\frac{a_1 \quad \dots \quad a_n}{c}$ in $\mathcal{N}$ is called an **application** of $\mathcal{N}$ and we say that we apply $\mathcal{N}$ to a set M of words with $a_1, \cdots, a_n \in M$ to obtain c.

Def 5.4. Axiom: An inference rule without assumptions is called an **axiom**

Def 5.5. Calculus: A **calculus** (or **inference system**) is a formal language $\mathcal{L}$ equipped with a set $\mathcal{C}$ of inference rules over $\mathcal{L}$.

Def 5.6. $\mathcal{C}$-derivation: Let $S := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, and $\mathcal{C}$ a calculus for $\mathcal{L}$, then a $\mathcal{C}$-derivation of a proposition $P \in \mathcal{L}$ from a set $\mathcal{H} \subseteq \mathcal{L}$ of hypotheses (write $\mathcal{H} \vdash_\mathcal{C} P$) is a sequence $A_1, \cdots, A_m$ of propositions where:

- $A_m = P$,

- for all $1 \leq i \leq m$, either $A_i \in \mathcal{H}$, or,

- there is an inference rule $\dfrac{A_{l_1} \quad \dots \quad A_{l_k}}{A_i} \, \mathcal{N}$ in $\mathcal{C}$ with $l_j < i$ for all $j \leq k$

We can also see a $\mathcal{C}$-derivation as a **derivation tree**, where A_{l_j} are the children of the node A_i

Def 5.7. $\mathcal{C}$-refutation: Let $\mathcal{C}$ be a calculus, then a formula set Φ is called $\mathcal{C}$-**refutable** if there is a $\mathcal{C}$-**refutation**, i.e. a $\mathcal{C}$-derivation of a contradiction from Φ. The act of finding a **refutation** for Φ is called **refuting** Φ

Def 5.8. Derivation System: We call $\langle \mathcal{L}, \mathcal{M}, \vDash, \vdash \rangle$ a **derivation system**, iff $\langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ is a logical system, and $\vdash$ a derivation relation for $\mathcal{L}$.

Assertion Let $\langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, and $\mathcal{C}$ a calculus, then $\vdash_\mathcal{C}$ is a derivation relation and thus $\langle \mathcal{L}, \mathcal{M}, \vDash, \vdash_\mathcal{C} \rangle$ is a derivation system

Def 5.9. Formal System: Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, and let $\mathcal{C}$ be a calculus over the language $\mathcal{L}$. Then the pair $\langle \mathcal{S}, \mathcal{C} \rangle$ is called a **formal system**.

> **Note**
>
> Let $\mathcal{C}$ be a calculus, then a $\mathcal{C}$-derivation $\emptyset \vdash_{\mathcal{C}} A$ is called a proof of A and if one exists (write $\vdash_{\mathcal{C}} A$) then A is called a $\mathcal{C}$-theorem.

Def 5.10. Admissible: An inference rule $\mathcal{J}$ is called **admissible** in a calculus $\mathcal{C}$, if the extension of $\mathcal{C}$ by $\mathcal{J}$ does not yield new theorems

Def 5.11. Derived Rules: An inference rule $\dfrac{A_1 \quad \dots \quad A_n}{C} \mathcal{N}$ is called **derived / derivable** in a calculus $\mathcal{C}$, if there is a $\mathcal{C}$-derivation $A_1, \cdots, A_n \vdash_{\mathcal{C}} C$.

> **Note**
>
> Derivable inference rules are admissible, but not the other way around.

Theorem 5.1. It is not true that every admissible inference rule is derivable

Proof.

1. Construct an empty calculus $\mathcal{C}_\emptyset$

2. Since we have no axioms and no inference rules, then the set of theorems is also empty $\text{Th}(\mathcal{C}_\emptyset) = \emptyset$

3. Let $\mathcal{J}$ be an arbitrary inference rule

$$\frac{A_1 \quad \dots \quad A_n}{B} \mathcal{J}$$

4. Now we extend our empty calculus by this rule $\mathcal{C}_\emptyset \cup \mathcal{J}$

5. Since we have no axioms, the rule $\mathcal{J}$ has no *input* to operate on, therefore, the set of theorems stays empty $\text{Th}(\mathcal{C}_\emptyset \cup \mathcal{J}) = \emptyset$

6. Thus, $\mathcal{J}$ is admissible.

7. Since $\mathcal{C}_\emptyset$ has no rule and no axioms, it is impossible to construt steps from $A_1, \cdots, A_n$ to B

8. Hence, $\mathcal{J}$ is not derivable

9. Since $\mathcal{J}$ is admissible but not derived, the statement *"every admissible inference rule is derivable"* is False, hence, it is not true that every admissible inference rule is derivable.

$\square$

6 Propositional Logic

Def 6.1. Propositional Logic: Propositional Logic (PL^0) is a logical system $\langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ Where

- the formal language $\mathcal{L}$ is defined as the well-formed formulae of propositional logic: $\text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0})$,
- the set $\mathcal{M}$ of models is given by the set $\mathcal{K}_0$ of total functions $\varphi : \mathcal{V}_0 \to \mathcal{D}_0$ (i.e., propositional variable assignments),
- and $\varphi \vDash A$ iff $\mathcal{I}_\varphi(A) = T$ (a proposition is satisfiable by model φ if the interpretation of it -under that φ- is true)

Hence, propositional logic is a logical system $\langle \text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0}), \mathcal{K}_0, \vDash \rangle$

Def 6.2. Formulae of PL^0: The formulae of PL^0 are made up from

- propositional variables $\mathcal{V}_{\text{PL}^0}$ (or just $\mathcal{V}_0$) : $\mathcal{V}_0 := \{P, Q, R, \cdots\}$ (countably infinite)
- propositional signature Σ_{PL^0} (or just Σ_0): $\Sigma_0 := \{\top, \bot, \neg, \vee, \wedge, \Rightarrow, \Leftrightarrow, \cdots\}$ (called *connectives*)

We define the set $\text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0})$ of **well-formed propositional formulae (wffs)** as:

- propositional variables
- logical constants $\top$ and $\bot$
- negations $\neg A$
- conjunctions $A \wedge B$
- disjunctions $A \vee B$
- implications $A \Rightarrow B$, and
- equivalence (biimplications) $A \Leftrightarrow B$

where $A, B \in \text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0})$ themselves.

Def 6.3. Grammar for PL^0: The set of propositional formulae $\{A \mid A \in \text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0})\}$ is given by the following grammar:

$$
\begin{aligned}
X &::= P \mid Q \mid R \mid \cdots \quad \text{(\textit{propositional variables})} \\
A &::= X \mid \top \mid \bot \mid \neg A \mid A \wedge A \mid A \vee A \mid A \Rightarrow A \mid A \Leftrightarrow A
\end{aligned}
$$

Def 6.4. Canonical Model: We call a logical system $\langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ a **single-model** with **canonical model** M iff $\mathcal{M} = \{M\}$

Def 6.5. PL^0 Models: A model $\mathcal{M} := \langle \mathcal{D}_0, \mathcal{I} \rangle$ for PL^0 consists of

- the universe $\mathcal{D}_0 := \{T, F\}$ (truth values)
- the interpretation $\mathcal{I}$ that assigns truth values to essential connectives
    1. $\mathcal{I}(\neg) : \mathcal{D}_0 \to \mathcal{D}_0; T \mapsto F, F \mapsto T$
    2. $\mathcal{I}(\wedge) : \mathcal{D}_0 \times \mathcal{D}_0 \to \mathcal{D}_0; \langle \alpha, \beta \rangle \mapsto T$, iff $\alpha = \beta = T$
    3. $\mathcal{I}(\top) = T$
    4. $\mathcal{I}(\bot) = F$

> **Note**
>
> PL^0 is a single-model logical system with canonical model $\langle \mathcal{D}_0, \mathcal{I} \rangle$

Def 6.6. Variable Assignment: PL^0 uses a total variable assignment $\varphi : \mathcal{V}_0 \to \mathcal{D}_0$ that assigns truth values to propositional variables.

Def 6.7. Value Function: PL^0 uses a value function $\mathcal{I}_\varphi : \text{wff}_0(\Sigma_{\text{PL}^0}, \mathcal{V}_{\text{PL}^0}) \to \mathcal{D}_0$ that assigns truth values to PL^0 formulae. It is defined as follows:

- $\mathcal{I}_\varphi(P) = \varphi(P)$
- $\mathcal{I}_\varphi(\neg A) = \mathcal{I}(\neg)(\mathcal{I}_\varphi(A))$

- $\mathcal{I}_\varphi(A \wedge B) = \mathcal{I}(\wedge)(\mathcal{I}_\varphi(A), \mathcal{I}_\varphi(B))$

Value Function Example: Compute $\mathcal{I}_\varphi(P \vee Q)$ under the assignment $\varphi := \{P \mapsto F, Q \mapsto T\}$

$$\mathcal{I}_\varphi(P \vee Q) = \mathcal{I}_\varphi(\neg(\neg P \wedge \neg Q))$$
$$\mathcal{I}_\varphi(\neg P) = \mathcal{I}(\neg)(\mathcal{I}_\varphi(P))$$
$$\mathcal{I}_\varphi(\neg Q) = \mathcal{I}(\neg)(\mathcal{I}_\varphi(Q))$$
$$\mathcal{I}_\varphi(\neg P \wedge \neg Q) = \mathcal{I}(\wedge)(\mathcal{I}_\varphi(\neg P), \mathcal{I}_\varphi(\neg Q))$$
$$\mathcal{I}_\varphi(P \vee Q) = \mathcal{I}(\neg)(\mathcal{I}_\varphi(\neg P \wedge \neg Q))$$
$$= \mathcal{I}(\neg)(\mathcal{I}(\wedge)(\mathcal{I}_\varphi(\neg P), \mathcal{I}_\varphi(\neg Q)))$$
$$= \mathcal{I}(\neg)(\mathcal{I}(\wedge)(\mathcal{I}(\neg)(\mathcal{I}_\varphi(P)), \mathcal{I}(\neg)(\mathcal{I}_\varphi(Q))))$$

We know that
- $\mathcal{I}_\varphi(P) = \varphi(P) = F$, and
- $\mathcal{I}_\varphi(Q) = \varphi(Q) = T$

Hence:

$$= \mathcal{I}(\neg)(\mathcal{I}(\wedge)(\mathcal{I}(\neg)(F), \mathcal{I}(\neg)(T)))$$
$$= \mathcal{I}(\neg)(\mathcal{I}(\wedge)(T, F))$$
$$= \mathcal{I}(\neg)(F)$$
$$= T$$

Def 6.8. Variable Occurrence: A function that maps a PL^0 formula to the set of variable occurred in that formula. It is defined as:

- $\text{Var}(P) = \{P\}$

- $\text{Var}(\top) = \emptyset$

- $\text{Var}(\bot) = \emptyset$

- $\text{Var}(\neg A) = \text{Var}(A)$

- $\text{Var}(A \circ B) = \text{Var}(A) \cup \text{Var}(B)$ where $\circ \in \{\wedge, \vee, \Rightarrow, \Leftrightarrow\}$

Def 6.9. Ground Formula: A PL^0 formula A is called **ground** iff $\text{Var}(A) = \emptyset$

Alternative Notation: write $[\![A]\!]_\varphi$ for $\mathcal{I}_\varphi(A)$ (and $[\![A]\!]$ if A is ground)

Def 6.10. Equivalent: Two formulae A and B are called **equivalent**, iff $\mathcal{I}_\varphi(A) = \mathcal{I}_\varphi(B)$ for all variable assignments φ

Def 6.11. PL^0 Identities: Table 3 shows the identities in PL^0

Table 3: PL^0 Identities

Name	for $\wedge$	for $\vee$
Idempotence	$\varphi \wedge \varphi = \varphi$	$\varphi \vee \varphi = \varphi$
Identity	$\varphi \wedge \top = \varphi$	$\varphi \vee \bot = \varphi$
Absorption 1	$\varphi \wedge \bot = \bot$	$\varphi \vee \top = \top$
Commutativity	$\varphi \wedge \psi = \psi \wedge \varphi$	$\varphi \vee \psi = \psi \vee \varphi$
Associativity	$\varphi \wedge (\psi \wedge \theta) = (\varphi \wedge \psi) \wedge \theta$	$\varphi \vee (\psi \vee \theta) = (\varphi \vee \psi) \vee \theta$
Distributivity	$\varphi \wedge (\psi \vee \theta) = (\varphi \wedge \psi) \vee (\varphi \wedge \theta)$	$\varphi \vee (\psi \wedge \theta) = (\varphi \vee \psi) \wedge (\varphi \vee \theta)$
Absorption 2	$\varphi \wedge (\varphi \vee \theta) = \varphi$	$\varphi \vee (\varphi \wedge \theta) = \varphi$
De Morgan rule	$\neg(\varphi \wedge \psi) = \neg\varphi \vee \neg\psi$	$\neg(\varphi \vee \psi) = \neg\varphi \wedge \neg\psi$
double negation	$\neg\neg\varphi = \varphi$	
Definitions	$\varphi \Rightarrow \psi \ = \ \neg\varphi \vee \psi$	$\varphi \Leftrightarrow \psi \ = \ (\varphi \Rightarrow \psi) \wedge (\psi \Rightarrow \varphi)$

Let $\mathcal{M} := \langle \mathcal{D}_0, \mathcal{I} \rangle$ be our model, then we say that a formula A is

- **true under** φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(A) = T$, (write $\mathcal{M} \vDash^\varphi A$)

- **falsifies** φ in $\mathcal{M}$, iff $\mathcal{I}_\varphi(A) = F$, (write $\mathcal{M} \nvDash^\varphi A$)

- **satisfiable in** $\mathcal{M}$, iff $\mathcal{I}_\varphi(A) = T$ for some assignment φ

- **valid in** M, iff $\mathcal{M} \vDash^\varphi A$ for all variable assignments φ

- **falsifiable in** $\mathcal{M}$, iff $\mathcal{I}_\varphi(A) = F$ for some assignment φ, and

- **unsatisfiable in** $\mathcal{M}$, iff $\mathcal{I}_\varphi(A) = F$ for all variable assignments φ.

> **Note**
>
> Since PL^0 is a single-model logical system, we can omit the explicit reference to the canonical model and write $\varphi \vDash A$ to mean "A is true under φ", recovering the notation from $\mathcal{M} \vDash^\varphi A$.

Def 6.12. Entailment: We say that A **entails** B (write $A \vDash B$), iff $\mathcal{I}_\varphi(B) = T$ for all φ with $\mathcal{I}_\varphi(A) = T$ (i.e. all variable assignments) that makes A true also make B true

Theorem 6.1. $A \vDash B \Rightarrow A \wedge C \vDash B \wedge C$

Proof.

1. Assume that $A \vDash B$ ($\mathcal{H}_1$), we want to show that for all φ that makes $A \wedge C$ true also make $B \wedge C$ true

2. Assume there is some φ that makes $A \wedge C$ true $\rightsquigarrow \mathcal{I}_\varphi(A \wedge C) = T$ ($\mathcal{H}_2$)

3. This means that $\mathcal{I}_\varphi(A) = \mathcal{I}_\varphi(C) = T$

4. Given $\mathcal{H}_1$ we now know that our φ also make B true $\rightsquigarrow \mathcal{I}_\varphi(B) = T$

5. From 3 and 4 and by the definition of conjunction: $\mathcal{I}_\varphi(B \wedge C) = T$

$\square$

Def 6.13. Soundness: Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system and $\mathcal{H} \subseteq \mathcal{L}$ a set of hypotheses, then we call a calculus $\mathcal{C}$ over $\mathcal{L}$ **sound** (correct), iff $\mathcal{H} \vDash A$ whenever $\mathcal{H} \vdash_\mathcal{C} A$

Def 6.14. Completeness: Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system and $\mathcal{H} \subseteq \mathcal{L}$ a set of hypotheses, then we call a calculus $\mathcal{C}$ over $\mathcal{L}$ **complete**, iff $\mathcal{H} \vdash_\mathcal{C} A$ whenever $\mathcal{H} \vDash A$

Def 6.15. Hilbert Calculus: The Hilbert calculus $\mathcal{H}^0$ is a calculus that consists of the following inference rules:

$$\frac{}{P \Rightarrow (Q \Rightarrow P)} \ K \qquad \frac{}{(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))} \ S$$

$$\frac{A \Rightarrow B \quad A}{B} \ \text{MP} \qquad \frac{A}{[B/X](A)} \ \text{Subst}$$

> **Note**
>
> Subst inference rule is admissible but not derivable

Def 6.16. Hilbert Formal System: Using propositional logic and the Hilbert Calculus, we can build a simple formal system:

$$\langle \ \underbrace{\langle \mathcal{L}, \mathcal{M}, \vDash \rangle}_{\text{propositional logic}} \ , \ \underbrace{\mathcal{H}^0}_{\text{Hilbert Calculus}} \ \rangle$$

Example: $C \Rightarrow C$ is a $\mathcal{H}^0$ and here is its proof:

Proof. We show that $\emptyset \vdash_{\mathcal{H}^0} C \Rightarrow C$

1. axiom S gives us:
$$(P \Rightarrow (Q \Rightarrow R)) \Rightarrow ((P \Rightarrow Q) \Rightarrow (P \Rightarrow R))$$

2. We apply Subst with $[C/P], [C \Rightarrow C/Q], [C/R]$, we get:

$$(C \Rightarrow ((C \Rightarrow C) \Rightarrow C)) \Rightarrow ((C \Rightarrow (C \Rightarrow C)) \Rightarrow (C \Rightarrow C))$$

3. axiom K gives us:

$$P \Rightarrow (Q \Rightarrow P)$$

4. We apply Subst with $[C/P], [C \Rightarrow C/Q]$, we get:

$$C \Rightarrow ((C \Rightarrow C) \Rightarrow C)$$

5. We use MP on 4 and 2, we get:

$$(C \Rightarrow (C \Rightarrow C)) \Rightarrow (C \Rightarrow C)$$

6. axiom K gives us:

$$P \Rightarrow (Q \Rightarrow P)$$

7. We apply Subst with $[C/P], [C/Q]$, we get:

$$C \Rightarrow (C \Rightarrow C)$$

8. We apply MP on 7 and 5, we get:

$$C \Rightarrow C$$

$\square$

Def 6.17. $\mathcal{ND}_0$: The **propositional natural deduction calculus** $\mathcal{ND}_0$ has inference rules for the introduction and elimination of connectives. $\mathcal{ND}_0$ rules are shown in figure 1

Figure 1: Propositional Natural Deduction Rules

Note

Rules below the dashed line "- - -" are all derived rules from the original set above that line

Def 6.18. $\mathcal{ND}_0$ **Derivation Tree:** Given a set $\mathcal{H} \subseteq \mathrm{wff}_0(\Sigma_{\mathrm{PL}^0}, \mathcal{V}_{\mathrm{PL}^0})$ of assumptions and a conclusion C, we write $\mathcal{H} \vdash_{\mathcal{ND}_0} C$, iff there is an $\mathcal{ND}_0$ derivation tree whose leaves are in $\mathcal{H}$

Def 6.19. Hypothetical Reasoning: Notice that the rule $(\Rightarrow I)$ proves $A \Rightarrow B$ by exhibiting an $\mathcal{ND}_0$ derivation D (*depicted by the three vertical dots*) of B from a **local hypothesis** A. $(\Rightarrow I)$ then **discharges** the local hypothesis (*get rid of A, which can only be used in D*) and concludes $A \Rightarrow B$. This mode of reasoning is called **hypothetical reasoning** (proof by local hypothesis).

> **Note**
>
> The $\mathcal{TND}$ axiom in the $\mathcal{ND}_0$ calculus is called *the law/principle of excluded middle/third* (in latin *tertium non datur*) is only acceptable in **classical logic**, intuitionistic/constructive logic/mathematics does not affirm that law. (That applies for the derived rule $(\neg E)$ too)

$$\frac{}{\Gamma, A \vdash A} \; \mathrm{Ax} \qquad\qquad \frac{\Gamma \vdash B}{\Gamma, A \vdash B} \; \text{weaken} \qquad\qquad \frac{}{\Gamma \vdash A \vee \neg A} \; \mathcal{TND}$$

$$\frac{\Gamma \vdash A \quad \Gamma \vdash B}{\Gamma \vdash A \wedge B} \; \wedge I \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash A} \; \wedge E_l \qquad\qquad \frac{\Gamma \vdash A \wedge B}{\Gamma \vdash B} \; \wedge E_r$$

$$\frac{\Gamma \vdash A}{\Gamma \vdash A \vee B} \; \vee I_l \qquad \frac{\Gamma \vdash A \vee B \quad \Gamma, A \vdash C \quad \Gamma, B \vdash C}{\Gamma \vdash C} \; \vee E \qquad \frac{\Gamma \vdash B}{\Gamma \vdash A \vee B} \; \vee I_r$$

$$\frac{\Gamma, A \vdash B}{\Gamma \vdash A \Rightarrow B} \; \Rightarrow I \qquad\qquad \frac{\Gamma \vdash A \Rightarrow B \quad \Gamma \vdash A}{\Gamma \vdash B} \; \Rightarrow E$$

$$\frac{\Gamma, A \vdash \bot}{\Gamma \vdash \neg A} \; \neg I \qquad\qquad \frac{\Gamma \vdash \neg\neg A}{\Gamma \vdash A} \; \neg E$$

$$\frac{\Gamma \vdash \neg A \quad \Gamma \vdash A}{\Gamma \vdash \bot} \; \bot I \qquad\qquad \frac{\Gamma \vdash \bot}{\Gamma \vdash A} \; \bot E$$

Figure 2: Propositional Sequent Style Natural Deduction Calculus

> **Note**
>
> The sequent style of $\mathcal{ND}_0$ calculus ($\mathcal{ND}_\vdash^0$) is shown in figure 2

Example: Figures 3 to 6 show different $\mathcal{ND}_0$ proof styles for the following propositional formula:

$$(A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C) \Rightarrow C$$

Let $\Gamma = (A \lor B) \land ((A \Rightarrow C) \land (B \Rightarrow C))$.

$$\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \Gamma}\ \text{Ax}}{\Gamma \vdash A \lor B}\ \land E_l \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \Gamma}\ \text{Ax}}{\Gamma, A \vdash \Gamma}\ \text{weaken}}{\Gamma, A \vdash (A \Rightarrow C) \land (B \Rightarrow C)}\ \land E_r}{\Gamma, A \vdash A \Rightarrow C}\ \land E_l \qquad \overline{\Gamma, A \vdash A}\ \text{Ax}}{\Gamma, A \vdash C}\ \Rightarrow E \qquad \dfrac{\dfrac{\dfrac{\dfrac{\overline{\Gamma \vdash \Gamma}\ \text{Ax}}{\Gamma, B \vdash \Gamma}\ \text{weaken}}{\Gamma, B \vdash (A \Rightarrow C) \land (B \Rightarrow C)}\ \land E_r}{\Gamma, B \vdash B \Rightarrow C}\ \land E_r \qquad \overline{\Gamma, B \vdash B}\ \text{Ax}}{\Gamma, B \vdash C}\ \Rightarrow E}{\Gamma \vdash C}\ \lor E}{\vdash \Gamma \Rightarrow C}\ \Rightarrow I$$

Figure 3: Gentzen Sequent Style Proof

Let $\Gamma = (A \lor B) \land ((A \Rightarrow C) \land (B \Rightarrow C))$.

$$\dfrac{\dfrac{[\Gamma]^1}{A \lor B}\ \land E_l \qquad \dfrac{\dfrac{\dfrac{[\Gamma]^1}{(A \Rightarrow C) \land (B \Rightarrow C)}\ \land E_r}{A \Rightarrow C}\ \land E_l \qquad [A]^2}{C}\ \Rightarrow E \qquad \dfrac{\dfrac{\dfrac{[\Gamma]^1}{(A \Rightarrow C) \land (B \Rightarrow C)}\ \land E_r}{B \Rightarrow C}\ \land E_r \qquad [B]^2}{C}\ \Rightarrow E}{\dfrac{C}{\Gamma \Rightarrow C}\ \Rightarrow I^1}\ \lor E^2$$

Figure 4: Gentzen-Prawitz Style Proof

Step	Formula	Rule Applied
(1)	$(A \lor B) \land (A \Rightarrow C) \land (B \Rightarrow C)$	Assumption
(2)	$A \lor B$	$\land E_l$ (on 1)
(3)	$(A \Rightarrow C) \land (B \Rightarrow C)$	$\land E_r$ (on 1)
(4)	$A \Rightarrow C$	$\land E_l$ (on 3)
(5)	$B \Rightarrow C$	$\land E_r$ (on 3)
(6)	A	Assumption (Case 1)
(7)	C	$\Rightarrow E$ (on 4 and 6)
(8)	B	Assumption (Case 2)
(9)	C	$\Rightarrow E$ (on 5 and 8)
(10)	C	$\lor E$ (on 2, 7 and 9)
		Discharges assumptions 6 & 8
(11)	$((A \lor B) \land (A \Rightarrow C) \land (B \Rightarrow C)) \Rightarrow C$	$\Rightarrow I$ (on 1 and 10)
		Discharges assumption 1

Figure 5: Linearized (Tabular) Style Proof

1	$(A \vee B) \wedge ((A \Rightarrow C) \wedge (B \Rightarrow C))$		Assumption
2	$A \vee B$		$\wedge\text{E}_l$ (on 1)
3	$(A \Rightarrow C) \wedge (B \Rightarrow C)$		$\wedge\text{E}_r$ (on 1)
4	$A \Rightarrow C$		$\wedge\text{E}_l$ (on 3)
5	$B \Rightarrow C$		$\wedge\text{E}_r$ (on 3)
6		A	Assumption
7		C	$\Rightarrow \text{E}$ (on 4, 6)
8		B	Assumption
9		C	$\Rightarrow \text{E}$ (on 5, 8)
10	C		$\vee\text{E}$ (on 2, 7, 9)
11	$((A \vee B) \wedge ((A \Rightarrow C) \wedge (B \Rightarrow C))) \Rightarrow C$		$\Rightarrow \text{I}$ (on 1–10)

Figure 6: Fitch Style Proof

Def 6.20. First-Order Signature: A First-Order (Logic) Signature (FOL Signature) is a tuple $\Sigma := \left\langle \Sigma^f, \Sigma^p \right\rangle$ where:

- $\Sigma^f := \bigcup_{k \in \mathbb{N}} \Sigma_k^f$ of **function constants** (or **terms**), where members of Σ_k^f denote the k-ary functions on **individuals**

- $\Sigma^p := \bigcup_{k \in \mathbb{N}} \Sigma_k^p$ of **predicate constants**, where member of Σ_k^p denote k-ary relations among individuals

- Σ_k^f and Σ_k^p are pairwise disjoint, countable sets of symbols for each $k \in \mathbb{N}$.

- A 0-ary **function constant** refers to a single **individual**, therefore, we call it an **individual constant**

Def 6.21. Predicate Logic Without Quantifiers PL$^{\text{nq}}$: Given a FOL Signature Σ, the **formulae** of PL$^{\text{nq}}$ are given by the following grammar:

$$
\begin{aligned}
f^k &\in \Sigma_k^f \\
p^k &\in \Sigma_p^f \\
t &:= f^0 \mid f^k(t_1, \cdots, t_k) \\
A &:= p^k(t_1, \cdots, t_k) \mid \neg A \mid A \wedge A
\end{aligned}
$$

Def 6.22. Well-Formed Terms: We denote the set of all well-formed terms over a FOL Signature Σ with $\text{wff}_\iota(\Sigma)$, and the closed ones with $\text{cwff}_\iota(\Sigma)$

Def 6.23. Well-Formed Formulae: We denote the set of all well-formed formulae over a FOL Signature Σ with $\text{wff}_o(\Sigma)$, and the closed ones with $\text{cwff}_o(\Sigma)$

> **Note**
>
> The Greek letter ι (read *iota*) stands for individuals (objects), however, o (read *omicron*) refers to propositions that can be true of false

Def 6.24. PL$^{\text{nq}}$ Universe: PL$^{\text{nq}}$ uses the universe $\mathcal{D}_{\text{PL}^{\text{nq}}} := \mathcal{D}_0 \cup \mathcal{D}_\iota$ where $\mathcal{D}_0 = \{T, F\}$ is the set of truth values and $\mathcal{D}_\iota \neq \emptyset$ is a non-empty set of individuals

Def 6.25. Interpretation in PL$^{\text{nq}}$: The interpretation function in PL$^{\text{nq}}$ assigns values to constants:

1. We use the interpretation from PL^0
   - $\mathcal{I}(\neg) : \mathcal{D}_0 \to \mathcal{D}_0; T \mapsto F, F \mapsto T$
   - $\mathcal{I}(\wedge) : \mathcal{D}_0 \times \mathcal{D}_0 \to \mathcal{D}_0; \langle \alpha, \beta \rangle \mapsto T$, iff $\alpha = \beta = T$
   - $\mathcal{I}(\top) = T$
   - $\mathcal{I}(\bot) = F$

2. We interpret individual constants as individuals: $\mathcal{I} : \Sigma_0^f \to \mathcal{D}_\iota$

3. We interpret function constants as functions: $\mathcal{I} : \Sigma_{k \neq 0}^f \to \mathcal{D}_\iota^{k \neq 0} \to \mathcal{D}_\iota$

4. We interpret predicate constants as relations: $\mathcal{I} : \Sigma^p_k \to \mathcal{P}(\mathcal{D}^k_\iota)$

> **Note**
>
> This definition naturally includes propositions as a special case. By defining $\mathcal{I} : \Sigma^p_k \to \mathcal{P}(\mathcal{D}^k_\iota)$, we see that for $k = 0$:
>
> $$\mathcal{D}^0_\iota = \{\langle\rangle\} \implies \mathcal{P}(\mathcal{D}^0_\iota) = \{\emptyset, \{\langle\rangle\}\} \cong \{F, T\}$$
>
> Thus, a 0-ary predicate constant is interpreted as a truth value (a proposition), just like in propositional logic.

Def 6.26. Value Function: The value function assigns values to formulae:

1. $\mathcal{I}(f(A_1, \cdots, A_k)) := \mathcal{I}(f)(\mathcal{I}(A_1), \cdots, \mathcal{I}(A_k))$

2. $\mathcal{I}(p(A_1, \cdots, A_k)) := T$, iff $\langle \mathcal{I}(A_1), \cdots, \mathcal{I}(A_k)\rangle \in \mathcal{I}(p)$

3. and Just as in PL^0:

    - $\mathcal{I}(\neg A) := \mathcal{I}(\neg)(\mathcal{I}(A))$

    - $\mathcal{I}(A \wedge B) := \mathcal{I}(\wedge)(\mathcal{I}(A), \mathcal{I}(B))$

Theorem 6.2. PL^{nq} is **isomorphic** to PL^0

Theorem 6.3. Unsatisfiability Theorem. $\mathcal{H} \vDash A \Leftrightarrow \mathcal{H} \cup \{\neg A\}$ is unsatisfiable.

Proof.

1. Assume $H \vDash A$

    - For any φ with $\varphi \vDash H$ we have $\varphi \vDash A$ and thus $\varphi \nvDash (\neg A)$

2. Assume $\mathcal{H} \cup \{\neg A\}$ is unsatisfiable

    - For any φ with $\varphi \vDash \mathcal{H}$ we have $\varphi \nvDash (\neg A)$ and thus $\varphi \vDash A$

$\square$

7 Machine-Oriented Calculi for Propositional Logic

> **Note**
>
> From Theorem 6.3, we observe that entailment can be tested via satisfiablity

Def 7.1. Theorem Proving: Given a formal system $\langle \mathcal{S}, \mathcal{C} \rangle$, where $\mathcal{C}$ is a calculus and $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ is a logical system, the task of **theorem proving** is the task of determining whether $\mathcal{H} \vdash_{\mathcal{C}} C$ for a **conjecture** $C \in \mathcal{L}$ and hypotheses $\mathcal{H} \subseteq \mathcal{L}$

Def 7.2. Automated Theorem Proving: **ATP** is the automation of theorem proving.

Def 7.3. Test Calculus: For a given conjecture A and hypotheses $\mathcal{H}$, a **test calculus** $\mathcal{T}$ tries to derive a refutation $\mathcal{H}, \neg A \vdash_{\mathcal{T}} \bot$ instead of $\mathcal{H} \vdash A$, where $\neg A$ is unsatisfiable iff A is valid and $\bot$ an unsatisfiable proposition.

Def 7.4. Atomic Formulae: A formula is called **atmoic** (or an **atom**) if it does not contain logical constants, otherwise, **complex**

Def 7.5. Labeled Formula: Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, $A \in \mathcal{L}$ a formula, L a **label set**, and $\alpha \in L$ a **label**, then we call a pair $\langle A, \alpha \rangle$ a **labeled formula** and write it as A^{α}. For a set Φ of propositions we use $\Phi^{\alpha} := \{A^{\alpha} \mid A \in \Phi\}$

> **Note**
>
> If the label set is $\mathcal{D}_0 := \{T, F\}$, we call a labeled formula A^T **positive** and A^F **negative**

> **Note**
>
> Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, and A^{α} a labeled formula. Then we say that $M \in \mathcal{M}$ **satisfies** A^{α} (write $M \vDash A^{\alpha}$), iff $\alpha = T$ and $M \vDash A$ or $\alpha = F$ and $M \nvDash A$

Def 7.6. Literal: Let $\mathcal{S} := \langle \mathcal{L}, \mathcal{M}, \vDash \rangle$ be a logical system, $A \in \mathcal{L}$ is an atomic formula, and $\alpha \in \{T, F\}$, then we call the labeled formula A^{α} a **literal**.

Def 7.7. Opposite Literal: For a literal A^{α}, we call the literal A^{β} with $\alpha \neq \beta$ the **opposite literal**.

Def 7.8. CNF: A formula is in **conjunctive normal form (CNF)** if it is $\top$ or a conjunction of disjunctions of literals, i.e. if it is of the form:

$$\bigwedge_{i=1}^{n} \bigvee_{j=1}^{m_i} l_{ij} := \overbrace{(l_{1,1} \vee \cdots \vee l_{1,m_1})}^{\text{Clause } i=1 \text{ has } m_1 \text{ literals}} \wedge \overbrace{(l_{2,1} \vee \cdots \vee l_{2,m_2})}^{\text{Clause } i=2 \text{ has } m_2 \text{ literals}} \wedge \cdots \wedge \underbrace{\overbrace{(l_{n,1} \vee \cdots \vee l_{n,m_n})}^{\text{Clause } i=n \text{ has } m_n \text{ literals}}}_{\text{There are } n \text{ total clauses}}$$

Def 7.9. DNF: A formula is in **disjunctive normal form (DNF)** if it is $\bot$ or a disjunction of conjunctions of literals, i.e. if it is of the form:

$$\bigvee_{i=1}^{n} \bigwedge_{j=1}^{m_i} l_{ij} := \overbrace{(l_{1,1} \wedge \cdots \wedge l_{1,m_1})}^{\text{Term } i=1 \text{ has } m_1 \text{ literals}} \vee \overbrace{(l_{2,1} \wedge \cdots \wedge l_{2,m_2})}^{\text{Term } i=2 \text{ has } m_2 \text{ literals}} \vee \cdots \vee \underbrace{\overbrace{(l_{n,1} \wedge \cdots \wedge l_{n,m_n})}^{\text{Term } i=n \text{ has } m_n \text{ literals}}}_{\text{There are } n \text{ total terms}}$$

> **Note**
>
> $\top$ is in CNF because it represents the **empty conjunction**. (The semantic identity for conjunction is T, so we use the constant $\top$ where $\mathcal{I}(\top) = T$). $\bot$ is in DNF because it represents the **empty disjunction**. (The semantic identity for disjunction is F, so we use the constant $\bot$ where $\mathcal{I}(\bot) = F$).

Def 7.10. Tableau Calculus: A **tableau calculus** is a test calculus that analyzes a labeled formula in a tree to determine satisfiability, its branches correspond to valuations (models)

Def 7.11. Saturated Tableau: We call a tableau **saturated** iff no application of a rule adds new material to any branch.

Def 7.12. Closed and Open Branches: A tableau branch is **closed** if it ends with $\bot$. Otherwise, the branch is **open**.

Def 7.13. Closed Tableau: A tableau is **closed** iff every one of its branches is closed. Otherwise, the tableau is **open**.

> **Note**
>
> It is crucial to distinguish between the **outcome** of a proof and the **completion** of the process:
> - **Closed vs. Open:** This refers to the *logical status*. A tableau is **closed** if a contradiction is found on **every** branch. It is **open** if at least one branch remains consistent.
> - **Saturated vs. Unsaturated:** This refers to the *construction status*. A tableau is **saturated** if no more rules can be applied to any branch.
>
> **Note:** A **closed** tableau is rarely **saturated**. Once a specific branch closes, we stop applying rules to **that branch** for efficiency, even if formulas on that branch could still be expanded.

> **Note**
>
> Open branches in a saturated tableaux yield satisfiying assignments

Def 7.14. $\mathcal{T}_0$: The Propositional Tableau Calculus $\mathcal{T}_0$ has the inference rules shown in figure 7

$$\frac{(A \wedge B)^T}{\begin{array}{c}A^T\\B^T\end{array}}\ \mathcal{T}_0\wedge \qquad\qquad \frac{(A \wedge B)^F}{A^F \mid B^F}\ \mathcal{T}_0\vee$$

$$\frac{(\neg A)^T}{A^F}\ \mathcal{T}_0\neg^T \qquad\qquad \frac{(\neg A)^F}{A^T}\ \mathcal{T}_0\neg^F$$

$$\frac{\begin{array}{c}A^\alpha\\ \vdots\\ A^\beta \quad (\alpha \neq \beta)\end{array}}{\bot}\ \mathcal{T}_0\bot$$

- -

$$\frac{(A \Rightarrow B)^T}{A^F \mid B^T} \qquad\qquad \frac{(A \Rightarrow B)^F}{\begin{array}{c}A^T\\B^F\end{array}} \qquad\qquad \frac{\begin{array}{c}A^T\\(A \Rightarrow B)^T\end{array}}{B^T}$$

$$\frac{(A \vee B)^T}{A^T \mid B^T} \qquad\qquad\qquad\qquad \frac{(A \vee B)^F}{\begin{array}{c}A^F\\B^F\end{array}}$$

$$\frac{(A \iff B)^T}{\begin{array}{c}A^T\\B^T\end{array} \Big| \begin{array}{c}A^F\\B^F\end{array}} \qquad\qquad \frac{(A \iff B)^F}{\begin{array}{c}A^T\\B^F\end{array} \Big| \begin{array}{c}A^F\\B^T\end{array}}$$

Figure 7: Propositional Tableau Calculus $\mathcal{T}_0$

Def 7.15. $\mathcal{T}_0$-Theorem: A is a $\mathcal{T}_0$-theorem ($\vdash_{\mathcal{T}_0} A$), iff there is a closed tableau with A^F at the root.

Def 7.16. : A labeled formula A^α is **valid under** φ, iff $\mathcal{I}_\varphi(A) = \alpha$

Def 7.17. Satisfiable Tableau: A tableau $\mathcal{T}$ is **satisfiable**, iff there is a satisfiable branch $\mathcal{B}$ in $\mathcal{T}$, i.e. if the set of

formulae on $\mathcal{B}$ is satisfiable.

Theorem 7.1. $\mathcal{T}_0$ is sound, i.e. $\Phi \subseteq \mathrm{wff}_0(\Sigma_{\mathrm{PL}^0}, \mathcal{V}_{\mathrm{PL}^0})$ is valid if there is a closed tableau $\mathcal{T}$ for Φ^F.

Theorem 7.2. $\mathcal{T}_0$ is complete, i.e. if $\Phi \subseteq \mathrm{wff}_0(\Sigma_{\mathrm{PL}^0}, \mathcal{V}_{\mathrm{PL}^0})$ is valid, then there is a closed tableau $\mathcal{T}$ for Φ^F.

Lemma 7.3. $\mathcal{T}_0$ terminates, i.e. every $\mathcal{T}_0$ tableau becomes saturated after finitely many rule applications.

Corollary 7.4. $\mathcal{T}_0$ induces a decision procedure for validity in PL^0.

Proof.

- By Lemma 7.3, it is decidable whether $\vdash_{\mathcal{T}_0} A$
- By Theorem 7.1 and Theorem 7.2, $\vdash_{\mathcal{T}_0} A$ iff A is valid

$\square$

Example: Figure 8 shows a $\mathcal{T}_0$ proof of the formula $(A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C) \Rightarrow C$

$$((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C) \Rightarrow C)^F$$
$$((A \vee B) \wedge (A \Rightarrow C) \wedge (B \Rightarrow C))^T$$
$$C^F$$
$$(A \vee B)^T$$
$$(A \Rightarrow C)^T$$
$$(B \Rightarrow C)^T$$
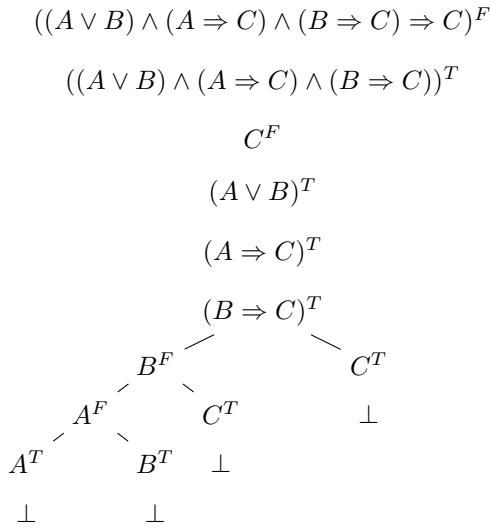
Figure 8: $\mathcal{T}_0$ Proof Example

Def 7.18. Clause: A **clause** is a disjunction $l_1^{\alpha_1} \vee \cdots \vee l_n^{\alpha_n}$ of literals. We use $\square$ for the *"empty"* disjunction (no disjuncts) and call it the **empty clause**. A clause with exactly one literal is called a **unit clause**.

Def 7.19. Clause Set: We often write a **clause set** $\{C_1, \cdots, C_n\}$ as $C_1; \cdots; C_n$, and write $S; T$ for the union of the clause sets S and T, and $S; C$ for the extension of S by a clause C.

Def 7.20. $\mathcal{R}_0$: The propositional resolution calculus $\mathcal{R}_0$ operates on clause sets via a single inference rule:

$$\frac{P^T \vee A \qquad P^F \vee B}{A \vee B} \; \mathcal{R}_0$$

This rule allows to add the **resolvant** $(A \vee B)$ to a clause set which contains the two clauses $P^T \vee A$ and $P^F \vee B$. The literals P^T and P^F are called **cut literals**.

> **Note**
>
> Modus Ponens can be viewed as a specific instance of the $\mathcal{R}_0$ rule. Given the premises $P \to Q$ and P, we first convert the implication to its clausal form $\neg P \vee Q$. Mapping this to the $\mathcal{R}_0$ definition:
>   - Let the first clause be $P^T \vee \bot$.
>   - Let the second clause be $P^F \vee Q$.
>
> The application of the rule then results in:
>
> $$\frac{P^T \vee \bot \qquad P^F \vee Q}{Q} \; \mathcal{R}_0$$
>
> which simplifies to the standard Modus Ponens conclusion Q.

Def 7.21. Resolution Refutation: Let S be a clause set, then we call an $\mathcal{R}_0$-derivation of the empty clause $\square$ from S $\mathcal{R}_0$-**refutation** and write $\mathcal{D} : S \vdash_{\mathcal{R}_0} \square$.

Def 7.22. CNF Calculus: $\mathcal{R}_0$ operates only on clause sets (CNF), to transform formulae into CNF we use the **propositional CNF calculus** (CNF$_0$) which is shown in figure 9.

$$\frac{C \vee (A \vee B)^T}{C \vee A^T \vee B^T} \; \vee^T \qquad\qquad\qquad \frac{C \vee (A \vee B)^F}{(C \vee A^F) \wedge (C \vee B^F)} \; \vee^F$$

$$\frac{C \vee \neg A^T}{C \vee A^F} \; \neg^T \qquad\qquad\qquad \frac{C \vee \neg A^F}{C \vee A^T} \; \neg^F$$

- -

$$\frac{C \vee (A \Rightarrow B)^T}{C \vee A^F \vee B^T} \; \Rightarrow^T \qquad\qquad\qquad \frac{C \vee (A \Rightarrow B)^F}{(C \vee A^T) \wedge (C \vee B^F)} \; \Rightarrow^F$$

$$\frac{C \vee (A \wedge B)^T}{(C \vee A^T) \wedge (C \vee B^T)} \; \wedge^T \qquad\qquad\qquad \frac{C \vee (A \wedge B)^F}{C \vee A^F \vee B^F} \; \wedge^F$$

Figure 9: Propositional CNF Calculus

> **Note**
>
> We write CNF$_0(A^\alpha)$ for the set of all clauses derivable from A^α via CNF$_0$

Example: We want to prove using $\mathcal{R}_0$.
$$(p \Rightarrow q \Rightarrow r) \Rightarrow (p \Rightarrow q) \Rightarrow p \Rightarrow r$$

We label it with F, then we transform it to CNF using CNF$_0$:

$$\frac{\dfrac{\dfrac{((p \Rightarrow (q \Rightarrow r)) \Rightarrow ((p \Rightarrow q) \Rightarrow (p \Rightarrow r)))^F}{(p \Rightarrow (q \Rightarrow r))^T \wedge ((p \Rightarrow q) \Rightarrow (p \Rightarrow r))^F} \; \Rightarrow^F}{(p \Rightarrow (q \Rightarrow r))^T \wedge (p \Rightarrow q)^T \wedge (p \Rightarrow r)^F} \; \Rightarrow^F}{(p \Rightarrow (q \Rightarrow r))^T \wedge (p \Rightarrow q)^T \wedge p^T \wedge r^F} \; \Rightarrow^F$$

Then, expanding the T terms:

$$\frac{\dfrac{\dfrac{(p \Rightarrow (q \Rightarrow r))^T \wedge (p \Rightarrow q)^T \wedge p^T \wedge r^F}{(p \Rightarrow (q \Rightarrow r))^T \wedge (p^F \vee q^T) \wedge p^T \wedge r^F} \; \Rightarrow^T}{(p^F \vee (q \Rightarrow r)^T) \wedge (p^F \vee q^T) \wedge p^T \wedge r^F} \; \Rightarrow^T}{(p^F \vee q^F \vee r^T) \wedge (p^F \vee q^T) \wedge p^T \wedge r^F} \; \Rightarrow^T$$

The final clause set is $\Phi := \left\{ p^F \vee q^F \vee r^T, p^F \vee q^T, p^T, r^F \right\}$

Resolution Proof:

1	$p^F \vee q^F \vee r^T$	initial
2	$p^F \vee q^T$	initial
3	p^T	initial
4	r^F	initial
5	$p^F \vee q^F$	resolve 1.3 with 4.1
6	q^F	resolve 3.1 with 5.1
7	p^F	resolve 2.2 with 6.1
7	$\square$	resolve 3.1 with 7.1

A resolution proof can be represented in a proof tree as shown in figure 10 or in a graph as shown in figure 11
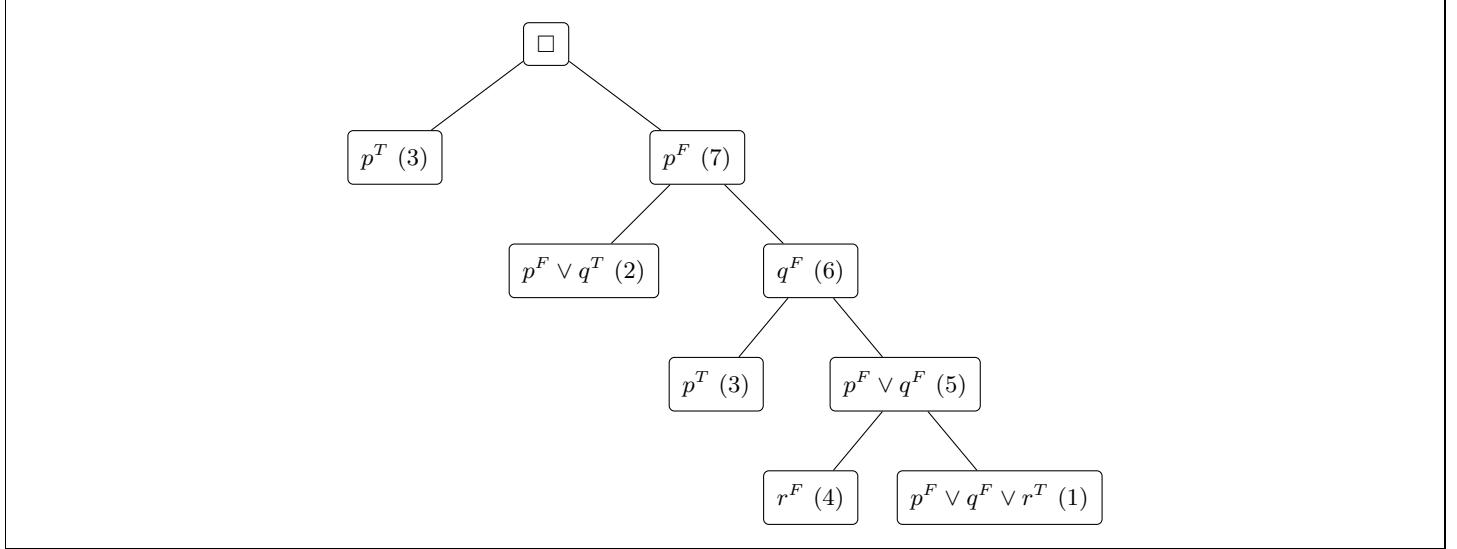
Figure 10: Resolution Proof Tree
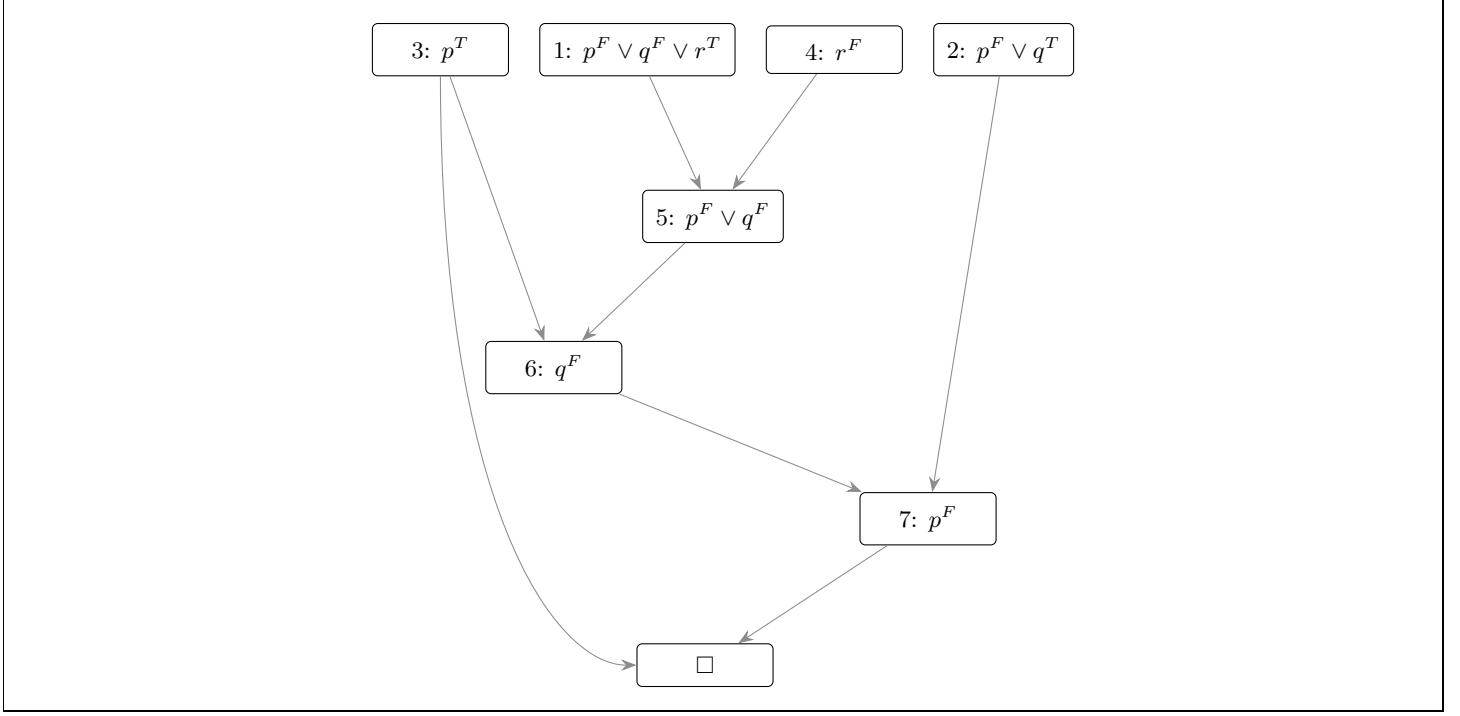
Figure 11: Resolution Proof Graph

Def 7.23. Clause Set Simplification: Let Δ be a clause set, $l \in \Delta$ is a unit clause, and Δ' be Δ where

- all clauses $l \vee C$ have been removed, and
- all clauses $\bar{l} \vee C$ have been shortened to C.

Then Δ is satisfiable iff Δ' is.

We call Δ' the **clause set simplification** of Δ w.r.t. l.

Note: as long as we have unit clauses, we can apply clause set simplification recursively, if we can derive the empty clause, we have a proof!

Example: Let $\Delta := \left\{ p^F \vee q^F \vee r^T, p^F \vee q^T, p^T, r^F \right\}$

- Choose a unit clause, say p^T
- Remove any clause with p^T and shorten any clause with $p^F \vee C$ to just C
- $\Delta' := \left\{ q^F \vee r^T, q^T, r^F \right\}$
- Choose a unit clause, say q^T
- Remove any clause with q^T and shorten any clause $q^F \vee C$ to just C
- $\Delta'' := \left\{ r^T, r^F \right\}$
- Choose any unit clause, say r^T
- Remove any clause with r^T and shorten any clause $r^F \vee C$ to just C
- $\Delta''' := \{\Box\}$

Using clause set simplification, we derived the empty clause without even resolving!

> **Note**
>
> If we apply clause set simplification to a satisfiable clause set, we can read off the model that satisfies the clause set (and we will end up with an empty set of clauses, not the the set containing the empty clause). For example, consider $\Delta := \left\{ p^T, q^T \vee r^T, r^F \right\}$
> - Choose a unit clause, say p^T
> - $\Delta' := \left\{ q^T \vee r^T, r^F \right\}$
> - Choose a unit clause, say r^F
> - $\Delta'' := \left\{ q^T \right\}$
> - Choose a unit clause, say q^T
> - $\Delta''' := \emptyset$
>
> The unit clauses we chose on the way correspond to the model φ that satisfies the clause set, namely $\varphi := \{p \mapsto T, r \mapsto F, q \mapsto T\}$. Which makes sense since $p \wedge (q \vee r) \wedge \neg r$ is indeed satisfiable under φ.
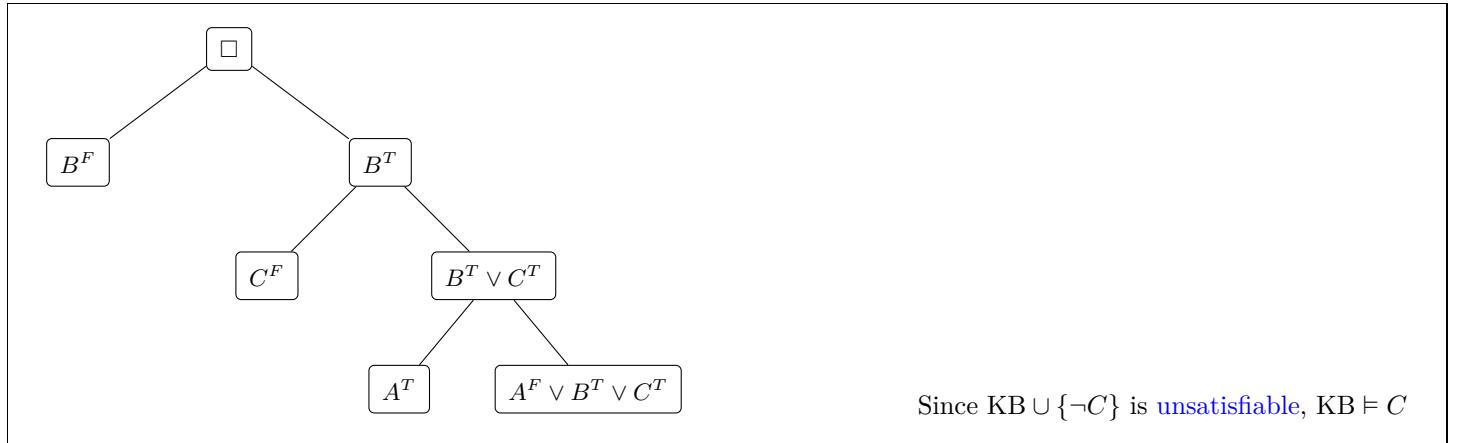
Example: Resolution in the real world

Consider we have a knowledge base KB $:= \{A \Rightarrow B \vee C, A, \neg B\}$, we want to check if KB $\vDash C$. KB $\vDash C$ iff KB $\cup \{\neg C\}$ is unsatisfiable.

First we need to compute $\mathrm{CNF}_0(A \Rightarrow B \vee C)$

- $\mathrm{CNF}_0(A \Rightarrow B \vee C) = \neg A \vee B \vee C$

KB $\cup \{\neg C\} := \{\neg A \vee B \vee C, A, \neg B, \neg C\} := \left\{ A^F \vee B^T \vee C^T, A^T, B^F, C^F \right\}$

Since KB $\cup \{\neg C\}$ is unsatisfiable, KB $\vDash C$

Def 7.24. SAT: Given a propositional formula A, the **Boolean Satisfiablity Problem** $\langle A, V \subset \mathcal{V}_{\mathrm{PL}^0} \rangle$ is to decide whether or not A is satisfiable. We denote the class of all these problems with **SAT**. **SAT** problems are NP-Complete.

Def 7.25. : The tools that address SAT are commonly referred to as **SAT Solvers**.

Theorem 7.5. Given any constraint network $\gamma := \langle V, D, C \rangle$, we can in low-order polynomial time construct a CNF formula $A(\gamma)$ that is satisfiable iff γ is solvable.

SAT → CSP:

Given a SAT instance $S = \langle A, V \rangle$, we define a CSP instance $\gamma := \langle V', D, C \rangle$ and bijections f, f^{-1} mapping satisfying assignments of S to solutions of γ and vice versa:

- we define $V' := V$.
- we define $D := \bigcup_v D_v$ where $D_v := \{T, F\} \quad \forall v \in V'$
- the contraints C consist of a single higher-order constraint which is simply the formula $\rightsquigarrow C := \{A\}$
- The bijections f, f^{-1} are identity mappings because solutions are identical

CSP → SAT:

Given a CSP instance $\gamma := \langle V, D, C \rangle$, we define a SAT instance $S := \langle V', A \rangle$ and bijections f, f^{-1}:

Variables: For each $v \in V$ and each value $a \in D_v$, we introduce a new propositional variable $P_{v,a}$, V' contains all those variables:

$$V' := \{P_{v,a} \mid v \in V \wedge a \in D_v\}$$

Formula: The formula A is defined as the conjunction of three sets of constraints:

$$A := \Phi_{\text{at-least-one}} \wedge \Phi_{\text{at-most-one}} \wedge \Phi_{\text{consistent}}$$

- **At-least-one assignment**: Ensures that each variable v is assigned at least one value from its domain D_v:

$$\Phi_{\text{at-least-one}} = \bigwedge_{v \in V} \left(\bigvee_{a \in D_v} P_{v,a} \right)$$

- **At-most-one assignment**: Ensures that a variable v cannot be assigned more than one value by forbidding all pairs of distinct values:

$$\Phi_{\text{at-most-one}} = \bigwedge_{v \in V} \bigwedge_{\substack{a,b \in D_v \\ a \neq b}} (\neg P_{v,a} \vee \neg P_{v,b})$$

- **Consistency**: Ensures only assignments that satisfy the CSP constraints C_{vw} are allowed by forbidding inconsistent pairs (a, b):

$$\Phi_{\text{consistent}} = \bigwedge_{\{v,w\} \subseteq V} \bigwedge_{(a,b) \notin C_{vw}} (\neg P_{v,a} \vee \neg P_{w,b})$$

Bijections:

- f maps a solution α of the CSP to a satisfying assignment φ for the SAT instance.

$$f(\alpha) = \varphi, \text{ such that } \forall v \in V, \forall a \in D_v : \varphi(P_{v,a}) = \begin{cases} T & \text{if } \alpha(v) = a \\ F & \text{otherwise} \end{cases}$$

- f^{-1} maps a satisfying assignment φ back to a solution α

$$f^{-1}(\varphi) = \alpha \text{ such that } \forall v \in V : \alpha(v) = a \Leftrightarrow \varphi(P_{v,a}) = T$$

Def 7.26. UR: Unit Resolution (UR) is a test calculus consisting of the following inference rule:

$$\frac{C \vee P^\alpha \qquad P^\beta \qquad \alpha \neq \beta}{C} \text{ UR}$$

It is a restricted form of $\mathcal{R}_0$ where at least one of the two clauses is a unit clause.

Def 7.27. UP: Unit Propagation (UP) is an iterative clause set simplification technique triggering a chain reaction of simplifications using only UR.

Def 7.28. Splitting: The process of picking a variiable p and branching the search into two cases: one where p is assumed to be T, and one where p is assumed to be F, resulting in a search tree.

Def 7.29. DPLL: The **Davis Putnam (Logemann-Loverland) Procedure (DPLL)** is a SAT solver called on a clause set Δ and the empty assignments ϵ. It is the standard algorithm foe deciding the satisfiablity of a clause set:

1. Perform Unit Propagation until no more unit clauses exists.

2. Check if the clause set is empty (satisfiable) or contains an empty clause (unsatisfiable)

3. If neither, perform splitting on a variable and repeat.

The pseudocode for DPLL algorithm is shown in Algorithm 3 and Algorithm 4.

Algorithm 3 DPLL

```
 1: function DPLL(Δ, φ)                                    ▷ Returns assignment φ or unsatisfiable
 2:     Δ' ← copy of Δ
 3:     φ' ← φ
 4:     while Δ' contains a unit clause C = Pᵅ do              ▷ Unit Propagation
 5:         φ' ← φ' ∪ {P ↦ α}
 6:         Δ' ← SIMPLIFY(Δ', Pᵅ)
 7:     end while
 8:     if □ ∈ Δ' then                                  ▷ Termination Test: Contradiction
 9:         return unsatisfiable
10:     end if
11:     if Δ' = ∅ then                                    ▷ Termination Test: Success
12:         return φ'
13:     end if
14:     P ← SELECT-UNASSIGNED-VARIABLE(Δ', φ')                       ▷ Splitting Rule
15:     φ'' ← φ' ∪ {P ↦ T}
16:     Δ'' ← SIMPLIFY(Δ', Pᵀ)
17:     res ← DPLL(Δ'', φ'')
18:     if res ≠ unsatisfiable then
19:         return res
20:     end if
21:     φ'' ← φ' ∪ {P ↦ F}
22:     Δ'' ← SIMPLIFY(Δ', Pᶠ)
23:     return DPLL(Δ'', φ'')
24: end function
```

The algorithm uses the following notation:

$$\text{DPLL}(\Delta, \varphi)$$

Line 4: **while** Δ' contains a unit clause $C = P^\alpha$ **do**
Line 5: $\varphi' \leftarrow \varphi' \cup \{P \mapsto \alpha\}$
Line 6: $\Delta' \leftarrow \text{SIMPLIFY}(\Delta', P^\alpha)$
Line 8: **if** $\square \in \Delta'$ **then**
Line 11: **if** $\Delta' = \emptyset$ **then**
Line 12: **return** φ'
Line 14: $P \leftarrow \text{SELECT-UNASSIGNED-VARIABLE}(\Delta', \varphi')$
Line 15: $\varphi'' \leftarrow \varphi' \cup \{P \mapsto T\}$
Line 16: $\Delta'' \leftarrow \text{SIMPLIFY}(\Delta', P^T)$
Line 17: $res \leftarrow \text{DPLL}(\Delta'', \varphi'')$
Line 21: $\varphi'' \leftarrow \varphi' \cup \{P \mapsto F\}$
Line 22: $\Delta'' \leftarrow \text{SIMPLIFY}(\Delta', P^F)$
Line 23: **return** $\text{DPLL}(\Delta'', \varphi'')$

Algorithm 4 Simplify

```
 1: function SIMPLIFY(Δ, Pᵅ)
 2:     Δ' ← ∅
 3:     for each clause C ∈ Δ do
 4:         if Pᵅ ∈ C then
 5:             continue                                ▷ Remove any clause that has Pᵅ
 6:         else if Pᵝ ∈ C then
 7:             C' ← C \ {Pᵝ}                  ▷ Remove Pᵝ from the clause and keep the rest
 8:             Δ' ← Δ' ∪ {C'}
 9:         else
10:             Δ' ← Δ' ∪ {C}                   ▷ Clause does not contain the variable P at all
11:         end if
12:     end for
13:     return Δ'
14: end function
```

DPLL Example: Let $\Delta := \left\{ p^T \vee q^T \vee r^F, p^F \vee q^F, p^T \vee q^F, r^T \right\}$

- We start with the empty assignments $\varphi := \emptyset$

- We see a unit clause r^T, we assign r to T, hence $\varphi := \{r \mapsto T\}$

- We apply UP $\rightsquigarrow \Delta' := \left\{ p^T \vee q^T, p^F \vee q^F, p^T \vee q^F \right\}$

- No unit clauses $\rightsquigarrow$ apply splitting:

- Choose any variable and branch, say we choose p

Branch 1: Assume p^T
We copy the assignment $\varphi_1 := \{r \mapsto T\}$ and we copy the clause set Δ_1'
  – we treat p^T as a unit clause, hence $\varphi_1 := \{r \mapsto T, p \mapsto T\}$
  – we apply UP on Δ_1'
  – $\Delta_1'' := \{q^F\}$
  – we see a unit clause q^F, we assign, $\varphi_1 := \{r \mapsto T, p \mapsto T, q \mapsto F\}$
  – we apply UP on Δ''
  – $\Delta_1''' := \emptyset$
  – we return satisfiable with φ_1

Branch 2: Assume p^F
We copy the assignment $\varphi_2 := \{r \mapsto T\}$ and we copy the clause set Δ_2'
  – we treat p^F as a unit clause, hence $\varphi_2 := \{r \mapsto T, p \mapsto F\}$
  – we apply UP on Δ_2'
  – $\Delta_2'' := \{q^T, q^F\}$
  – we choose a unit clause, say q^T, we assign, $\varphi_2 := \{r \mapsto T, p \mapsto F, q \mapsto T\}$
  – we apply UP on Δ_2''
  – $\Delta_2''' := \{\square\}$
  – we return unsatisfiable for this branch.

The DPLL procedure can be visualized as a search tree as shown in figure 12
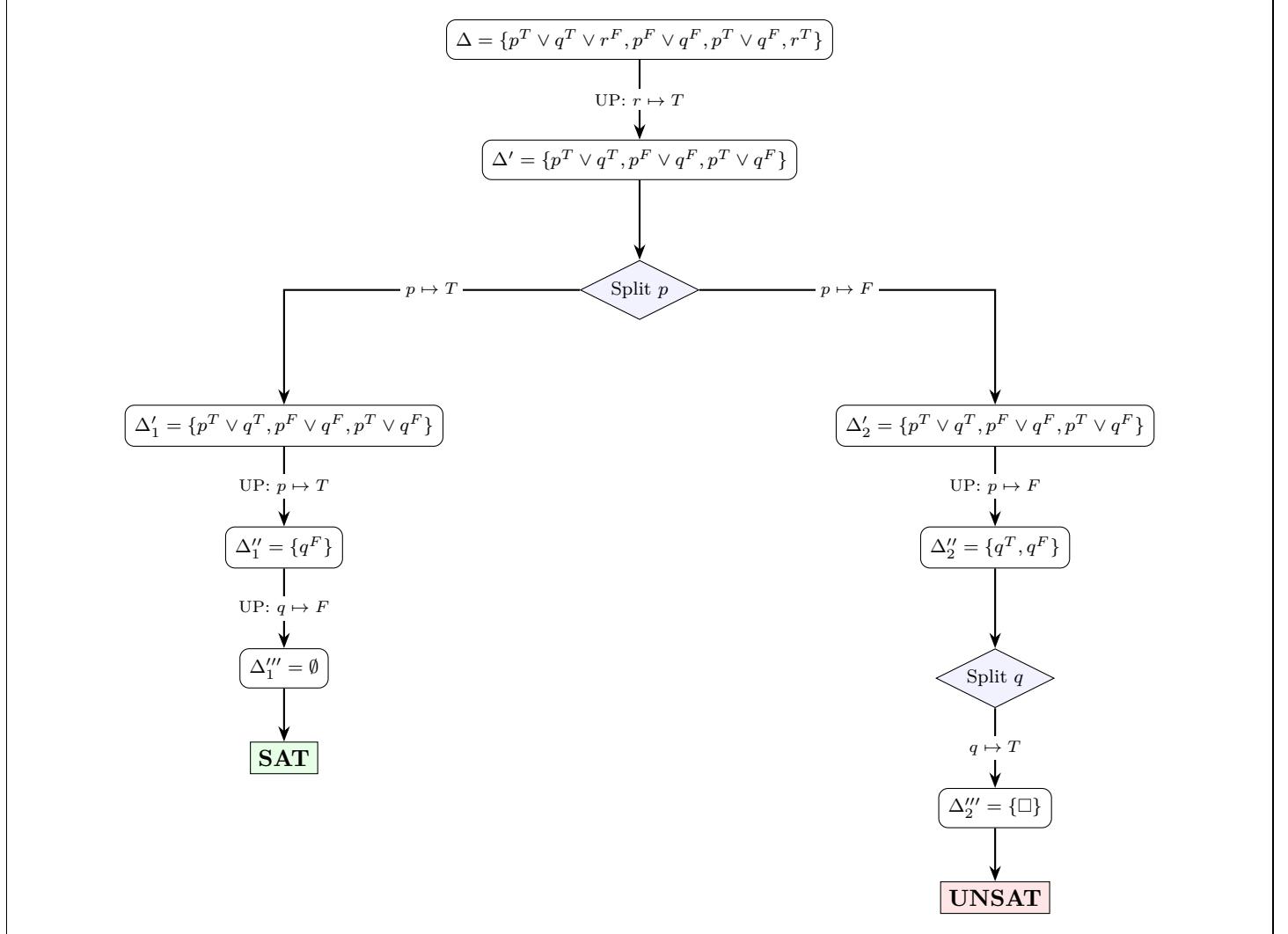
Figure 12: DPLL Search Tree

Theorem 7.6. UP is sound

Theorem 7.7. UR is refutation-sound (*since $\mathcal{R}_0$ is*)

Theorem 7.8. UR is not refutation-complete

Proof Sketch: UR makes only limited inference, as long as there are unit clauses. It does not guarantee to infer everything that can be inferred

Theorem 7.9. If DPLL returns unsatisfiable on Δ, then $\Delta \vdash_{\mathcal{R}_0} \square$ with a $\mathcal{R}_0$-refutation

Observation: we can read of a resolution proof from a DPLL trace. From a (top-down) DPLL tree, we generate a (bottom-up) resolution proof as shown in figure 13.

DPLL Search Tree (Top-Down)

$\Delta = \{p^T \vee q^T, p^T \vee q^F, p^F\}$

UP: $p \mapsto F$

$\Delta' = \{q^T, q^F\}$

UP: $q \mapsto T$

$\Delta'' = \{\square\}$

UNSAT

Resolution Proof (Bottom-Up)

$\square$

q^T

q^F

$p^T \vee q^T$

p^F

$p^T \vee q^F$

p^F
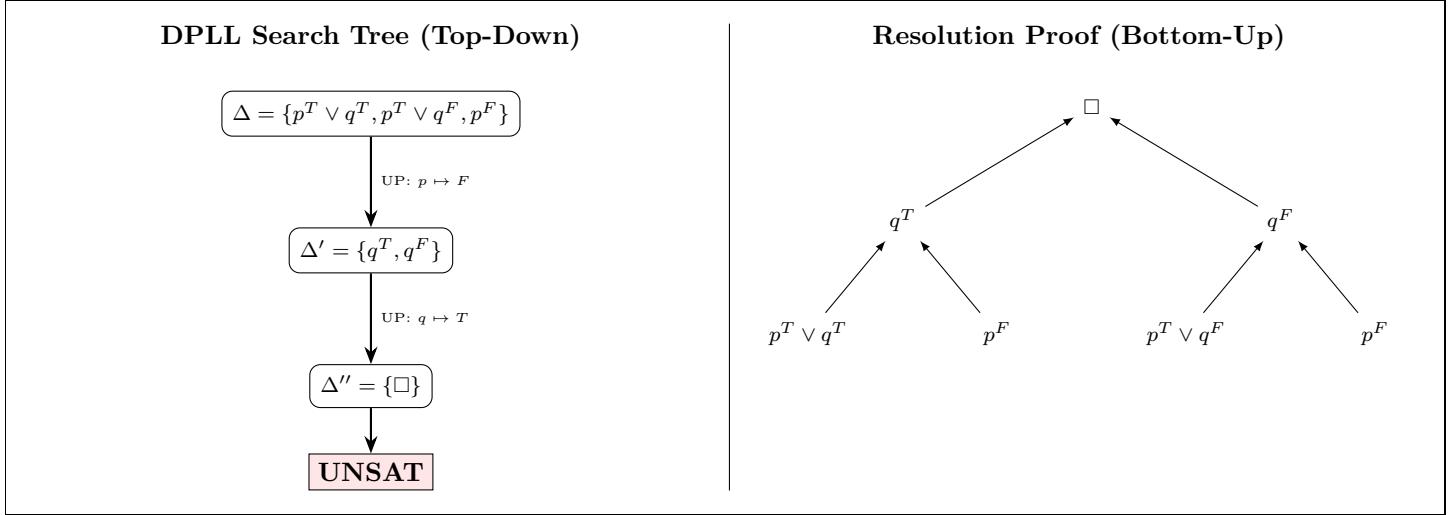
Figure 13: DPLL vs Resolution

8 First-Order Predicate Logic

Def 8.1. PL^1: First-Order (Predicate) Logic PL^1 is a formal system extensively used in mathematics, philosophy, linguistics, and computer science. It combines PL^0 with the ability to quantify over individuals

Def 8.2. First-Order Signature: A First-Order Signature is a tuple $\Sigma := \langle \Sigma_1, \Sigma_0 \rangle$ where $\Sigma_1 := \Sigma^f \cup \Sigma^p \cup \Sigma^{sk}$:

- $\Sigma^f := \bigcup_{k \in \mathbb{N}} \Sigma_k^f$ of **function constants** (or **terms**), where members of Σ_k^f denote the k-ary functions on **individuals**

- $\Sigma^p := \bigcup_{k \in \mathbb{N}} \Sigma_k^p$ of **predicate constants**, where member of Σ_k^p denote k-ary relations among individuals

- $\Sigma^{sk} := \bigcup_{k \in \mathbb{N}} \Sigma_k^{sk}$ of **Skolem constants** that act as witness constructors

- All are pairwise disjoint, countable sets of symbols for each $k \in \mathbb{N}$.

We also assume a set of **individual variables** $V_\iota := \{X, Y, Z, \cdots\}$ which we can quantify over them.

> **Note**
>
> Def 8.2 is the full FOL Signature. In PL^{nq}, we did not explicitly define Σ_0 (assumed to be there) and we did not need Σ^{sk} since no quantification over individuals was allowed. That is why in Def 6.20 we only mentioned what PL^{nq} actually uses of the full FOL Signature

> **Note**
>
> In PL^{nq}, we defined terms and formulae in Def 6.22 and Def 6.23 as follows:
> - We denote the set of all well-formed terms over a FOL Signature Σ with wff$_\iota(\Sigma)$, and the closed ones with cwff$_\iota(\Sigma)$
> - We denote the set of all well-formed formulae over a FOL Signature Σ with wff$_o(\Sigma)$, and the closed ones with cwff$_o(\Sigma)$

Def 8.3. PL^1 Terms: The set wff$_\iota(\Sigma_1, V_\iota)$ of **well-formed terms** over the signature Σ_1 and variables V_ι is defined by the following grammar:

$$
\begin{aligned}
f^k &\in& \Sigma_k^f \cup \Sigma_k^{sk} \\
X &\in& V_\iota \\
t &::=& X \mid f^k(t_1, \cdots, t_k)
\end{aligned}
$$

Def 8.4. PL^1 Propositions: The set wff$_\circ(\Sigma_1, V_\iota)$ of **well-formed formulae** over the signature Σ_1 and variables V_ι is defined by the following grammar:

$$
\begin{aligned}
p^k &\in& \Sigma_k^p \\
t_i &\in& \text{wff}_\iota(\Sigma_1, V_\iota) \\
X &\in& V_\iota \\
A &::=& p^k(t_1, \cdots, t_k) \mid \top \mid \neg A \mid A \wedge A \mid \forall X.A
\end{aligned}
$$

Def 8.5. Universal Quantifier: The **universal quantifier** $\forall$ is a binding operator used to express $\forall X.A$, i.e. a formula A holds for all possible values of an individual variable $X \in V_\iota$

> **Note**
>
> Notice that we only need $\top, \wedge, \neg$ as connectives because the others ($\bot, \vee, \Rightarrow, \Leftrightarrow$) are defined via **abbreviations**:
> - $\bot := \neg\top$
> - $A \vee B := \neg(\neg A \wedge \neg B)$
> - $A \Rightarrow B := \neg A \vee B$
> - $A \Leftrightarrow B := (A \Rightarrow B) \wedge (B \Rightarrow A)$

Def 8.6. Existential Quantifier: The **existential quantifier** $\exists$ is a binding operator defined as an **abbreviation** for a negated universal statement: $\exists X.A :\equiv \neg(\forall X.\neg A)$. It allows us to express that at least one individual satisfies the formula A.

Def 8.7. Free Variables: An occurrence of a variable X is **free** in a formula A if it is not within the scope of a binder. We define the set of free variables in A (free(A)) recursively as follows:

- **For Terms** $(t \in \text{wff}_\iota(\Sigma_1, V_\iota))$

  - free$(X) = \{X\}$

  - free$(f^k(t_1, \cdots, t_k)) := \bigcup_{i=1}^k \text{free}(t_i)$

- **For Formulae** $(A \in \text{wff}_\circ(\Sigma_1, V_\iota))$

  - free$(p^k(t_1, \cdots, t_k)) := \bigcup_{i=1}^k \text{free}(t_i)$

  - free$(\top) := \emptyset$

  - free$(\neg A) := \text{free}(A)$

  - free$(A \wedge B) := \text{free}(A) \cup \text{free}(B)$

  - free$(\forall X.A) := \text{free}(A) \setminus \{X\}$

Def 8.8. Bound Variables: An occurrence of a variable X is **bound** if it occurs within the scope of a quantifier. We define the set of bound variables $\text{BVar}(A)$ as the set of all variables that are *captured* by a binder:

- **For Terms** $(t \in \text{wff}_\iota(\Sigma_1, V_\iota))$

  - $\text{BVar}(t) := \emptyset$ (*variables in terms are always free until a formula provides a binder*)

- **For Formulae** $(A \in \text{wff}_\circ(\Sigma_1, V_\iota))$

  - $\text{BVar}(p^k(t_1, \cdots, t_k)) := \emptyset$

  - $\text{BVar}(\neg A) := \text{BVar}(A)$

  - $\text{BVar}(A \wedge B) := \text{BVar}(A) \cup \text{BVar}(B)$

  - $\text{BVar}(\forall X.A) := \text{BVar}(A) \cup \{X\}$

Def 8.9. Ground Formulae: A PL^1 formula A is called **ground/closed** iff free$(A) := \emptyset$. A closed formula is called a sentence. We denote the set of all **ground** terms with $\text{cwff}_\iota(\Sigma_1, V_\iota)$ and the set of all sentences with $\text{cwff}_\circ(\Sigma_1, V_\iota)$

Def 8.10. Alphabetical Variants: Bound variables can be **renamed**; specifically, if $\forall X.B$ is a sub-formula of A, let A' be the result of replacing $\forall X.B$ in A with $\forall Y.B'$, where B' is obtained from B by replacing all occurrences of $X \in \text{free}(B)$ with a new variable Y that does not occur in A. We call A' an **alphabetical variant** of A (and vice versa).

Def 8.11. PL^1 Universe: PL^1 uses the universe $\mathcal{D}_{\text{PL}^1} := \mathcal{D}_0 \cup \mathcal{D}_\iota$ where $\mathcal{D}_0 = \{T, F\}$ is the set of truth values and $\mathcal{D}_\iota \neq \emptyset$ is a non-empty set of individuals

Def 8.12. Interpretation in PL^1: The interpretation function in PL^1 assigns values to constants:

1. We use the interpretation from PL^0

   - $\mathcal{I}(\neg) : \mathcal{D}_0 \to \mathcal{D}_0; T \mapsto F, F \mapsto T$

   - $\mathcal{I}(\wedge) : \mathcal{D}_0 \times \mathcal{D}_0 \to \mathcal{D}_0; \langle \alpha, \beta \rangle \mapsto T$, iff $\alpha = \beta = T$

   - $\mathcal{I}(\top) = T$

   - $\mathcal{I}(\bot) = F$

2. We interpret individual constants as individuals: $\mathcal{I} : \Sigma_0^f \to \mathcal{D}_\iota$

3. We interpret function constants as functions: $\mathcal{I} : \Sigma_{k \neq 0}^f \to \mathcal{D}_\iota^{k \neq 0} \to \mathcal{D}_\iota$

4. We interpret predicate constants as relations: $\mathcal{I} : \Sigma_k^p \to \mathcal{P}(\mathcal{D}_\iota^k)$

Def 8.13. PL^1 Value Function: Given a model $\langle \mathcal{D}, \mathcal{I} \rangle$, the value function $\mathcal{I}_\varphi$ is recursively defined:

1. $\mathcal{I}_\varphi : \text{wff}_\iota(\Sigma_1, V_\iota) \to \mathcal{D}_\iota$ assigns values to terms

- $\mathcal{I}_\varphi(X) := \varphi(X)$, and

- $\mathcal{I}_\varphi(f(A_1, \cdots, A_k)) := \mathcal{I}(f)(\mathcal{I}_\varphi(A_1), \cdots, \mathcal{I}_\varphi(A_k))$

2. $\mathcal{I}_\varphi : \mathrm{wff}_\circ(\Sigma_1, V_\iota) \to \mathcal{D}_\circ$ assigns values to formulae

- $\mathcal{I}_\varphi(\top) = \mathcal{I}(\top) = T$

- $\mathcal{I}_\varphi(\neg A) := \mathcal{I}(\neg)(\mathcal{I}_\varphi(A))$

- $\mathcal{I}_\varphi(A \wedge B) := \mathcal{I}(\wedge)(\mathcal{I}_\varphi(A), \mathcal{I}_\varphi(B))$

- $\mathcal{I}_\varphi(p(A_1, \cdots, A_k)) := T$, iff $\langle \mathcal{I}_\varphi(A_1), \cdots, \mathcal{I}_\varphi(A_k) \rangle \in \mathcal{I}(p)$

- $\mathcal{I}_\varphi(\forall X.A) := T$, iff $\mathcal{I}_{\varphi,[a/X]}(A) = T$ for all $a \in \mathcal{D}_\iota$

Def 8.14. Assignment Extension: Let $\varphi : V_\iota \to \mathcal{D}_\iota$ be a **variable assignment** and $a \in \mathcal{D}_\iota$. The **extension** of φ by a for X, denoted by $\varphi, [a/X]$ is a variable assignment defined as:

$$\varphi, [a/X](Y) := \begin{cases} a & \text{if } Y = X \\ \varphi(Y) & \text{if } Y \neq X \end{cases}$$

Equivalently, viewing the assignment as a set of ordered pairs:

$$\varphi, [a/X] = \{(Y, \varphi(Y)) \mid Y \in V_\iota \setminus \{X\}\} \cup \{(X, a)\}$$

This assignment coincides with φ for all variables except X, where it maps to the individual a.

Def 8.15. Expression Language: Let signature Σ and variables $\mathcal{V}$ be disjoint sets and G a grammar whose vocabulary is $\Sigma \cup \mathcal{V}$, then we call $\langle \Sigma, \mathcal{V}, G \rangle$ an **expression language**, and denote the language of G with $\mathrm{wfe}(\Sigma, \mathcal{V})$: the set of **well-formed expressions**.

> **Note**
>
> For PL^1, we have:
> $$\mathrm{wfe}(\Sigma_1, \mathcal{V}_\iota) := \mathrm{wff}_\circ(\Sigma_1, V_\iota) \cup \mathrm{wff}_\iota(\Sigma_1, V_\iota)$$

Def 8.16. Support: Let $f : A \to B$ be a function. The **support** of f, denoted by $\mathrm{supp}(f)$, is the set of all elements in the domain A for which the function does not return a "default" or "identity" value.

Def 8.17. Substitution: Let $\mathrm{wfe}(\Sigma, \mathcal{V})$ be an expression language, then we call $\sigma : \mathcal{V} \to \mathrm{wfe}(\Sigma, \mathcal{V})$ a **substitution** iff the support $\mathrm{supp}(\sigma) := \{X \mid (X, A) \in \sigma \wedge X \neq A\}$ of σ is finite. We denote the **empty substitution** with ϵ.

Def 8.18. : We can **discharge** a variable X from a substitution σ by setting $\sigma_{-X} := \sigma, [X/X]$

Def 8.19. : Let σ be a substitution, then we call $\mathrm{intro}(\sigma) := \bigcup_{X \in \mathrm{supp}(\sigma)} \mathrm{free}(\sigma(X))$ the set of variables **introduced** by σ.

Example: consider the formula $A := \forall X.p(X, Y)$ and the substitution $\sigma := \{[Z/Y]\}$:

- $\mathrm{supp}(\sigma) = \{Y\}$

- $\mathrm{free}(\sigma(Y)) = \mathrm{free}(Z) = \{Z\}$

- $\mathrm{intro}(\sigma) = \mathrm{free}(\sigma(Y)) = \{Z\}$

Def 8.20. Substitution Application: We define substitution application by:

- $\sigma(c) = c$ for $c \in \Sigma$

- $\sigma(X) = A$ iff $X \in \mathcal{V}$ and $(X, A) \in \sigma$

- $\sigma(f(A_1, \cdots, A_n)) = f(\sigma(A_1), \cdots, \sigma(A_n))$

- $\sigma(\forall X.A) = \forall X.\sigma_{-X}(A)$

> **Note**
>
> The concept of support is useful to handle finiteness in systems that are infinite. For example, in PL^1, the set of variables $\mathcal{V}_\iota$ is infinite. However, when we apply a substitution, we don't want to deal with an infinite number of changes. By requiring the support of a substitution to be finite, we ensure that only a managable, countable number of variables are being modified, while the infinite rest of them simply stay the same (mapping to their *"identity"* value).

Substitution Example: consider the formula $g(x, y, h(z))$, and we want to apply the substitution $\sigma := \{x \mapsto a, y \mapsto f(b), z \mapsto a\}$ we get the formula $g(a, f(b), h(a))$

> **Note**
>
> Usually we use another notation for writing the substitution $\sigma := \{x \mapsto a, y \mapsto f(b), z \mapsto a\}$ which is $[a/x], [f(b)/y], [a/z]$ where $[A/X]$ (read as A *for* X i.e. substitute X with A) $\equiv X \mapsto A \equiv (X, A) \in \sigma$

Def 8.21. Substitution Extension: Let σ be a substitution, then we denote the **extension** of σ with $[A/X]$ by $\sigma, [A/X]$ and define it as $\{(Y, B) \in \sigma \mid Y \neq X\} \cup \{(X, A)\}$.

Def 8.22. Capture-Avoiding Substitution Application: Let σ be a substitution, A a formula, and A' an alphabetical variant of A, such that $\mathrm{intro}(\sigma) \cap \mathrm{BVar}(A') = \emptyset$. Then we define **capture-avoiding substitution application** via $\sigma(A) := \sigma(A')$

> **Note**
>
> We said $\sigma(\forall X.A) = \forall X.\sigma_{-X}(A)$ but this can lead to **variable capture**, i.e. $[f(X)/Y](\forall X.p(X, Y))$ would evaluate to $\forall X.p(X, f(X))$ where the second occurrence of X is now bound whereas it was free before.
> The solution for this is to rename the bound variable X in $\forall X.p(X, Y)$ before applying the substitution and there where we need capture-avoiding substitution application

Example: we want to apply $\sigma := \{[f(X)/Y]\}$ to the formula $A = \forall X.p(X, Y)$

- $\mathrm{intro}(\sigma) \cap \mathrm{BVar}(A) = \{X\} \cap \{X\} = \{X\}$
- Hence, we need to find an alphabetical variant A' for A, say $A' = \forall Z.p(Z, Y)$
- Now $\mathrm{intro}(\sigma) \cap \mathrm{BVar}(A') = \{X\} \cap \{Z\} = \emptyset$
- Hence, we define $\sigma(A) = \sigma(A')$
- $\sigma(A') = \sigma(\forall Z.p(Z, Y)) = \forall Z.\sigma_{-Z}(p(Z, Y))$
- $[f(X)/Y](p(Z, Y)) = p(Z, f(X))$
- $\sigma(A) = \sigma(A') = \forall Z.p(Z, f(X))$

Lemma 8.1. Let A and B be terms, then $\mathcal{I}_\varphi([B/X](A)) = \mathcal{I}_\psi(A)$, where $\psi = \varphi, [\mathcal{I}_\varphi(B)/X]$.

Proof. By induction on the depth of A:

1. depth $= 0$

   Then A is either a variable or a constant. If A is variable, say Y, we have two cases $Y = X$ and $Y \neq X$:

   (a) $A = Y = X$:

   then $\mathcal{I}_\varphi([B/X](A)) = \mathcal{I}_\varphi([B/X](X)) = \mathcal{I}_\varphi(B) = \psi(X) = \mathcal{I}_\psi(X) = \mathcal{I}_\psi(A)$

   (b) $A = Y \neq X$:

   then $\mathcal{I}_\varphi([B/X](A)) = \mathcal{I}_\varphi([B/X](Y)) = \mathcal{I}_\varphi(Y) = \varphi(Y) = \psi(Y) = \mathcal{I}_\psi(Y) = \mathcal{I}_\psi(A)$

   (c) A is a constant $\rightsquigarrow$ analogous to 1.b

2. depth > 0

   then $A = f(A_1, \cdots, A_n)$ and we have

   $$
   \begin{aligned}
   \mathcal{I}_\varphi([B/X](A)) &= \mathcal{I}(f)(\mathcal{I}_\varphi([B/X](A_1)), \cdots, \mathcal{I}_\varphi([B/X](A_n))) \\
   &= \mathcal{I}(f)(\mathcal{I}_\psi(A_1), \cdots, \mathcal{I}_\psi(A_n)) \\
   &= \mathcal{I}_\psi(A)
   \end{aligned}
   $$

   by induction hypothesis

   $\square$

Lemma 8.2. Let A and B be propositions, then $\mathcal{I}_\varphi([B/X]A) = \mathcal{I}_\psi(A)$, where $\psi = \varphi, [\mathcal{I}_\varphi(B)/X]$.

Proof. By induction on the number of n connectives and quantifiers in A:

1. $n = 0$

   then A is an atomic formula, and we can argue like in the induction step of the Lemma 8.1

2. $n > 0$ and $A = \neg B$ or $A = C \circ D$

   Here we argue like in the induction step of Lemma 8.1 as well

3. $n > 0$ and $A = \forall Y.C$ where (WLOG) $X \neq Y$ (otherwise, rename)

   (a) then $\mathcal{I}_\psi(A) = \mathcal{I}_\psi(\forall Y.C) = T$, iff $\mathcal{I}_{\psi,[a/Y]}(C) = T$ for all $a \in \mathcal{D}_\iota$

   (b) But $\mathcal{I}_{\psi,[a/Y]}(C) = \mathcal{I}_{\varphi,[a/Y]}([B/X](C)) = T$, by induction hypothesis.

   (c) So $\mathcal{I}_\psi(A) = \mathcal{I}_\varphi(\forall Y.[B/X](C)) = \mathcal{I}_\varphi([B/X](\forall Y.C)) = \mathcal{I}_\varphi([B/X](A))$

$\square$

Def 8.23. $\mathcal{ND}_1$: The First-Order Natural Deduction Calculus extends $\mathcal{ND}_0$ by the following four inference rule:

$$\frac{A}{\forall X.A} \; \forall I^* \qquad\qquad\qquad \frac{\forall X.A}{[B/X](A)} \; \forall E$$

* means that A does not depend on any hypothesis in which X is free

$$\frac{[B/X](A)}{\exists X.A} \; \exists I \qquad\qquad \frac{\exists X.A \qquad \overset{[[c/X](A)]^1}{\underset{\vdots}{C}} \qquad c \in \sum_0^{sk} \text{ new}}{C} \; \exists E^1$$

> **Note**
>
> If our knowledge base has loves(Anna, Anna), using $\exists I$ we can infer several things, e.g.:
> - $\exists X.\text{loves}(X, X)$
> - $\exists X.\text{loves}(X, \text{Anna})$
> - $\exists X.\text{loves}(\text{Anna}, X)$
> - $\exists X \exists Y.\text{loves}(X, Y)$
>
> Hence, you might see the rule written in the following notation:
>
> $$\frac{A(\cdots c \cdots c \cdots)}{\exists X.A(\cdots c \cdots X \cdots)} \; \exists I$$
>
> to emphasize that we do not need to replace all instances of the terms c with the variable X (but we can!)

Notes on $\forall I$

If we are trying to introduce something of the form $\forall X.A$, conceptually, this is a tricky thing to do! Because how could we prove that A holds for everything! It is not practical to go through every individuals! Therefore, we need a clever way of doing that.

Let's compare between two types of reasoning that involve generalization, first, let's see the following one:

Step	Formula	Rule Applied
(1)	Blonde(Anna)	Assumption
(2)	Blonde(Anna) $\vee$ Brown(Anna)	$\vee I_l$ (on 1)
(3)	Blonde(Anna) $\Rightarrow$ (Blonde(Anna) $\vee$ Brown(Anna))	$\Rightarrow I^1$ (on 1, 2) $\rightsquigarrow$ Discharge (1)
(4)	$\forall X.(\text{Blonde}(X) \Rightarrow (\text{Blonde}(X) \vee \text{Brown}(X)))$	$\forall I$ (on 3)

That was a **valid** reasoning. Now let's see another one that looks *(similar)*:

Step	Formula	Rule Applied
(1)	Blonde(Anna)	Assumption
(2)	Tall(Anna)	Assumption
(3)	Blonde(Anna) $\wedge$ Tall(Anna)	$\wedge I$ (on 1,2)
(4)	Tall(Anna) $\Rightarrow$ (Blonde(Anna) $\wedge$ Tall(Anna))	$\Rightarrow I^2$ (on 2,3) $\leadsto$ Discharge (2)
(5)	$\forall X.(\text{Tall}(X) \Rightarrow (\text{Blonde}(X) \wedge \text{Tall}(X)))$	$\forall I$ (on 4)

That was invalid reasoning in an obvious way! Just because someone is tall, that doesn't mean they are both tall and blonde!

The difference is in the way the individuals "Anna" was used. In the incorrect proof, we assumed Anna is blonde, then we assumed something further about her (being tall), under that new assumption we inferred she's both tall and blonde and then we inferred something further to conclude everyone is like that!

In the correct one, however, we derived the formula from one assumption about Anna and Anna did not appear anywhere before that. Anna was just an *arbitrary* name.

Therefore, the rule might also be written in a different notation:

$$\frac{A(\cdots c \cdots c \cdots)}{\forall X.A(\cdots X \cdots X \cdots)} \ \forall I$$

where c is arbitrary and $X \notin A$

Def 8.24. $\mathcal{ND}_\vdash^1$**:** The inference rules of the **First-Order Sequent Style Natural Deduction Calculus** consist of those from $\mathcal{ND}_\vdash^0$ plus the following quantifiers rules:

$$\frac{\Gamma \vdash A \quad X \notin \text{free}(\Gamma)}{\Gamma \vdash \forall X.A} \ \forall I \qquad\qquad \frac{\Gamma \vdash \forall X.A}{\Gamma \vdash [B/X](A)} \ \forall E$$

$$\frac{\Gamma \vdash [B/X](A)}{\Gamma \vdash \exists X.A} \ \exists I \qquad\qquad \frac{\Gamma \vdash \exists X.A \quad \Gamma, [c/X]A \vdash C \quad c \in \Sigma_0^{\text{sk}}\text{new}}{\Gamma \vdash C} \ \exists E^1$$
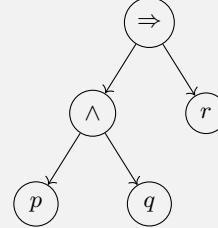
Def 8.25. $PL^1_=$: We extend PL^1 with a new logical constant for **equality** $=\in \Sigma^p_2$ and fix its interpretation to $\mathcal{I}(=) := \{(x,x) \mid x \in \mathcal{D}_\iota\}$. We call the extended logic **First-Order Logic with Equality**.

Def 8.26. Position: A position is a finite sequence of natural numbers $p = \langle i_1, i_2, \ldots, i_n \rangle$ that describes a path from the root of a well-formed expression to a specific sub-node. For an expression A, the subexpression at position p is denoted by $A|_p$. We write $C[A]_p$ to indicate that the expression C contains the subexpression A at position p. If A is an atomic formula, we call p an **occurrence** of A in C.

Example: consider $C := (p \wedge q) \Rightarrow r$
- Position $\langle 1 \rangle$ is the subexpression $(p \wedge q)$
- Position $\langle 1, 1 \rangle$ is the occurrence of the atom p
- Position $\langle 2 \rangle$ is the occurrence of the atom r

Def 8.27. Replacement: Let p be a position, then $[A/p]C$ is the expression obtained from C by **replacing** the subexpression at p by A

Def 8.28. $\mathcal{ND}^1_=$: **The First-Order Logic with Equality Natural Deduction Calculus** extends $\mathcal{ND}_1$ by the following two inference rule:

$$\frac{}{A = A} = I \qquad\qquad\qquad \frac{A = B \quad C[A]_p}{[B/p]C} = E$$

Note

equivalence $(\Leftrightarrow)$ behaves exactly as equality $(=)$

Def 8.29. : $\Leftrightarrow I$ is derivable and $\Leftrightarrow E$ is admissible in $\mathcal{ND}_1$:

$$\frac{}{A \Leftrightarrow A} \Leftrightarrow I \qquad\qquad\qquad \frac{A \Leftrightarrow B \quad C[A]_p}{[B/p]C} \Leftrightarrow E$$

Figure 14 shows all inference rules of the natural deduction calculus, i.e. the union of $\mathcal{ND}_0$, $\mathcal{ND}_1$, $\mathcal{ND}^1_=$, and equivalence rules

$$\frac{A \quad B}{A \wedge B} \wedge I \qquad\qquad \frac{A \wedge B}{A} \wedge E_l \qquad\qquad \frac{A \wedge B}{B} \wedge E_r$$

$$\begin{array}{c} [A]^1 \\ \vdots \\ \dfrac{B}{A \Rightarrow B} \Rightarrow I^1 \end{array} \qquad\qquad \frac{A \Rightarrow B \quad A}{B} \Rightarrow E$$

$$\frac{}{A \vee \neg A} \mathcal{TND}$$

$$\frac{}{A \Leftrightarrow A} \Leftrightarrow I \qquad\qquad \frac{A \Leftrightarrow B \quad C[A]_p}{[B/p]C} \Leftrightarrow E$$

$$\frac{A}{A \vee B} \vee I_l \qquad\qquad \frac{B}{A \vee B} \vee I_r$$

$$\begin{array}{c} [A]^1 \quad [B]^1 \\ \vdots \qquad \vdots \\ \dfrac{A \vee B \quad C \quad C}{C} \vee E^1 \end{array}$$

$$\begin{array}{c} [A]^1 \quad [A]^1 \\ \vdots \qquad \vdots \\ \dfrac{C \quad \neg C}{\neg A} \neg I^1 \end{array} \qquad\qquad \frac{\neg\neg A}{A} \neg E$$

$$\frac{\neg A \quad A}{\bot} \bot I \qquad\qquad \frac{\bot}{A} \bot E$$

$$\frac{A}{\forall X.A} \forall I^* \qquad\qquad \frac{\forall X.A}{[B/X](A)} \forall E$$

* means that A does not depend on any hypothesis in which X is free

$$\frac{[B/X](A)}{\exists X.A} \exists I \qquad\qquad \begin{array}{c} [[c/X](A)]^1 \\ \vdots \\ \exists X.A \quad \dfrac{ C c \in \sum_0^{sk} \text{ new}}{C} \exists E^1 \end{array}$$

$$\frac{}{A = A} = I \qquad\qquad \frac{A = B \quad C[A]_p}{[B/p]C} = E$$

Figure 14: All Natural Deduction Rules

Example Proofs

Figure 15 shows a linearized $\mathcal{ND}_1$ proof of the formula:

$$((\exists X.f(X)) \Rightarrow g(a)) \Rightarrow (\forall X.(f(X) \Rightarrow g(a)))$$

Step	Formula	Rule Applied
(1)	$(\exists X.f(X)) \Rightarrow g(a)$	Assumption
(2)	$f(b)$	Assumption
(3)	$\exists X.f(X)$	$\exists I$ (on 2)
(4)	$g(a)$	$\Rightarrow E$ (on 1,3)
(5)	$f(b) \Rightarrow g(a)$	$\Rightarrow I^2$ (on 2, 4) $\rightsquigarrow$ Discharge (2)
(6)	$\forall X.(f(X) \Rightarrow g(a))$	$\forall I$ (on 5)
(7)	$((\exists X.f(X)) \Rightarrow g(a)) \Rightarrow (\forall X.(f(X) \Rightarrow g(a)))$	$\Rightarrow I^1$ (on 1, 6) $\rightsquigarrow$ Discharge (1)

Figure 15: $\mathcal{ND}_1$ Linearized Style Example Proof

The same formula proof in Gentzen-Prawitz Style is shown in figure 16:

$$
\cfrac{
  \cfrac{
    \cfrac{
      [(\exists X.f(X)) \Rightarrow g(a)]^1 \quad \cfrac{\cfrac{[f(b)]^2}{\exists X.f(X)}\ \exists I}{}
    }{g(a)}\ \Rightarrow E
    \bigg/ \cfrac{}{f(b) \Rightarrow g(a)}\ \Rightarrow I^2
  }{\forall X.(f(X) \Rightarrow g(a))}\ \forall I
}{((\exists X.f(X)) \Rightarrow g(a)) \Rightarrow (\forall X.(f(X) \Rightarrow g(a)))}\ \Rightarrow I^1
$$

Figure 16: $\mathcal{ND}_1$ Gentzen-Prawitz Style Example Proof

Figure 17 shows an example Sequent-Style proof for the validity of the sequent:

$$(\exists X.(f(X) \wedge g(X))) \vdash (\exists X.f(X) \wedge \exists X.g(X))$$

Let $\Phi = \exists X.(f(X) \wedge g(X))$.

$$
\cfrac{
  \cfrac{}{\Phi \vdash \exists X.(f(X) \wedge g(X))}\ \text{Ax}
  \qquad
  \cfrac{
    \cfrac{
      \cfrac{\cfrac{}{\Phi, f(a) \wedge g(a) \vdash f(a) \wedge g(a)}\ \text{Ax}}{\Phi, f(a) \wedge g(a) \vdash f(a)}\ \wedge E_l
      \bigg/ \Phi, f(a) \wedge g(a) \vdash \exists X.f(X)\ \exists I
      \qquad
      \cfrac{\cfrac{}{\Phi, f(a) \wedge g(a) \vdash f(a) \wedge g(a)}\ \text{Ax}}{\Phi, f(a) \wedge g(a) \vdash g(a)}\ \wedge E_r
    }{\Phi, f(a) \wedge g(a) \vdash \exists X.f(X) \wedge \exists X.g(X)}\ \wedge I \qquad \substack{a \in \Sigma_0^{sk} \\ \text{new}}
  }{}
}{\Phi \vdash \exists X.f(X) \wedge \exists X.g(X)}\ \exists E
$$

Figure 17: $\mathcal{ND}_\vdash^1$ Example Proof

Excursion

Def 8.30. Integer Division (Euclidean Division): For any integers a (the **dividend**) and b (the **divisor**) where $b \neq 0$, there exist unique integers q (the **quotient**) and r (the **remainder**) such that: $a = b \cdot q + r$ and $0 \leq r < |b|$

Def 8.31. Quotient: The result q obtained from integer division. It represents how many full times the divisor b fits into the dividend a.

Def 8.32. Remainder: The amount r "left over" after integer division. If $r = 0$, then $b \mid a$ (the divisor divides the dividend).

Def 8.33. Set of Divisors: For a given integer $n \in \mathbb{Z}$, the **set of divisors** is the set of all integers that stand in the divides relation to n. Formally: $\mathrm{Div}(n) := \{d \in \mathbb{Z} \mid d \mid n\}$

Def 8.34. Prime Number: A natural number $p \in \mathbb{N}$ is **prime** iff $p > 1$ and its only positive divisors are 1 and itself. Formally: $p \in \mathbb{P} :\equiv p > 1 \wedge \forall d \in \mathbb{N}.((d \mid p) \Rightarrow d = 1 \vee d = p)$

Def 8.35. Composite Number: A natural number $n \in \mathbb{N}$ is **composite** iff there exists at least one divisor other than 1 and itself. Formally: n is composite $:\equiv n > 1 \wedge \exists d \in \mathbb{N}.(1 < d < n \wedge d \mid n)$

Def 8.36. Prime Factor: An integer p is a **prime factor** of $n \in \mathbb{Z}$ iff $p \in \mathbb{P}$ and $p \in \mathrm{Div}(n)$ (i.e., p is a prime number that stands in the divides relation to n).

Def 8.37. Prime Factor Multiset: For any $n \in \mathbb{N}, n > 1$, the **prime factor multiset** $\mathcal{M}(n)$ over the domain $\mathbb{P}$ is the multiset where the multiplicity $m(p)$ of each prime p is its exponent in the prime factorization. The product of all elements in the multiset equals n: $\prod_{p \in \mathbb{P}} p^{m(p)} = n$.

Def 8.38. Prime Factorization: The representation of a natural number $n > 1$ as its prime factor multiset. By the Fundamental Theorem of Arithmetic, this multiset is unique.

> ### Note
>
> **Example: Prime Factorization of 12** To find the prime factor multiset, we use **Trial Division**. We repeatedly divide by the smallest possible prime number ($p \in \{2, 3, 5, \dots\}$) until the quotient reaches 1.
> 1. **Trial with $p = 2$:** $12 \div 2 = 6$. Record $m(2) = 1$. Target is 6.
> 2. **Trial with $p = 2$ again:** $6 \div 2 = 3$. Update $m(2) = 2$. Target is 3.
> 3. **Trial with $p = 2$:** $3 \div 2 = 1$ with remainder 1. Since $2 \nmid 3$, we move to the next prime, $p = 3$.
> 4. **Trial with $p = 3$:** $3 \div 3 = 1$. Record $m(3) = 1$. Target is 1. Terminate.
>
> **Final Multiset:** The multiset is $\mathcal{M}(12) = \{2, 2, 3\}$, where $m(2) = 2$ and $m(3) = 1$. **Final Notation:** $12 = 2^2 \cdot 3$.

Def 8.39. Coprime: Two integers $a, b \in \mathbb{Z}$ are **coprime** iff the intersection of their prime factor multisets is empty. Formally: $\mathcal{M}(a) \cap \mathcal{M}(b) = \emptyset$, where $\cap$ for multisets is defined by $m_\cap(x) = \min(m_a(x), m_b(x))$.

Def 8.40. Greatest Common Divisor (GCD): For two natural numbers $a, b \in \mathbb{N}$, the **Greatest Common Divisor**, denoted $\gcd(a, b)$, is the largest positive integer that divides both a and b. If $\gcd(a, b) = 1$, then a and b are coprime.

Method 1: Prime Factorization This method identifies the shared "building blocks" of two numbers. The GCD is the product of the elements in the multiset intersection of their prime factor multisets.

Example: Find gcd$(24, 36)$

1. Find prime factor multisets: $\mathcal{M}(24) = \{2, 2, 2, 3\}$ and $\mathcal{M}(36) = \{2, 2, 3, 3\}$.

2. Find the multiset intersection: $\mathcal{M}(24) \cap \mathcal{M}(36) = \{2, 2, 3\}$ (using the minimum multiplicity for each prime).

3. Multiply the elements: $2 \times 2 \times 3 = 12$.

4. Result: $\gcd(24, 36) = 12$.

Method 2: Successive Division This method uses the remainder from integer division. We repeatedly replace the larger number with the remainder until the remainder is 0. The last non-zero remainder is the GCD.

Example: Find gcd$(48, 18)$

1. $48 \div 18$: quotient 2, remainder 12. $\hfill (48 = 18 \cdot 2 + 12)$

2. $18 \div 12$: quotient 1, remainder 6. $\hfill (18 = 12 \cdot 1 + 6)$

3. $12 \div 6$: quotient 2, remainder 0. $\hfill (12 = 6 \cdot 2 + 0)$

The last non-zero remainder is **6**, so $\gcd(48, 18) = 6$.

Def 8.41. Even Integer: An integer $n \in \mathbb{Z}$ is **even** iff there exists an integer $k \in \mathbb{Z}$ such that $n = 2k$.

Def 8.42. Odd Integer: An integer $n \in \mathbb{Z}$ is **odd** iff there exists an integer $k \in \mathbb{Z}$ such that $n = 2k + 1$.

Lemma 8.3. $\forall n, m \in \mathbb{Z}, \neg(2n + 1 = 2m)$ (an odd number can never equal an even number)

Lemma 8.4. $\forall n, m \in \mathbb{N} \setminus \{1\}. \mathcal{M}(nm) \equiv \mathcal{M}(mn)$ (The prime factor multiset of nm is identical to that of mn, as $\mathcal{M}(n) \uplus \mathcal{M}(m) \equiv \mathcal{M}(m) \uplus \mathcal{M}(n)$.)

Lemma 8.5. $\forall n \in \mathbb{N} \setminus \{1\}, \forall p \in \mathbb{P}. |\mathcal{M}(n \cdot p)| = |\mathcal{M}(n)| + 1$ (Multiplying a number by a prime p increases the total count of prime factors by exactly one, as $\mathcal{M}(np) \equiv \mathcal{M}(n) \uplus \{p\}$.)

Lemma 8.6. $\forall n \in \mathbb{N} \setminus \{1\}, \forall k \in \mathbb{N}. |\mathcal{M}(n^k)| = k \cdot |\mathcal{M}(n)|$ (The total count of prime factors of n^k is k times the count of prime factors of n. Specifically, for $k = 2$, the count **doubles**: $|\mathcal{M}(n^2)| = 2 \cdot |\mathcal{M}(n)|$.)

Def 8.43. Rational Number: A real number $x \in \mathbb{R}$ is **rational** iff it can be expressed as a ratio of two integers. Formally: $x \in \mathbb{Q} :\equiv \exists p, q \in \mathbb{Z}. (q \neq 0 \wedge x = \frac{p}{q})$

Def 8.44. Irrational Number: A real number $x \in \mathbb{R}$ is **irrational** iff it is not rational. Formally: $x \in \mathbb{I} :\equiv x \in \mathbb{R} \setminus \mathbb{Q}$

Theorem 8.7. $\sqrt{2} \notin \mathbb{Q}$ (irrational).

Proof. Suppose for the sake of contradiction that $\sqrt{2}$ is rational.

1. Then there exist $p, q \in \mathbb{N}$ such that $\sqrt{2} = \frac{p}{q}$.

2. Squaring both sides gives $2 = \frac{p^2}{q^2}$, which implies:
$$2q^2 = p^2$$

3. Let $|\mathcal{M}(n)|$ be the total count of prime factors of n. We analyze the parity of the counts on both sides of the equation:
   - **Right Side (p^2):** By the Scaling Lemma (lemma 8.6), $|\mathcal{M}(p^2)| = 2 \cdot |\mathcal{M}(p)|$. This count is even.
   - **Left Side ($2q^2$):** By the Increment Lemma (lemma 8.5), $|\mathcal{M}(2 \cdot q^2)| = |\mathcal{M}(q^2)| + 1$.
   - Since $|\mathcal{M}(q^2)|$ is $2 \cdot |\mathcal{M}(q)|$ (even), then $|\mathcal{M}(q^2)| + 1$ is odd.

4. This implies an even number of prime factors equals an odd number of prime factors.

5. By the Parity Non-Equivalence Lemma (lemma 8.3), $\forall n, m. \neg(2n = 2m + 1)$.

This is a contradiction. Therefore, the assumption that $\sqrt{2} \in \mathbb{Q}$ must be false. Thus, $\sqrt{2}$ is irrational. $\qquad\square$

We can do real mathematics with $\mathcal{ND}^1_=$, figure 18 shows an $\mathcal{ND}^1_=$ proof by contradiction that $\sqrt{2}$ is irrational:

Step	Formula	Rule Applied
(1)	$\forall n, m \in \mathbb{Z}, \neg(2n + 1 = 2m)$	Lemma 8.3
(2)	$\forall n \in \mathbb{N} \setminus \{1\}, \forall k \in \mathbb{N}. \left|\mathcal{M}(n^k)\right| = k \cdot |\mathcal{M}(n)|$	Lemma 8.6
(3)	$\forall n \in \mathbb{N} \setminus \{1\}, \forall p \in \mathbb{P}. |\mathcal{M}(n \cdot p)| = |\mathcal{M}(n)| + 1$	Lemma 8.5
(4)	$\forall x.\mathrm{irr}(x) \Leftrightarrow \neg(\exists p, q.x = \frac{p}{q})$	Def 8.44
(5)	$\mathrm{irr}(\sqrt{2}) \Leftrightarrow \neg(\exists p, q.\sqrt{2} = \frac{p}{q})$	$\forall E$ (on 4)
(6)	$\neg\mathrm{irr}(\sqrt{2})$	Assumption
(7)	$\neg\neg(\exists p, q.\sqrt{2} = \frac{p}{q})$	$\Leftrightarrow E$ (on 5,6)
(8)	$\exists p, q.\sqrt{2} = \frac{p}{q}$	$\neg E$ (on 7)
(9)	$\sqrt{2} = \frac{p}{q}$	Ax (follows from 6)
(10)	$2 = \frac{p^2}{q^2} \rightsquigarrow 2q^2 = p^2$	Arithmetic (on 9)
(11)	$\left|\mathcal{M}(p^2)\right| = 2 \cdot |\mathcal{M}(p)|$	$\forall E$ (on 2)
(12)	$q^2 \in \mathbb{N} \wedge 2 \in \mathbb{P} \Rightarrow \left|\mathcal{M}(q^2 \cdot 2)\right| = \left|\mathcal{M}(q^2)\right| + 1$	$\forall E$ (on 3)
(13)	$q^2 \in \mathbb{N} \wedge 2 \in \mathbb{P}$	Ax, Def 8.34
(14)	$\left|\mathcal{M}(2q^2)\right| = \left|\mathcal{M}(q^2)\right| + 1$	$\Rightarrow E$ (on 12, 13)
(15)	$\left|\mathcal{M}(q^2)\right| = 2 \cdot |\mathcal{M}(q)|$	$\forall E$ (on 2)
(16)	$\left|\mathcal{M}(2q^2)\right| = 2 \cdot |\mathcal{M}(q)| + 1$	$= E$ (on 14, 15)
(17)	$\left|\mathcal{M}(p^2)\right| = \left|\mathcal{M}(p^2)\right|$	$= I$
(18)	$\left|\mathcal{M}(2q^2)\right| = \left|\mathcal{M}(p^2)\right|$	$= E$ (on 10,17)
(19)	$2 \cdot |\mathcal{M}(q)| + 1 = \left|\mathcal{M}(p^2)\right|$	$= E$ (on 16,18)
(20)	$2 \cdot |\mathcal{M}(q)| + 1 = 2 \cdot |\mathcal{M}(p)|$	$= E$ (on 11,19)
(21)	$\neg(2\,|\mathcal{M}(q)| + 1 = 2\,|\mathcal{M}(p)|)$	$\forall E$ (on 1)
(22)	$\bot$	$\bot I$ (on 20,21)
(23)	$\bot$	$\exists E^6$ (on 22)
(24)	$\neg\neg\mathrm{irr}(\sqrt{2})$	$\neg I$ (on 23)
(25)	$\mathrm{irr}(\sqrt{2})$	$\neg E$ (on 24)

Figure 18: $\mathcal{ND}_=^1$ Proof of $\sqrt{2} \notin \mathbb{Q}$

9 Automated Theorem Proving in First-Order Logic

Def 9.1. $\mathcal{T}_1$: The **Standard First-Order Tableau Calculus** extends $\mathcal{T}_0$ with the following inference rules

$$\frac{(\forall X.A)^T \quad C \in \mathrm{cwff}_\iota(\Sigma)}{([C/X](A))^T} \; \mathcal{T}_1 \forall \qquad\qquad \frac{(\forall X.A)^F \quad c \in \Sigma_0^{sk} \mathrm{new}}{([c/X]A)^F} \; \mathcal{T}_1 \exists$$

> **Note**
>
> **Problem:** The rule $\mathcal{T}_1 \forall$ displays a case of *"don't know indeterminism"*: to find a refutation we have to guess a formula C from the infinite set $\mathrm{cwff}_\iota(\Sigma)$.
> For proof search, this means $\mathcal{T}_1 \forall$ is infinitely branching in general

Def 9.2. $\mathcal{T}_1^f$: The **Free Variable Tableau Calculus** extends $\mathcal{T}_0$ with the following inference rules

$$\frac{(\forall X.A)^T \quad Y \mathrm{new}}{([Y/X](A))^T} \; \mathcal{T}_1^f \forall \qquad\qquad \frac{(\forall X.A)^F \quad \mathrm{free}(\forall X.A) = \left\{ X^1, \cdots, X^k \right\} \quad f \in \Sigma_k^{sk} \mathrm{new}}{([f(X^1, \cdots, X^k)/X]A)^F} \; \mathcal{T}_1^f \exists$$

and generalizes its cut rule $\mathcal{T}_0 \perp$ to:

$$
\begin{array}{c}
A^\alpha \\
\vdots \\
\dfrac{B^\beta \qquad (\alpha \neq \beta) \qquad \sigma(A) = \sigma(B)}{\perp : \sigma} \; \mathcal{T}_1^f \perp
\end{array}
$$

$\mathcal{T}_1^f$ **instantiates** the **whole tableau** by σ

Def 9.3. : Derivable rules in $\mathcal{T}_1^f$:

$$\frac{(\exists X.A)^F \quad Y \mathrm{new}}{([Y/X](A))^F} \qquad\qquad \frac{(\exists X.A)^T \quad free(\forall X.A) = \left\{ X^1, \cdots, X^k \right\} \quad f \in \Sigma_k^{sk} \mathrm{new}}{([f(X^1, \cdots, X^k)/X](A))^T}$$

Combining $\mathcal{T}_0$ and $\mathcal{T}_1^f$ we get the inference rules shown in figure 19:

$$\frac{(A \wedge B)^T}{\begin{array}{c}A^T\\B^T\end{array}} \; \mathcal{T}_0\wedge \qquad\qquad\qquad \frac{(A \wedge B)^F}{A^F \mid B^F} \; \mathcal{T}_0\vee$$

$$\frac{(\neg A)^T}{A^F} \; \mathcal{T}_0\neg^T \qquad\qquad\qquad\qquad \frac{(\neg A)^F}{A^T} \; \mathcal{T}_0\neg^F$$

$$\begin{array}{c}A^\alpha\\ \vdots\\ \underline{A^\beta \quad (\alpha \neq \beta)}\\ \bot\end{array} \; \mathcal{T}_0\bot$$

$$\frac{(\forall X.A)^T \quad Y\,\text{new}}{([Y/X](A))^T} \; \mathcal{T}_1^f\forall \qquad\qquad \frac{(\forall X.A)^F \quad \text{free}(\forall X.A) = \left\{X^1,\cdots,X^k\right\} \quad f \in \Sigma_k^{sk}\text{new}}{([f(X^1,\cdots,X^k)/X]A)^F} \; \mathcal{T}_1^f\exists$$

$$\begin{array}{c}A^\alpha\\ \vdots\\ \underline{B^\beta \quad (\alpha \neq \beta) \quad \sigma(A) = \sigma(B)}\\ \bot : \sigma\end{array} \; \mathcal{T}_1^f\bot$$

$$\frac{(A \Rightarrow B)^T}{A^F \mid B^T} \qquad\qquad \frac{(A \Rightarrow B)^F}{\begin{array}{c}A^T\\B^F\end{array}} \qquad\qquad \frac{\begin{array}{c}A^T\\(A \Rightarrow B)^T\end{array}}{B^T}$$

$$\frac{(A \vee B)^T}{A^T \mid B^T} \qquad\qquad\qquad\qquad \frac{(A \vee B)^F}{\begin{array}{c}A^F\\B^F\end{array}}$$

$$\frac{(A \iff B)^T}{\begin{array}{c}A^T\\B^T\end{array} \; \middle| \; \begin{array}{c}A^F\\B^F\end{array}} \qquad\qquad\qquad \frac{(A \iff B)^F}{\begin{array}{c}A^T\\B^F\end{array} \; \middle| \; \begin{array}{c}A^F\\B^T\end{array}}$$

$$\frac{(\exists X.A)^F \quad Y\,\text{new}}{([Y/X](A))^F} \qquad\qquad \frac{(\exists X.A)^T \quad free(\forall X.A) = \{X^1,\cdots,X^k\} \quad f \in \Sigma_k^{sk}\text{new}}{([f(X^1,\cdots,X^k)/X](A))^T}$$

Figure 19: $\mathcal{T}_0 \cup \mathcal{T}_1^f$

Example Proof

$$[(\forall X.P(X) \Rightarrow Q(X)) \Rightarrow ((\forall X.P(X)) \Rightarrow (\forall X.Q(X)))]^F$$

$$[\forall X.P(X) \Rightarrow Q(X)]^T$$

$$((\forall X.P(X)) \Rightarrow (\forall X.Q(X)))^F$$

$$[\forall X.P(X)]^T$$

$$[\forall X.Q(X)]^F$$

$$[P(Y)]^T$$

$$[Q(c)]^F$$

$$[P(Z) \Rightarrow Q(Z)]^T$$

$$[P(Z)]^F \qquad\qquad [Q(Z)]^T$$

$$\bot : [Y/Z] \qquad\qquad [Q(Y)]^T$$

$$\bot : [c/Y]$$
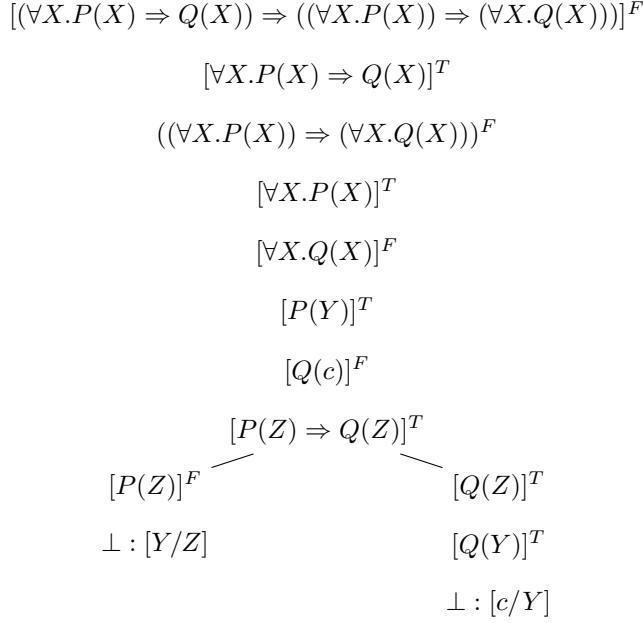
Figure 20: $\mathcal{T}_1^f$ Proof Example

All $\mathcal{T}_1^f$ rules need to be applied once except for $\mathcal{T}_1^f \forall$. The number of applications of $\mathcal{T}_1^f \forall$ to $\forall X.A$ is called it's **multiplicity**.

Theorem 9.1. $\mathcal{T}_1^f$ is only complete with unbounded multiplicities

> **Note**
>
> That is why validity in PL^1 is undecidable

For example, see the proof in figure 21:

$$[(P(a) \vee P(b)) \Rightarrow \exists X.P(X)]^F$$

$$[P(a) \vee P(b)]^T$$

$$[\exists X.P(X)]^F$$

$$[P(Y)]^F \xrightarrow{[a/Y]} [P(a)]^F$$

$$[P(a)]^T \qquad\qquad [P(b)]^T \qquad \text{2nd application of } \mathcal{T}_1^f \forall$$

$$\bot : [a/Y]$$

$$[P(Z)]^F \xrightarrow{[b/Z]} [P(b)]^F$$

$$\bot : [b/Z]$$
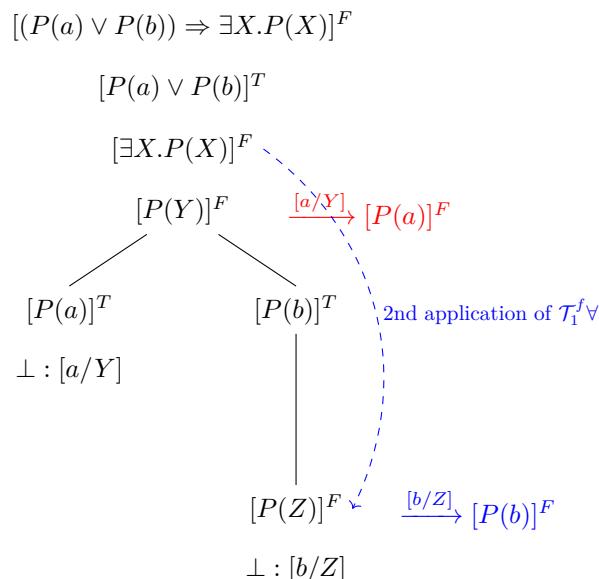
Figure 21: $\mathcal{T}_1^f \forall$ Multiplicity Example

Def 9.4. Unification: For given terms A_1, A_2, **unification** is the problem of finding a substitution σ (called **unifier**), such that $\sigma(A_1) = \sigma(A_2)$

Def 9.5. Unification Problem: We call a formula $A^1 =^? B^1 \wedge \cdots \wedge A^n =^? B^n$ a **unification problem** iff $A^i, B^i \in \mathrm{wff}_\iota(\Sigma_1, V_\iota)$

> **Note**
>
> We consider unification problems as sets of equations. $\wedge$ here is associative, commutative, and idempotent. We consider equations as two-element multisets, $=^?$ here is commutative

Def 9.6. : A substitution is called a **unifier** for a unification problem $\mathcal{E}$ (and thus $\mathcal{E}$ is **unifiable**), iff it is a (simultaneous) unifier for all pairs in $\mathcal{E}$

Def 9.7. : The solution for a unification problem $U(A_1, A_2)$ is the set of unifiers $\{\sigma \mid \sigma(A_1) = \sigma(A_2)\}$. For example, let $\mathcal{E} := f(X) =^? f(g(Y))$, solutions are, e.g.

- $[g(a)/X], [a/Y]$
- $[g(g(a))/X], [g(a)/Y]$
- $[g(Z)/X], [Z/Y]$
- $[g(Y)/X]$

Def 9.8. : Let σ and θ be substitutions and $W \subseteq \mathcal{V}_\iota$, we say that a substitution σ is **more general** than θ on W (write as $\sigma \leq \theta[W]$), iff there is a substitution ρ, such that $\theta = \rho \circ \sigma[W]$, where $\sigma = \rho[W]$, iff $\sigma(X) = \rho(X)$ for all $X \in W$

- The substitution $\sigma = [g(Z)/X, Z/Y]$ is more general than $\theta = [g(a)/X, a/Y]$ because there exists $\rho = [a/Z]$ such that:

$$\theta = \rho \circ \sigma$$

  This identity $\theta = \rho \circ \sigma$ is verified by applying the composition to the variables in the domain:

  − $(\rho \circ \sigma)(X) = \rho(\sigma(X)) = \rho(g(Z)) = g(a)$
  − $(\rho \circ \sigma)(Y) = \rho(\sigma(Y)) = \rho(Z) = a$

  Since the mapping for each variable matches θ, we confirm that θ is a specialization of σ via ρ.

- Likewise, σ is more general than $\theta' = [g(g(a))/X, g(a)/Y]$ via $\rho' = [g(a)/Z]$, since:

$$\theta' = \rho' \circ \sigma$$

  where $(\rho' \circ \sigma)(X) = g(g(a))$ and $(\rho' \circ \sigma)(Y) = g(a)$.

Def 9.9. MGU: σ is called a **most general unifier (mgu)** of A_1, A_2, iff it is minimal in $U(A_1, A_2)$ **with respect to** the more general ordering ($\leq$) restricted to the variables occurring in the problem: $W = \mathrm{free}(A_1) \cup \mathrm{free}(A_2)$.

Def 9.10. : We call a pair $A =^? B$ **solved** in a unification problem $\mathcal{E}$, iff A is a variable X, $\mathcal{E} = X =^? A \wedge \mathcal{E}'$, and $X \notin (\mathrm{free}(A) \cup \mathrm{free}(\mathcal{E}'))$. We call a unification problem $\mathcal{E}$ a **solved form**, iff all its pairs are **solved**.

Lemma 9.2. Solved forms are of the form $X^1 =^? B^1 \wedge \cdots \wedge X^n =^? B^n$ where X^i are distinct and $X^i \notin \mathrm{free}(B^j)$

Def 9.11. $\mathcal{E}_\sigma$: Any substitution $\sigma = [B^1/X^1], \cdots, [B^n/X^n]$ induces a **solved unification problem**

$$\mathcal{E}_\sigma := (X^1 =^? B^1 \wedge \cdots \wedge X^n =^? B^n)$$

Lemma 9.3. If $\mathcal{E} = X^1 =^? B^1 \wedge \cdots \wedge X^n =^? B^n$ is a solved form, then $\mathcal{E}$ has the unique MGU $\sigma_\epsilon := [B^1/X^1], \cdots, [B^n/X^n]$.

Def 9.12. $\mathcal{U}$: The inference system $\mathcal{U}$ consists of the following inference rules:

$$\frac{\mathcal{E} \wedge f(A^1, \cdots, A^n) =^? f(B^1, \cdots, B^n)}{\mathcal{E} \wedge A^1 =^? B^1 \wedge \cdots \wedge A^n =^? B^n} \; \mathcal{U}_{dec} \qquad\qquad \frac{\mathcal{E} \wedge A =^? A}{\mathcal{E}} \; \mathcal{U}_{triv}$$

$$\frac{\mathcal{E} \wedge X =^? A \quad X \notin free(A) \quad X \in free(\mathcal{E})}{[A/X](\mathcal{E} \wedge X =^? A)} \; \mathcal{U}_{elim} \qquad\qquad \frac{\mathcal{E} \wedge f(X^1, \cdots, X^n) =^? g(Y^1, \cdots, Y^n)}{\bot} \; \mathcal{U}\bot$$

Unification Example : Let $\mathcal{E} := f(g(X,X),h(a)) =^? f(g(a,Z),h(Z))$

$$\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{\cfrac{f(g(X,X),h(a)) =^? f(g(a,Z),h(Z))}{g(X,X) =^? g(a,Z) \wedge h(a) =^? h(Z)}\,\mathcal{U}_{dec}}{X =^? a \wedge X =^? Z \wedge h(a) =^? h(Z)}\,\mathcal{U}_{dec}}{X =^? a \wedge X =^? Z \wedge a =^? Z}\,\mathcal{U}_{dec}}{X =^? a \wedge a =^? Z \wedge a =^? Z}\,\mathcal{U}_{elim}}{X =^? a \wedge Z =^? a \wedge a =^? a}\,\mathcal{U}_{elim}}{X =^? a \wedge Z =^? a}\,\mathcal{U}_{triv}$$

The final $\mathcal{E}$ is in solved form since all the pairs are solved. Hence $\mathcal{E}$ has the unique MGU $\sigma_\epsilon := [a/X],[a/Z]$.

Unification Example : Let $\mathcal{E} := S_1(g(f(x),g(x,y)),y) =^? S_1(g(z,v),f(w))$

$$\cfrac{\cfrac{\cfrac{\cfrac{S_1(g(f(x),g(x,y)),y) =^? S_1(g(z,v),f(w))}{g(f(x),g(x,y)) =^? g(z,v) \wedge y =^? f(w)}\,\mathcal{U}_{dec}}{z =^? f(x) \wedge v =^? g(x,y) \wedge y =^? f(w)}\,\mathcal{U}_{dec}}{z =^? f(x) \wedge v =^? g(x,y) \wedge y =^? f(w)}\,\mathcal{U}_{elim}}{z =^? f(x) \wedge v =^? g(x,f(w)) \wedge y =^? f(w)}\,\mathcal{U}_{elim}$$

The final equation $z =^? f(x) \wedge v =^? g(x,f(w)) \wedge y =^? f(w)$ is in solved form.
The MGU is $\{z \to f(x), v \to g(x,f(w)), y \to f(w)\}$.

Unification Example : Let $\mathcal{E} := S_2(g(f(x),g(x,u)),f(y),z) =^? S_2(g(g(g(u,v),f(w)),f(c)),f(g(u,v)),f(c))$ (not unifiable)

$$\cfrac{\cfrac{\cfrac{S_2(g(f(x),g(x,u)),f(y),z) =^? S_2(g(g(g(u,v),f(w)),f(c)),f(g(u,v)),f(c))}{g(f(x),g(x,u)) =^? g(g(g(u,v),f(w)),f(c)) \wedge f(y) =^? f(g(u,v)) \wedge z =^? f(c)}\,\mathcal{U}_{dec}}{f(x) =^? g(g(u,v),f(w)) \wedge g(x,u) =^? f(c) \wedge f(y) =^? f(g(u,v)) \wedge z =^? f(c)}\,\mathcal{U}_{dec}}{\bot}\,\mathcal{U}\bot$$

Lemma 9.4. $\mathcal{U}$ is sound: $\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $U(\mathcal{F}) \subseteq U(\mathcal{E})$

Lemma 9.5. $\mathcal{U}$ is complete: $\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ implies $U(\mathcal{E}) \subseteq U(\mathcal{F})$

Lemma 9.6. $\mathcal{U}$ is confluent: the order of derivations does not matter

Corollary 9.7. First-Order unification is unitary: i.e. MGUs are unique up to renaming of introduced variables
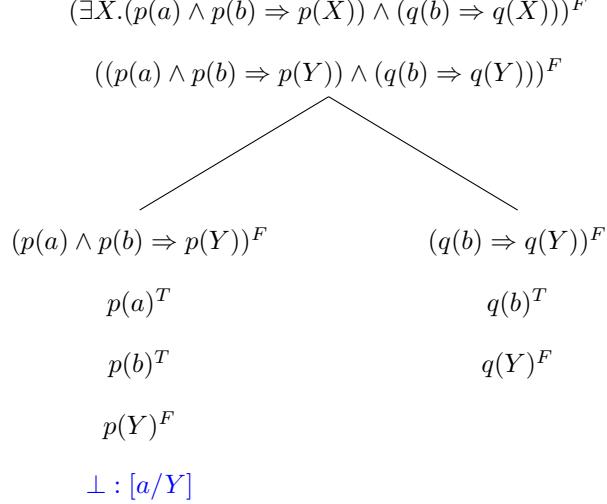
Lemma 9.8. $\mathcal{U}$ is terminating; any $\mathcal{U}$--derivation is finite

Def 9.13. : We call an equational problem $\mathcal{E}$ $\mathcal{U}$-**reducible**, iff there is a $\mathcal{U}$-step $\mathcal{E} \vdash_{\mathcal{U}} \mathcal{F}$ from $\mathcal{E}$

Lemma 9.9. If $\mathcal{E}$ is unifiable but not solved, then it is $\mathcal{U}$-reducible

Corollary 9.10. First-Order unification is decidable in PL^1

Recall: in $\mathcal{T}_1^f$, the rule $\mathcal{T}_1^f \bot$ instantiates the whole tableau. The problem is that there might be more than one $\mathcal{T}_1^f \bot$ opportunity on a branch and choosing which one makes a difference. For example, in the following proof, if we choose to close first on the right branch, the tableau does not close! However, choosing the other $\mathcal{T}_1^f \bot$ in the left branch allows closure!

$$(\exists X.(p(a) \wedge p(b) \Rightarrow p(X)) \wedge (q(b) \Rightarrow q(X)))^F$$

$$((p(a) \wedge p(b) \Rightarrow p(Y)) \wedge (q(b) \Rightarrow q(Y)))^F$$

$(p(a) \wedge p(b) \Rightarrow p(Y))^F$	$(q(b) \Rightarrow q(Y))^F$
$p(a)^T$	$q(b)^T$
$p(b)^T$	$q(Y)^F$
$p(Y)^F$	
$\bot : [a/Y]$	

Therefore, when implementing $\mathcal{T}_1^f$, there are two ways of systematic search, namely **backtracking search** back over $\mathcal{T}_1^f \bot$ opportunities and saturate without $\mathcal{T}_1^f \bot$ and find **spanning matings**.

Def 9.14. Spanning Mating: Let $\mathcal{T}$ be a $\mathcal{T}_1^f$ tableau, then we call a unification problem $\mathcal{E} := A_1 =? B_1 \cdots A_n =? B_n$ a **mating** for $\mathcal{T}$, iff A_i^T and B_i^F occur in the same branch in $\mathcal{T}$. We say taht $\mathcal{E}$ is a **spanning mating**, if $\mathcal{E}$ is unifiable and every branch $\mathcal{B}$ of $\mathcal{T}$ contains A_i^T and B_i^F for some i.

Theorem 9.11. A $\mathcal{T}_1^f$-tableau with spanning mating induces a closed $\mathcal{T}_1$ tableau.