Program Verification & Testing; Review of Propositional Logic

CS 536: Science of Programming, Fall 2022

A. Why

- Course guidelines are important.
- Understanding what Science of Programming is is important.
- Reviewing/overviewing logic is necessary because we'll be using it in the course.

B. Outcomes

At the end of this class, you should

- Know how the course will be structured and graded.
- Have practiced some of the techniques we'll be using in class.
- Know what Science of Programming is about and how it differs from and is related to program testing.
- Understand what a propositional formula is, how to write them, how to tell whether one is a tautology or contradiction using truth tables, and see a basic set of logical rules for transforming propositions.

C. Introduction and Welcome

- The course webpages for are at http://cs.iit.edu/~cs536/ . You're responsible for the information there, even if you don't read it.
- We'll use myIIT → Blackboard for submitting homework and viewing grades.
- The lectures are automatically recorded and posted to Blackboard under Panopto.

D. Course Prerequisite: Basic Logic

- The course prerequisite formally is CS 401. In reality, what you need is some background in boolean logic (propositions and predicates) and some comfort in syntactic operations and formal languages.
- For a very rough assessment of your preparedness for this course, study the following questions: If you get all five correct, you have more than enough background for this course;. If you get five correct, you're probably okay but will need to brush up. If you get none correct, consider quickly dropping this course.

Prerequisite Quiz

1. Are 2+2 and 4 syntactically equal and why?
2. If AND higher precedence than OR and OR is left associative, how do you parenthesize V AND W OR X OR Y?
3. If p and q are propositions then what are the contrapositive, converse, and inverse of the implication $p \rightarrow q$ and how are they related?
4. How do you pronounce $(\neg \forall x \in \mathbb{Z} \,.\, \exists y \in \mathbb{Z} \,.\, y^2 < x)$ in English, and is it true?
5. What's the difference between saying that a predicate p is valid versus saying that you have a formal proof of p?*

- We'll quickly review basic logic in class. If you want other references to study, try the references linked to the home page.

E. So What Is Science of Programming Anyway?

- Science of Programming is about ***program verification***.
- Program verification aims to get reliable programs by discerning properties about programs.
  - It's harder to do this by writing programs and then proving them correct.
  - In practice, it's better to reason about programs as we write them.
- For this class, we'll look at a simple programming language. The syntax will be simple (that's not the important part).
  - What's important is formally (= mathematically, logically) specifying the semantics of programs and connecting them to the semantics of logical statements.
  - Put another way, if we want to be very sure about whether a program works or not, we have to be sure what we want the program to do (hence logical statements), and we have to be sure how programs execute (hence formal semantics), and we have to be able to connect the two (which will lead to studying formal rules for logical reasoning about programs).

F. Neither Reasoning or Testing is Completely Sufficient

- Let's contrast program verification and program testing.
  - In testing, we run a program and verify that it behaves correctly.
  - In verification, we reason about a program to predict that it will behave correctly.
- We need both testing of programs and reasoning about programs; neither is always better than the other.

---

* Answers: (1) No, because operator expressions aren't constants. (2) ((V AND W) OR X) OR Y. (3) Contrapositive: $\neg q \rightarrow \neg p$; Converse: $q \rightarrow p$; Inverse: $\neg p \rightarrow \neg q$? An implication and its contrapositive are semantically equivalent, as are the converse and inverse, but an implication and its converse are not. (4) "It's not the case that for every integer x, there exists an integer y such that y squared is less than x." It's true (try x = 0). (5) Validity is a semantic claim; having a formal proof of it is a syntactic claim.

  - When we reason about a program, we can make mistakes or overlook cases. We need testing as a reality check to show that our reasoning is sound.
  - In the other direction, complete testing of a program might involve a too-large set of test cases (infinite or close enough to infinite) to be practical. So we reason about our programs to identify a practical number of test cases that should represent all the possible test cases.
- As an example, say our specification is "If $z \geq c$ before the program, then $z > c$ after it", where the program is just "add x to z, but only if x is nonnegative."
  - In C, we can write */* z >= c */ if (x >= 0) z = z+x; else ++z; /* z > c */*
  - To figure out which test cases are good, we reason about how the statements and properties interact.
  - E.g., take $x \geq 0$ (and its negation $x < 0$) and break up ≥ into separate > and = cases ($x > 0$, $x = 0$), to get $x < 0$, $x = 0$, and $x > 0$ as the general set of cases. If we think $x = -1$ and 1 are good enough generalizations of $x < 0$ and $x > 0$, then we're done: Our test cases are $x = -1$, $x = 0$, $x = 1$.
  - If we decide we want to be more thorough and treat $x = -1$ and $x < -1$ as different cases (and $x > 1$ similarly), we can turn them into $x = -2$ and $x = 2$, and end up with five test cases, $x = -2$, $x = -1$, $x = 0$, $x = 1$, $x = 2$.
  - Of course, if we keep breaking edge cases off of the < and > tests, we could get $x = -3$ and $x = 3$, then $x = -4$ and $x = 4$, and so on to infinity. A big part of testing is figuring when to stop doing all this.

G. Type-Checking as a Kind of Program Verification

- ***Static*** (i.e., compile-time) ***type-checking*** is an example of program verification: We analyze a program textually to reason about how it uses types, to check for type-correctness.
- The reasoning is symbolic / textual because we aren't actually running the program, so a type-checker is a mechanical theorem prover for judgements of the form "this variable or expression has type …" and "This operation is type-correct."
  - E.g., if variables x and y are of type integer, then $x+1$ and x/y are integers, so $x+1 = x/y$ is type-correct, etc. (Note x/y might still cause a runtime error, but it wouldn't be a type error.)
- A ***strong type-checker*** produces proofs that provide complete evidence for type safety. A ***weak type-checker*** produces proofs that provide only partial evidence for type safety.
  - E.g., type-checkers for Haskell or Standard ML are very strong; they guarantee type safety. (Note: You might still get runtime errors, but not for type-incorrect operations.)
  - However, type-checkers for C are weak; they have to assume you know what you're doing when you cast pointers.

H. Reasoning About One State of Memory vs Many States of Memory

- In testing, we have a finite number of specific values we use for our variables. We can verify that our program works with those specific values.
- In program verification we aim to say that our programs work in all possible cases. Typically, we have an infinite number of cases[†]. (Actually, it's a finite number, since memory is finite, but who wants to deal with, e.g., 2^{32} separate individual tests for an integer variable x?)
- In program verification, we use ***predicates*** like $x > 0$ to stand for a possibly infinite number of values. (A predicate is a syntactic object that has a truth value once you plug in specific values for its variables.)
- Using predicates, we can talk about an infinite number of possible execution paths simultaneously. Instead of actually executing a program, we simulate its execution symbolically, using rules of logic to manipulate our predicates. "If $x > 0$, then after adding 1 to x, we have $x > 1$" stands for an infinite number of execution paths.
- One way to describe program verification is that instead of actually executing a program on one set of inputs to get one set of outputs, we simulate execution on sets of states using reasoning on predicates. We describe a set of input states using a logical predicate and reason about the possible output states using rules of logic plus rules for program execution.
- So to do program verification we need predicates to describe sets of memory states, rules of logic to reason about predicates, plus rules for how our programs execute (i.e., how they take and modify memory states).

I. Logic Review/Overview, Part 1: Propositional Logic

- If you weren't a CS major as an undergrad and haven't seen propositional and predicate logic before, you should study up on it (see ***Course Prerequisites: Basic Logic*** on page 2.)
- ***Propositional logic*** is logic over ***proposition variables***, which are just variables that can have the values true or false. In propositional logic we study the logical connectives and ($\wedge$), or ($\vee$), not ($\neg$), implication ($\rightarrow$), and biconditional ($\leftrightarrow$) operating over variables that have true or false as their values. In computer science terms, propositional logic is the logic used for boolean expressions: True and false are boolean constants, and the connectives are boolean operators (in C, $\wedge$, $\vee$, $\neg$, and $\leftrightarrow$ are written *&&*, *||*, *!*, and ==).
- ***Notation***: Typically we'll use p, q, ... for preposition variables or propositions and T, F for true and false.

Terminology

- $p \wedge q$ is the ***conjunction*** or ***logical and*** of p and q. p and q are ***conjuncts*** of $p \wedge q$.
- $p \vee q$ is the ***disjunction*** or ***logical or*** of p and q. p and q are ***disjuncts*** of $p \vee q$.

[†] Actually, it's probably a finite number of cases but still so many that "infinity" is a decent generalization.

- $p \rightarrow q$ is the ***implication*** or ***conditional*** of p and q. p is the ***antecedent*** or ***hypothesis*** and q is the ***consequent*** or ***conclusion***.

Other Ways to Phrase Implications

- Other phrasings of $p \rightarrow q$: "***if*** p ***then*** q"; "p is ***sufficient for*** q"; "p ***only if*** q", "q ***if*** p"
- Other phrasings of $q \rightarrow p$: $p \leftarrow q$, "p is ***necessary for*** q"; "p ***if*** q"; "***if*** q ***then*** p", "q ***only if*** p".

Biconditional

- $p \leftrightarrow q$ is the ***equivalence*** or ***biconditional*** of p and q; they are both true or both false. p is the ***antecedent*** or ***hypothesis*** and q is the ***consequent*** or ***conclusion***.
- Note $\leftrightarrow$ is not the same as "equivalence" in the sense that "p is equivalent to q, which is equivalent to r, so p is equivalent to r". For two items, $p \leftrightarrow q$ is true exactly when p and q are both true or both false, so e.g., F $\leftrightarrow$ F evaluates to T. But for three items, $\leftrightarrow$ doesn't behave as you might expect: Since F $\leftrightarrow$ F evaluates to T, we can substitute it for the first T in T $\leftrightarrow$ T and get that (F $\leftrightarrow$ F) $\leftrightarrow$ T evaluates to T.
- The kind of equivalence where "p is equivalent to q, and q is equivalent to r, so p is equivalent to r" is called "logical equivalence" and it's related to $\leftrightarrow$ but not exactly the same. We'll look at it in a bit.

Precedences and Associativities for Propositional Operators

- **Precedences**: For the precedences of propositional operators, let's use $\neg$, $\wedge$, $\vee$, $\rightarrow$, $\leftrightarrow$ (going from strong to weak). E.g., $((((\neg p) \wedge q) \vee r) \rightarrow s) \leftrightarrow t$ means $\neg p \wedge q \vee r \rightarrow s \leftrightarrow t$. E.g., we'll take $p \rightarrow q \leftrightarrow p \vee \neg q$ to mean $(p \rightarrow q) \leftrightarrow (p \vee \neg q)$. This is pretty standard except sometimes people take $\rightarrow$ and $\leftrightarrow$ as having the same precedence.
- ***Associativity***: $\wedge$ and $\vee$ are associative, so $((p \wedge q) \wedge r) \equiv (p \wedge (q \wedge r))$. Let's make these operators left associative, so the full parenthesization of $p \wedge q \wedge r$ will be $((p \wedge q) \wedge r)$.
- Implication is not associative; we'll take it to be right associative.
  - To see non-associativity, compare ((F $\rightarrow$ T) $\rightarrow$ F) and (F $\rightarrow$ (T $\rightarrow$ F)).
  - Right associativity tells us that $p \rightarrow q \rightarrow r$ means $(p \rightarrow (q \rightarrow r))$.
- For the biconditional ($\leftrightarrow$), we'll use right associativity: $p \leftrightarrow q \leftrightarrow r$ means $(p \leftrightarrow (q \leftrightarrow r))$.
  - Note $p \leftrightarrow q$ evaluates to T if p and q evaluate to the same value, T or F.
  - On the other hand, $p \leftrightarrow q \leftrightarrow r$ means $(p \leftrightarrow (q \leftrightarrow r))$, which leads to possibly-puzzling properties like T $\leftrightarrow$ T $\leftrightarrow$ T means (T $\leftrightarrow$ (F $\leftrightarrow$ F)), which evaluates to true.

Semantic Equality

- ***Semantic equality*** is equality of meanings or results. This is usually what we mean when we write "=": 2+2 = 4, $a+b = b+a$
- For propositions, we'll use $\Leftrightarrow$ to indicate semantic equality. (This is the "logical equivalence" we'll discuss in a bit.)
  - Example: You can distribute $\vee$ over $\wedge$: $(p \wedge q) \vee r \Leftrightarrow (p \vee r) \wedge (q \vee r)$

- For propositions, where we have only the values T and F (and only boolean variables), semantic equality can be mechanically determined (though for propositions, it can take time exponential in the number of basic variables).
- Later, with predicates, semantic equality can be impractical or even impossible to determine, so we usually fall back on a property that's easier to determine, namely, syntactic equality.

Syntactic Equality

- ***Syntactic equality*** (written ≡) means equality as structured text: Two expressions or propositions are syntactically equal if they are textually identical — with one exception: We'll ignore redundant parentheses. E.g., (*1*2*)+*3* ≡ *1*2+3*. We'll use ≢ for syntactic inequality. E.g., *2+2* ≢ *4*.
- We'll consider three kinds of redundancy for parentheses:
  - ***Precedence***: $(p \wedge q) \vee r \equiv p \wedge q \vee r$ because "∧" has higher precedence than "∨".
  - ***Left or Right Associativity***: *a* - *b* - *c* ≡ (*a* - *b*) - *c* because "-" is left associative. Similarly, $p \rightarrow q \rightarrow r \equiv p \rightarrow (q \rightarrow r)$ because "→" is right associative.
  - ***Associative Operators***: For an associative operator, all parenthesizations will be syntactically equal. E.g., $(p \wedge q) \wedge r \equiv p \wedge (q \wedge r)$. But (*a* - *b*) - *c* ≢ *a* - (*b* - *c*) because "-" is not associative.
- We are not going to take commutativity of operators into account, so $p \wedge q \not\equiv q \wedge p$. Of course, they evaluate to the same truth value, so we can see that syntactically unequal items can stand for the same value. We leave out commutativity because first, it makes "≡" easier to calculate. Second, without commutativity, the *non-parenthesis symbols* of two ≡ items have to appear in the same order. E.g., no parenthesizations of 1+3+2 and 1+2+3 make them ≡.
- What about operator pairs like * and / or + and – where the members of the pair have equal precedence and one of the pair (but not both) are associative. In particular, do we want (*x* * (*y* / *z*)) ≡ ((*x* * *y*) / *z*) or (*x* + (*y* – *z*)) ≡ ((*x* + *y*) – *z*)? It turns out we'll take each pair to be ≢, but the justification has to do with how syntactic and semantic equality are related, so we'll put off the explanation for a bit.

Syntactic Equality Versus Semantic Equality

- **Why Use Syntactic Equality?** We often want to know whether two items are semantically equal, but depending on the kind of item, semantic equality can be hard or even impossible to calculate. Syntactic equality is easy to calculate, and if we define ≡ carefully, then we can guarantee that if two items are ≡, then they're semantically =. In other words, we use syntactic equality to be a rough approximation of semantic equality — "rough' because two items can be ≢ but =.
- **Syntactic Equality Implies Semantic Equality**: Keeping this property in mind makes it easy to see why we can ignore redundant parentheses when determining ≡ because preserving = is what makes parentheses redundant. E.g., 1+2*3 ≡ (1+(2*3)), so they stand for the same value. Since "≡ implies =" is true, the contrapositive "≠ implies ≢" is also true. E.g., 2+2 ≠ 5, so 2+2 ≢ 5.
- **Semantic Equality Does Not Imply Syntactic Equality**: A separate question from "≡ implies =" is the converse: Does = imply ≡? It's easy to find examples where two expressions are = but not ≢. E.g., 2 + 2 ≢ 4. Similarly, *a* + 0 ≢ *a* and $p \wedge q \not\equiv q \wedge p$.

- **Back to mixing * and / (or + and -)**: Now let's go back to the question of "Should (x * (y / z)) $\equiv$ or $\not\equiv$ ((x * y) / z)?" On integers, the two expressions are not = because of truncation: (2 * (2 / 4)) = 2 * 0 = 0 but (2 * 2) / 4 = 4 / 4 = 1. On the other hand, on infinite precision reals, the two expressions are =. So the question of = or $\neq$ depends on the types of x, y, and z. Having to take the types of x, y, and z into consideration for determining $\equiv$ or $\not\equiv$ would make things more complicated, so it seems better to take our pair of expressions to be $\not\equiv$. More generally, we can say that if we have two operators that have the same precedence and are both associative, then it's okay to treat them as being associative.

Parenthesizations

- The ***minimal parenthesization*** of a syntactic item is the one with the fewest parentheses that preserves $\equiv$. (I.e., it is still $\equiv$ to the original.)
- For associative operators, let's omit parentheses inside sequences like $p \wedge q \wedge r$. We may still need them around the whole proposition, as in $p_1 \wedge (q_1 \vee q_2 \vee q_3)$
- The ***full parenthesization*** of an item is the one that preserves $\equiv$ and also includes parentheses around each operator expression (i.e., (op p) for a unary operator or (p op q) for a binary operator).
  - We'll omit parentheses around constants, variables, and already-parenthesized expressions; we don't want to be writing things like (((1) + (((2) * (3))))).
  - Parentheses around the whole propositions are implied if omitted[‡]. So the full parenthesization of 1 + 2 * 3 can be written as (1 + (2 * 3)) or 1 + (2 * 3).
  - For associative operators like + and *, let's write using left associativity, just to avoid having multiple correct results. So we'll take (((1 + 2) + 3) + 4) to be the full parenthesization of 1 + 2 + 3 + 4.
  - Note: If you include the outermost parentheses, then a fully parenthesized item has the same number of pairs of parentheses as it has number of operators. E.g., (1 + (2 * 3)) has two pairs of parentheses: one for the + and one for the *.

J. Semantics of Propositional Logic

- The typical semantics for propositional logic uses truth tables as below.

p	q	$p \wedge q$	$p \vee q$	$p \rightarrow q$	$p \leftrightarrow q$
F	F	F	F	T	T
F	T	F	T	T	F
T	F	F	T	F	F
T	T	T	T	T	T

p	$\neg p$
F	T
T	F

- Implication sometimes bothers people ("Why does F → T ?")

[‡] This is just a hack tossed in to save me if I forget to write the outermost parentheses sometimes.

- Basically, $p \to q$ means "p is less true than or equal to q".
- If you treat true, false, and $\to$ as being like 1, 0, and $\leq$, then F $\to$ T is like $0 \leq 1$.

States and Satisfaction

- Here's one way to talk about the truth of a proposition. First, we'll look at states — a state represents one truth table row of possible values for a set of proposition variables. Then we'll look at satisfaction, the notion that a proposition is true in that particular truth table row.
- **Definition**: A (**well-formed**) **state** σ is a set of pairs (a.k.a. bindings) of a proposition letter and a truth value where there's only one binding for any given variable. If a set of bindings σ has more than one binding for some variable, we'll say it's **ill-formed** as a state.
- **Examples**: Examples of states are $\{p = \mathrm{T}\}$ (has one binding), $\{p = \mathrm{F}, q = \mathrm{T}\}$ and $\{q = \mathrm{T}, p = \mathrm{F}\}$ (two ways to write the same state), and the empty state $\varnothing$ (the empty set of bindings). On the other hand, $\{p = \mathrm{T}, p = \mathrm{F}\}$ is ill-formed, since it has two bindings for p.
  - ***Note***: It's important for the bindings to involve only proposition letters, not more complicated propositions. E.g., $\{p \vee q = \mathrm{F}\}$ is not well-formed.
- **Definition**: A proposition is **satisfied** in (or by) a state if it evaluates to true in that state. E.g., $p \vee \neg q$ is satisfied in $\{p = \mathrm{T}, q = \mathrm{F}\}$. The proposition is **not satisfied** (or **unsatisfied**) in a state if it evaluates to false in that state.
- **Notation**: If σ is a state and p is a proposition, then $\sigma \models p$ means that σ satisfies p and $\sigma \not\models p$ means σ does not satisfy p. (The $\models$ symbol is a "double turnstile", if you haven't run across it before.)
- **Examples**: $\{p = \mathrm{T}\} \models p$, $\{p = \mathrm{F}\} \models \neg p$, and $\{p = \mathrm{T}, q = \mathrm{F}\} \models p \vee \neg q$. For nonsatisfaction, $\{p = \mathrm{T}\} \not\models \neg p$, $\{p = \mathrm{F}\} \not\models p$, and $\{p = \mathrm{T}, q = \mathrm{F}\} \models p \wedge \neg q$. A less-obvious case is $\varnothing \models \mathrm{T} \wedge (\mathrm{F} \to \mathrm{T})$. Here, the state is empty, but that's okay because the proposition doesn't contain any variables, just the constants T and F.
- **Note**: For satisfaction purposes, it's okay for a state to have unused bindings (bindings of variables that don't appear in the proposition). So if σ and τ are two states with $\sigma \subseteq \tau$, then if $\sigma \models p$, then $\tau \models p$. Since every state extends the empty set and $\varnothing \models \mathrm{T} \wedge (\mathrm{F} \to \mathrm{T})$, then this proposition is satisfied in every state.
- Though extra bindings are okay, not having enough bindings makes for a somewhat unsettled situation. For example, we can't say $\varnothing \models p \wedge q$ because we can't evaluate $p \wedge q$ in $\varnothing$ and get true. But we also can't say $\varnothing \not\models p \wedge q$, since we can't evaluate $p \wedge q$ and get false. Does this prevent us from making statements like $p \vee \neg p$ is satisfied by every state"?
- **Definition**: A state is **proper** for a proposition p if it includes bindings for (at least) all the variables of p. If σ is **improper** (i.e., not proper) for p, then we can't even evaluate p in σ, so we can't evaluate p and get true or to false, so we can't say $\sigma \models p$ and we can't say $\sigma \not\models p$. Since $\sigma \models$ or $\not\models p$ only makes sense if σ is proper, when we make statements like "p is satisfied in all σ", we'll quietly assume that we're talking about only the σ that are proper for p. So we can say "$\sigma \models p \vee \neg p$

for all σ" even though $p \vee \neg p$ certainly isn't satisfied in $\varnothing$ or in $\{q = \mathrm{T}\}$, for of the infinitely many examples.

- To sum up, for any arbitrary set of bindings σ and a proposition p, we can have four situations
  - σ is ill-formed (not a state at all)
  - σ is (well-formed but) improper for p
  - $\sigma \vDash p$ and $\sigma \nvDash \neg p$
  - $\sigma \nvDash p$ and $\sigma \vDash \neg p$
- And again, when we look at "all states for p," we mean only the well-formed proper states. We can now make statements like "if it is not the case that $\sigma \vDash p$, then $\sigma \nvDash p$" because we're ignoring the ill-formed and the improper σ.

Validity, Tautologies, and Logical Equivalence

- Now that we have the notion of a proposition being true in a given truth table row, we can go further and talk about a proposition being true in every truth table row.
- **Definition**: p is **valid** (notation: $\vDash p$) if $\sigma \vDash p$ for every σ. p is **invalid** (not valid), written $\nvDash p$, if this is not the case. (Remember, we're talking only about well-formed proper states here, so $\nvDash p$ is equivalent to "for some σ, $\sigma \nvDash p$.")
- **Examples**: $\vDash \mathrm{T}$, $\vDash \neg \mathrm{F}$ (the two simplest examples), $\vDash p \vee \neg p$, $\vDash (p \rightarrow q) \leftrightarrow (\neg p \vee q)$.
- If $\nvDash p$**,** then it is not the case that $\sigma \vDash p$ for every σ; this is equivalent to saying for some σ, $\sigma \nvDash p$. Since (by assumption) σ is proper, $\sigma \nvDash p$ is equivalent to $\sigma \vDash \neg p$.
- One way to categorize propositions is through their validity.
- **Definition**: p is a **tautology** if $\vDash p$, a **contradiction** if $\vDash \neg p$, and a ***contingency*** if $\nvDash p$ and $\nvDash \neg p$ (simultaneously). Another way to say this is that a tautology has a truth table column of all true; a contradiction has a column of all false, and a contingency has a mix of true and false (at least one row T and at least one row F).
- Some properties:
  - If p is a tautology, then $\neg p$ is a contradiction and vice versa.
  - If p is a contingency, then so is $\neg p$ and vice versa.
  - If p is not a tautology, then it is a contingency or a contradiction.
  - If p is not a contradiction, then it is a contingency or a tautology.
  - If p is not a contingency, then it is a tautology or a contradiction.

- **Definition**: Two propositions p and q are **logically equivalent** (written $p \Leftrightarrow q$) if $\vDash p \leftrightarrow q$. Note since $\vDash \mathrm{T} \leftrightarrow \mathrm{T}$ and $\mathrm{F} \leftrightarrow \mathrm{F}$ (and $\vDash \neg(\mathrm{T} \leftrightarrow \mathrm{F})$ and $\vDash \neg(\mathrm{F} \leftrightarrow \mathrm{T})$), p and q are logically equivalent if they have matching columns in their truth tables.
- **Notation**: "⇔" is often pronounced "if and only if", so $p \Leftrightarrow q$ is also often written as "p iff q".

- It's easy to show that $\Leftrightarrow$ is **transitive**: If $p_1 \Leftrightarrow p_2$ and $p_2 \Leftrightarrow p_3$, then $p_1 \Leftrightarrow p_3$. So $\Leftrightarrow$ is the notion of equivalence used when we write a sequence of step-by-step transitions like $p \rightarrow q \Leftrightarrow \neg p \vee q \Leftrightarrow q \vee \neg p \leftrightarrow \neg\neg q \vee \neg p \Leftrightarrow \neg q \rightarrow p$.
- $\Leftrightarrow$ on propositions is like = on arithmetic expressions: If $p \Leftrightarrow q$, then in any semantic context, we can always substitute one for the other. The context has to be semantic because $\Leftrightarrow$ has to do with the ... $\vDash$ ... relationships between states and propositions
- Though they are similar, it's important to keep the difference between $\Leftrightarrow$ and $\leftrightarrow$ straight in your head.
  - $\leftrightarrow$ is a symbol that can actually appear in a boolean expression. (In C, $\leftrightarrow$ is written ==.) I.e., $\leftrightarrow$ is a syntactic operator. On the other hand, $\Leftrightarrow$ doesn't appear in propositions because it indicates semantic equality.
  - $\Leftrightarrow$ is transitive and $\leftrightarrow$ is not transitive. Consider (F $\leftrightarrow$ F $\leftrightarrow$ T), which means (F $\leftrightarrow$ (F $\leftrightarrow$ T)), since $\leftrightarrow$ is right associative. Semantically, (F $\leftrightarrow$ (F $\leftrightarrow$ T)) $\Leftrightarrow$ (F $\leftrightarrow$ F) $\Leftrightarrow$ T
  - Second, iterated $\Leftrightarrow$ means all the propositions are logically equivalent to each other. E.g., (T $\vee$ T) $\Leftrightarrow$ (T $\vee$ F) $\Leftrightarrow$ T. For an analogy, remember how in algebra we might write "$(x+1)^2 + 3 = (x^2 + 2x + 1) + 3 = x^2 + 2x + 4$"? Here, "=" is being used on numbers the same way we use $\Leftrightarrow$ on propositions.
  - So $p \Leftrightarrow q \Leftrightarrow r \Leftrightarrow s$ means ($p \Leftrightarrow q$ and $q \Leftrightarrow r$ and $r \Leftrightarrow s$); i.e., all four are true or all four are false.
  - Compare this to F $\leftrightarrow$ F $\leftrightarrow$ T $\leftrightarrow$ T, which $\equiv$ (F $\leftrightarrow$ (F $\leftrightarrow$ (T $\leftrightarrow$ T))), which evaluates to true.

Relations Between Implications

- The ***contrapositive*** of $p \rightarrow q$ is $\neg q \rightarrow \neg p$; its ***converse*** is $q \rightarrow p$; its ***inverse*** is $\neg p \rightarrow \neg q$. An implication is $\Leftrightarrow$ to its contrapositive; similarly, the converse of an implication is $\Leftrightarrow$ to its inverse: $(p \rightarrow q) \Leftrightarrow (\neg q \rightarrow \neg p)$ and $(q \rightarrow p) \Leftrightarrow (\neg p \rightarrow \neg q)$

More Equivalences

- ***Definition of implication***: $p \rightarrow q \Leftrightarrow \neg p \vee q$.
- ***Negation of implication***: $\neg(p \rightarrow q) \Leftrightarrow p \wedge \neg q$.
  - Note $\neg(p \rightarrow q)$ and $(\neg p \rightarrow \neg q)$ are different: $\neg(p \rightarrow q) \Leftrightarrow (p \wedge \neg q)$ but $(\neg p \rightarrow \neg q) \Leftrightarrow (p \vee \neg q)$
- ***Definition of biconditional***: $(p \leftrightarrow q) \Leftrightarrow (p \rightarrow q) \wedge (q \rightarrow p)$
- ***Exclusive or*** of p and q: One of p and q is true but not the other. Turns out to be the opposite of the biconditional:
  - $(p \wedge \neg q) \veebar (\neg p \wedge q) \Leftrightarrow \neg(p \leftrightarrow q)$
- The usual "inclusive" or allows one or both of p and q to be true:
  - $(p \vee q) \Leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q) \vee (p \wedge q)$