Computation and Logic Definitions

- **True and False**
  - Also called Boolean **truth values**, True and False represent the two values or states an atom can assume.
    We can use any two distinct objects to represent truth values, for example,

True	False
1	0
$\top$	$\bot$

- **Sets and Subsets**
  - A **set** S is a collection of elements. We write $x \in S$ to mean that x is an element of S and $y \notin S$ to mean that y is not an element of S. For example the set $\mathbb{B}$ of Boolean truth values { True, False } has two elements.

    $\text{True} \in \mathbb{B}$ $\quad \text{False} \in \mathbb{B}$ $\quad 42 \notin \mathbb{B}$

  - A is a **subset** of B, in symbols, $A \subset B$,
    iff every element of A is an element of B,
    which means that, for all $x \in A.\ x \in B$

    In this case we also say B is a superset of A, in symbols, $B \supset A$
  - A is a **proper subset** of B if $A \subset B$ and $A \neq B$, i.e. if there is some element of B that is not in A.

- **Atom**
  - These are the basic propositions (often represented by single letters) that can be in one of two states, true or false.
    They the variables in Boolean algebra, sometimes called **propositional variables**.

- **Expression/Well-Formed Formula/WFF**
  - Expressions are built from the atoms by applying the **logical connectives** (negation, conjunction, disjunction, implication, etc.)
    - Every atom is a WFF
    - If P and Q are WFFs so are

$P \vee Q$	$P \wedge Q$
$P \rightarrow Q$	$\neg P$

    We sometimes consider languages with more (e.g. $\oplus$, $\downarrow$, $\leftrightarrow$), or fewer connectives.
  - Expressions that are not atoms are called **compound** expressions.

- **Negation :**
  - When an expressions is negated using NOT ($\neg$)

- **Literal**
    - A literal is either an atom or its negation.
      If L is the literal ¬A then when talking about literals we may write ¬L for the literal A. (If K is the literal B then ¬K is the literal ¬B.)

- **Valuation**
    - A **valuation** is an assignment of truth values to a set of atoms.
      If a valuation is **total** if it assigns a value to every atom.
    - Any total valuation V determines a value for each expression. The value of a compound expression is computed by applying the truth table corresponding to each logical connective occurring in the expression, to the values of its subexpressions.
      Using Haskell notation for the operations on Boolean values we can express this as follows:

      V(P ∨ Q) = V(P) || V(Q)
      V(P ∧ Q) = V(P) && V(Q)
      V(P → Q) = if V(P) then V(Q) else True
      V(¬P) = not (V(P))

      We often use valuations to represent the states of some system, and refer to the valuations that make an expression E true, as the set of states in which E is true.

      E is true in state V iff V(E) = True

- **How do valuations relate to Venn diagrams?**
    - A Venn-Diagram represents the Boolean function corresponding to an expression. Each region (by which we mean, each area within the diagram bounded by arcs of the circles) of the Venn diagram corresponds to a valuation. The colored regions represent the valuations that make the expression true.
    - Two expressions are equivalent iff have the same Venn diagram. This means that the same valuations
    - In a three-circle Venn-Diagram we represent sets of states for a system with 8 states encoded by three state variables. Each region of the diagram represents one of the 8 states, and we colour the regions corresponding to the states in the set of interest.

- **Satisfiability**
    - An expression is **satisfiable** if and only if there exists some valuation of the atoms which makes the expression true.

- **Tautology**
    - An expression is tautologous if and only if it is true for all valuations of its atoms.

- **Contingent**
    - An expression is contingent if and only if it is satisfiable but not tautologous, that is, it is true for some valuations, but not all.

- **Consistency**
    - A set of expressions is **consistent** if there is some valuation that makes every expression in the set true.
    - A set of expressions is **inconsistent** if no valuation makes every expression in the set true.

- **Boolean Algebra**
    - If E and F are Boolean expressions, we write E = F to mean that every valuation V gives the same value to E and F
      for all V. V(E) = V(F)
    - Boolean Algebra is the study of such equations. For example,
      $(A \wedge B) \vee C = (A \wedge C) \vee (B \wedge C)$ (distributivity)

- **Clausal Form**
    - In Boolean Algebra the operations of disjunction, $\vee$, and conjunction, $\wedge$, are symmetric, associative, and idempotent. So any combination of disjunctions, **or** any combination of disjunctions can be written without parentheses or repetition.
      We can write $\bigwedge S$ and $\bigvee S$ for the conjunction/disjunction of a finite set of expressions, S.
      For V a valuation,
      $V(\bigwedge S) = \top$ iff for all $e \in S$. $V(e) = \top$
      $V(\bigvee S) = \top$ iff for some $e \in S$. $V(e) = \top$
      These definitions tell us that we should interpret
      $\bigvee\{\}$ as $\bot$ and $\bigwedge S$ as $\top$
    - Furthermore, $X \vee A \vee \neg A = X \vee \top = \top$ so any disjunctive combination of literals can be rewritten either as $\top$, or as the disjunction of a set of literals in which no atom occurs both positively and negatively. We call a set S of literals in which no atom occurs both positively and negatively a **clause**, and interpret it as representing the disjunction $\bigvee S$. When talking about clauses, we may write $\bot$ for the empty clause
    - A clausal form is a set C of clauses, interpreted as representing the conjunction $\bigwedge C$. If the empty clause is in C , that is, $\bot \in C$, then $V(C) = \bot$, for every valuation V.
    - We can use Boolean algebra to convert any Boolean expression to an equivalent clausal form. This is called a **conjunctive normal form** (CNF), because it expresses any boolean function as a conjunction of disjunctions of literals.

- **Resolution**
  - Resolution is an inference rule for clauses given by:

$$\frac{X \vee A \quad Y \vee \neg A}{X \vee Y}$$

    More formally, if we have two clauses $\mathscr{X}$ and $\mathscr{Y}$ such that $A \in \mathscr{X}$ and $\neg A \in \mathscr{Y}$, then by resolution we can derive the clause $(\mathscr{X} \cup \mathscr{Y}) \setminus \{ A, \neg A \}$, whose elements are those literals occurring in $\mathscr{X}$ or $\mathscr{Y}$ (or both) that do not mention A.
    The key property of this rule is that it is sound:
    If $V(\mathscr{X}) = \top$ and $V(\mathscr{Y}) = \top$ then $V((\mathscr{X} \cup \mathscr{Y}) \setminus \{ A, \neg A \}) = \top$
    Or contrariwise,
    If $V((\mathscr{X} \cup \mathscr{Y}) \setminus \{ A, \neg A \}) = \bot$ then $V(\mathscr{X}) = \bot$ or $V(\mathscr{Y}) = \bot$
  - If we start from a clausal form C, and the resolution process produces the empty clause (which is not satisfied by any valuation), this shows us that the conjunction of the clauses excludes all valuations. Thus we can see that the clausal form has no satisfying valuations; it is contradictory.
  - Thus, resolution provides a method for determining whether a given set of constraints, expressed in CNF, is consistent.

- **Contradiction**
  - Contradiction occurs when a set of clauses is inconsistent. It contains clauses that contradict each other (there is no valuation of atoms that makes all clauses true)
    Note that we can construct an inconsistent set of n clauses
    Such that any proper subset of these is consistent.
    (Hint: construct an inconsistent cycle of implications.)

- **DNF (Disjunctive Normal Form) :**
  - An expression is in DNF if it is a disjunction of conjunctions of literals. So $\{\{A \wedge B\} \vee \{C \wedge D\}\}$ is an example.

- **Sequent**
  - Premises |- Conclusions , where the premises on the LHS and the conclusions on the RHS are both finite sets of expressions.
  - An entailment is a sequent with exactly one conclusion

- **Counterexample**
  - A valuation is a counterexample to a sequent if it makes everything on the LHS (left-hand side) of the sequent true and everything to the RHS (right-hand side) of the sequent false.
  - An entailment is simply a sequent with exactly one expression on the RHS. A valuation is a counterexample to an entailment if it makes everything on the LHS (left-hand side) of the turnstile true and the expression on the RHS (right-hand side) of the entailment false.

- **Valid and Invalid**
  - A sequent is valid if it has no counterexample. We can also express this positively, for both entailments and sequents
    - An entailment is valid iff every valuation that makes all of the premises true also makes the conclusion true.
    - A sequent is valid iff every valuation that makes all of the premises true also makes at least one of the conclusions true.

- **Rule**
  - A rule of the form $\frac{\beta_1 \quad \cdots \quad \beta_n}{\alpha}$ has a finite set of **assumptions**, $\beta_0 \ldots \beta_{n-1}$ and a single **conclusion**, α. In different proof systems the assumptions and conclusions may be expressions, entailments, or sequents. We focus on Gentzen's sequent calculus.
  - A proof tree is constructed by connecting rules so that the conclusion of one rule becomes an assumption of another.
  - A proof is a proof tree such that every branch terminates in a rule with no assumptions. It proves the sequent at the root of the tree. (In Gentzen's sequent calculus, the only rule with no assumptions is the immediate rule, (I).)
  - A sequent is provable iff it has a proof.

- **Sound**
  - A proof system is sound if any sequent that is provable is valid.
  - A rule is sound iff whenever all of the assumptions are valid, then so is the conclusion.
  - So, if all of the rules we use are sound then any provable sequent is valid - and the proof system is sound.
    In fact this is a necessary and sufficient condition for soundness.
  - We can also show that a rule is sound by showing that any counterexample to the conclusion is a counterexample to (at least) one of the assumptions.

- **Complete**
  - A proof system is complete if every sequent that is valid is provable.
  - One way to show that a proof system is complete is to show that if a sequent is not provable, then it has a counterexample.
  - Some authors say that a rule is complete if a counterexample to any of the assumptions provides a counterexample to the conclusion. However, having a set of complete rules, in this sense, is neither a necessary nor a sufficient condition for the completeness of a proof system.

    - Nevertheless, for Gentzen's system the completeness of each rule is a key component of the proof of completeness.

- **Automaton (DFA/NFA)**
    - An automaton has states Q, including a set A of **accepting states** and a set B of **start states**, an alphabet Σ, and transitions δ. Each transition is an arrow between two states (which may be the same state), labelled with a symbol from the alphabet Σ.

- **State (DFA/NFA)**
    - A position within an Automaton.

        - Start State: A state in the set of start states.
        - Accepting State: A state in the set of accepting states
        - Alphabet: A set of **symbols** used to label transitions
        - Transition $s : a \rightarrow b$
          An ordered pair (a, b) of states, together with a symbol s from the alphabet.

- **Trace**
    - A trace for a string of length n is a sequence of n+1 states (say states $s_0 \ldots s_n$) connected by a chain of n transitions, $t_i : s_{i-1} \rightarrow s_i$, for i=1…n, such that the i^{th} transition is labelled by the i^{th} symbol in the string.
    - A trace is just the path that you follow given a string. Like how you might "trace" your finger over the arrows to keep track of what state you're in.
    - An automaton **accepts** a string σ if there is a trace $s_0 \ldots s_n$ for σ such that s_0 is a start state and s_n is an accepting state.
    - The language accepted by the automaton is the set of strings that the automaton accepts.

- **Regular Expression (RegEx):**
    - We define a regular expression for an alphabet Σ
        - Every symbol s in Σ is a regular expression
        - If R and S are regular expressions so are
            - R* iteration
            - RS sequence
            - R|S alternation

          The precedence of these operations is
          Iteration > sequence > alternation
          and we use parentheses where necessary.

This document was written by
Asmita Dulan, Riccardo Fiorista, and Likhitha Sai,
and revised by Michael Fourman.