Program Verification & Testing; Review of Propositional Logic

CS 536: Science of Programming, Fall 2022

Mon 2023-01-10: p.5

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 has 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 $p \to q$ 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?

For answers, see the footnote[1]

- 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.

---

[1] Answers: (1) No, because operator expressions aren't constants. (2) ((V AND W) OR X) OR Y. (3) Contrapositive: $\neg q \to \neg p$; Converse: $q \to p$; Inverse: $\neg p \to \neg q$. 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 $x = 1$ 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, namely $x = -2$, $x = -1$, $x = 0$, $x = 1$, $x = 2$.
  - 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[2]. (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. We say that p and q are ***conjuncts*** of $p \wedge q$.
- $p \vee q$ is the ***disjunction*** or ***logical or*** of p and q. We say that p and q are ***disjuncts*** of $p \vee q$.

---

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

- $p \to q$ is the ***implication*** or ***conditional*** of p and q. We say that p is the ***antecedent*** or ***hypothesis*** and q is the ***consequent*** or ***conclusion***.

Other Ways to Phrase Implications

- Other phrasings of $p \to q$: "***if*** p ***then*** q"; "p is ***sufficient for*** q"; "p ***only if*** q", "q ***if*** p"
- Other phrasings of $q \to p$: $p \leftarrow q$, "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". Equivalence is transitive: "If p is equivalent to q, and q is equivalent to r, then 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 we want 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 $\neg$, $\wedge$, $\vee$, $\to$, $\leftrightarrow$ (going from strong to weak). E.g., $\neg p \wedge q \vee r \to s \leftrightarrow t$ means $((((\neg p) \wedge q) \vee r) \to s) \leftrightarrow t)$ [2023-01-10]. E.g., we'll take $p \to q \leftrightarrow p \vee \neg q$ to mean $(p \to q) \leftrightarrow (p \vee \neg q)$.
  - Sometimes people take $\to$ and $\leftrightarrow$ as having the same precedence, but in general, it doesn't much matter.
- ***Associativity***: $\wedge$ and $\vee$ are associative, so $((p \wedge q) \wedge r)$ and $(p \wedge (q \wedge r))$ have the same logical value. Let's make these operators left associative, so the fully parenthesization of $p \wedge q \wedge r$ is $((p \wedge q) \wedge r)$.
- Implication is right associative: $p \to q \to r$ means $(p \to (q \to r))$.
  - Unlike $\wedge$ and or, $\to$ is not associative: $((F \to T) \to F)$ and $(F \to (T \to F))$ don't ~~always~~ [2023-01-10] evaluate to the same result
- For the biconditional ($\leftrightarrow$), we'll use right associativity: $p \leftrightarrow q \leftrightarrow r$ means $(p \leftrightarrow (q \leftrightarrow r))$.
  - The biconditional is indeed logical equivalence on two values: $T \leftrightarrow T$ and $F \leftrightarrow F$ both evaluate to T, but $T \leftrightarrow F$ and $F \leftrightarrow T$ both evaluate to F.
  - But it's not logical equivalence on three values. If p and q are both F and r is T, then p, q, and r are not equivalent, but $(p \leftrightarrow (q \leftrightarrow r))$ equals $(F \leftrightarrow (F \leftrightarrow T))$ equals $F \leftrightarrow F$ equals T.

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, which turns out to be what we want for logical equivalence.
  - ***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, when we add other kinds of values (like integers) to get 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 $\equiv$) 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., $(1*2)+3 \equiv 1*2+3$. We'll use $\not\equiv$ for syntactic inequality. E.g., $2+2 \not\equiv 4$.
- We'll consider three kinds of redundancy for parentheses:
  - ***Precedence***: For example, $(p \wedge q) \vee r \equiv p \wedge q \vee r$ because the conjunction operator has higher precedence than the disjunction operator. If we want $p \wedge (q \vee r)$, the parentheses are necessary.
  - ***Left or Right Associativity***: For example, $p \rightarrow q \rightarrow r \equiv p \rightarrow (q \rightarrow r)$ because the implication operator is right associative, so if we want $(p \rightarrow q) \rightarrow r$, then the parentheses are necessary.
  - ***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 since the subtraction operator is not associative, we have $(a-b)-c \not\equiv a-(b-c)$.
- ***Commutativity Not Included:*** We aren't going to take commutativity of operators into account, so $p \wedge q \not\equiv q \wedge p$.
  - Leaving out commutativity gives us two helpful properties: First, it makes "≡" easy to calculate. (It takes $O(n)$ time.) Second, 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 ≡. Since they both equal (evaluate to) 6, we see that syntactically unequal items can stand for the same value.
- 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. For example, we know $(x+(y-z))$ and $((x+y)-z)$ are semantically equivalent[3]; do we want them to be syntactically equivalent?
- The answer will be no, but to justify this, we need to look a bit more at how syntactic and semantic equality are related, so we'll put off the explanation for a bit.

---

[3] Technically, on floating point numbers, they might be different.

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 syntactically unequal but semantically equal.
- ***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., $1+2*3 \equiv (1+(2*3))$, so they stand for the same value. Since "≡ implies =" is true, the contrapositive "≠ implies ≢" is also true. E.g., $2+2 \neq 5$, so $2+2 \not\equiv 5$.
- ***Semantic Equality Does Not Imply Syntactic Equality***: A separate question from "≡ implies =" is the converse: Does = imply ≡? It's easy to find examples that tell us "No": $2+2 \not\equiv 4$, $a+0 \not\equiv a$, and $p \wedge q \not\equiv q \wedge p$.
- Back to mixing * and / (or + and –): We'd like syntactic equality to depend only on syntactic properties, but the equality of $(x+(y-z))$ and $((x+y)-z)$ is a property of the semantics of + and –, so we'll take the answer to be "No".

Parenthesizations

- The ***minimal parenthesization*** of a syntactic item is the one with the fewest parentheses that preserves ≡. (I.e., it is still ≡ to the original.)
- For associative operators, let's omit parentheses inside sequences like $p \wedge q \wedge r$ or $p_1 \wedge (q_1 \vee q_2 \vee q_3)$.
- The ***full parenthesization*** of an item is the one that preserves ≡ and also includes parentheses around each operator expression (i.e., $($operator $p)$ for a unary operator or $(p$ operator $q)$ for a binary operator).
  - We'll omit parentheses around constants, variables, and already-parenthesized expressions; we don't want to be writing things like $(((1)+(((2)*(3)))))$.
  - Technically, the outer parentheses of $(1+(2*3))$ are required, but let's take omitting them to be an ignorable small mistake[4].
  - 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: 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 *.

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

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

States and Satisfaction *[2023-01-11 starts here]*

[2023-01-10: various changes start]

- To talk about the truth of a proposition, we'll use ***states***. In CS terms, our states just model computer memory states: They connect variables and values, and we'll talk about a proposition as being ***satisfied*** by a state if it's true given the state's variables and values.
- In terms of propositions, a state represents one truth table row of possible values for a set of proposition variables. Satisfaction means that a proposition is true for that particular truth table row. E.g., the table above has p and q both false for its first row's state; $p \rightarrow q$ is satisfied in that state but $p \wedge q$ isn't.
- ***Definition***: A (***well-formed***) ***state*** σ is a finite function from proposition letters to truth values. We'll normally represent them as a finite set of pairs (a.k.a. ***bindings***) of a proposition letter and a truth value. Since a state is a function, there can only be one binding for any given variable.
- ***Notation***: Instead of write a binding as a pair (p, T), we'll write it as $p = T$, just for readability. E.g., the state where p and q are both false is $\{p = F, q = F\}$.
- ***Definition***: A set of bindings is ***ill-formed*** as state if includes more than one binding for some variable, or it includes things that aren't bindings of proposition letters to truth values. E.g., $\{p = T, \ p = F\}$ is ill-formed as a state, since it has two bindings for p.
- A couple of remarks: The smallest state is the empty state $\varnothing$ (it has the empty set of bindings). Second, since states are sets of bindings, the order of presentation of bindings doesn't matter: $\{p = F, \ q = T\}$ and $\{q = T, \ p = F\}$ are the same state.
- ***Not***e: It's important for the bindings to involve only proposition letters, not more complicated propositions. E.g., $\{p \vee q = F\}$ is ill-formed. On the other hand, though we might write just T or F for the value, you can use any mathematical description of a value. E.g., "$\{p = T\}$" and "$\{p = \alpha\}$ where α is T if two plus two equals four" describe the same state.

[end of 2023-01-10 changes]

- ***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 $\{p=T,\ q=F\}$. The proposition is ***not satisfied*** (or ***unsatisfied***) in a state if it evaluates to false in that state[5].
- ***Notation***: If σ is a state and p is a proposition, then $\sigma \vDash p$ means that σ satisfies p, and $\sigma \nvDash p$ means σ does not satisfy p. (The $\vDash$ symbol is a "double turnstile", if you haven't run across it before.) If σ is not a well-formed state, then we can't even ask $\sigma \vDash$ or $\nvDash p$.
- ***Examples***: $\{p=T\} \vDash p$, $\{p=F\} \vDash \neg p$, and $\{p=T,\ q=F\} \vDash p \vee \neg q$. For nonsatisfaction, $\{p=T\}$ $\neg p$, $\{p=F\} \nvDash p$, and $\{p=T,\ q=F\} \vDash p \wedge \neg q$. A less-obvious case is $\varnothing \vDash T \wedge (F \rightarrow T)$. 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, then if $\sigma \vDash p$ and $\sigma \subseteq \tau$, then $\tau \vDash p$. Since every state extends the empty set and the empty set satisfies $T \wedge (F \rightarrow T)$, we get that every state satisfies $T \wedge (F \rightarrow T)$.
- Though extra bindings are okay, not having enough bindings can produce an unresolvable situation. For example, we can't say $\varnothing \vDash p \wedge q$ because we can't evaluate $p \wedge q$ in $\varnothing$ and get true. But we also can't say $\varnothing \nvDash p \wedge q$, since we can't evaluate $\mathrm{p} \wedge \mathrm{q}$ and get false. Does this prevent us from making statements like $\mathrm{p} \vee \neg \mathrm{p}$ is satisfied by every state"?
- So ill-formedness prevents us from using something as a state. There's another kind of error to look at too. Say σ is well-formed, so that questions like $\sigma \vDash 2+2=4$ have an answer. If σ maps a variable to something other than T or F, then even though σ is well-formed, it's not useable for propositions that include that variable.
- ***Definition***: A state is ***proper*** for a proposition p if it includes bindings for all the variables of p and the state binds each variable to a value of the correct type (T or F for now; it gets more complicated once we include data values). Note σ could be proper for p_1 but not for p_2; it depends on what variables σ, p_1, and p_2 use.
- ***Asking About All/Some/etc. States***: When we say things like "p is satisfied in all σ" or "Let σ be a state that doesn't satisfy p", we'll quietly assume that we're only talking about the states proper for p. So we can say "$\sigma \vDash p \vee \neg p$ for all σ" even though $p \vee \neg p$ certainly isn't satisfied in $\varnothing$ or in $\{q=T\}$, as two 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.
    - σ is well-formed for p, p evaluates to T under σ, so $\sigma \vDash p$ and $\sigma \nvDash \neg p$.
    - σ is well-formed for p, p evaluates to F under σ, so $\sigma \nvDash p$ and $\sigma \vDash \neg p$.

---

[5] When we get to runtime errors, there will be a third possibility. E.g., asking "Does $\{x=0\}$ satisfy $x/x=1$?" yields an error.

- Note since we don't have to worry about runtime errors, if σ is well-formed for p, then the statements $\sigma \models \neg p$ and $\sigma \not\models p$ have the same answer. Similarly, $\sigma \models p$ and $\sigma \not\models \neg p$ have the same answer.

Validity, Tautologies, and Logical Equivalence

- Now that we have the notion of a proposition being true in a given truth table column, we can go further and talk about a proposition being true in every truth table column.
- ***Definition***: p is ***valid*** (notation: $\models p$) if $\sigma \models p$ for every σ.
- ***Examples***: $\models T$, $\models \neg F$ (the two simplest examples), $\models p \vee \neg p$, $\models (p \to q) \leftrightarrow (\neg p \vee q)$.
- ***Definition:*** p is ***invalid*** (i.e., not valid), written $\not\models p$, if $\sigma \not\models p$ for some σ. (And recall that $\sigma \not\models p$ and $\sigma \models \neg p$ mean the same thing (until we get to runtime errors).
- ***Examples***: $\not\models p \vee q$, since $\{p=F,\ q=F\} \not\models p \vee q$. Another example: $\not\models p \wedge \neg p$, since $p \wedge \neg p$ is false for $\{p=T\}$ and $\{p=F\}$ both. Note that no state satisfies $p \wedge \neg p$ but some states do satisfy $p \vee q$. (E.g., $\{p=T,\ q=F\} \models p \vee q$.)
- One way to categorize propositions is through their validity.
- ***Definition***: p is a ***tautology*** if $\models p$, a ***contradiction*** if $\models \neg p$, and a ***contingency*** if $\not\models p$ and $\not\models \neg p$ (simultaneously). Another way to say this is that a tautology has a truth table column with only T values, a contradiction has a truth table column with only F values, and a contingency has a column that includes at least one T and at least one 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 $\models p \leftrightarrow q$. I.e., in a truth table, the column for p matches the one for q. It's easy to show that $\Leftrightarrow$ is transitive: If $p_1 \Leftrightarrow p_2$ and $p_2 \Leftrightarrow p_3$, then the columns for p_1 and p_2 match, the columns for p_2 and p_3 match, so the columns for so p_1 and p_3 match.
- ***Notation***: "$\Leftrightarrow$" is often pronounced "if and only if", so people often write "p iff q" for what we're writing as $p \Leftrightarrow q$.
- We most often use $\Leftrightarrow$ to talk about how a sequence of step-by-step transitions produces propositions that are all logically equivalent. E.g., $p \to q \Leftrightarrow \neg p \vee q \Leftrightarrow q \vee \neg p \Leftrightarrow \neg\neg q \vee \neg p \Leftrightarrow \neg q \to p$.
- It turns out that $\Leftrightarrow$ on propositions is like = on arithmetic expressions.
  - "$(x+1)^2 = (x^2+2x+1)$" means we can substitute one for the other in any semantic context.

- Similarly $p \Leftrightarrow q$, then in any semantic context, we can always substitute p for q or vice versa.
- Note in both cases, the context has to be semantic. E.g., the length of the expression $(x+1)^2$ is 6, but we can't replace $(x+1)^2$ with (x^2+2x+1) in that statement.

- Remember: $\Leftrightarrow$ and $\leftrightarrow$ are similar but not identical.
  - The $\leftrightarrow$ symbol is a syntactic operator and can appear in propositions. (More generally, we can use it in boolean expressions: In C, $\leftrightarrow$ is written ==.) On the other hand, $\Leftrightarrow$ is a semantic operator: It doesn't appear in propositions because it describes a semantic property.
  - The $\Leftrightarrow$ operation is transitive: if $T \Leftrightarrow p_1 \Leftrightarrow p_2 \Leftrightarrow \ldots$ etc., then all the p's evaluate to T. The $\leftrightarrow$ operation, however, is not transitive. For example, $(T \leftrightarrow (F \leftrightarrow F))$ evaluates to T but T, F, and F are certainly not all logically equivalent.
  - "$(x+1)^2+3 = (x^2+2x+1)+3 = x^2+2x+4$"? 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 equivalent to its contrapositive; similarly, the converse of an implication is equivalent 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 the negation and the inverse of $p \rightarrow q$ 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: The exclusive or of p and q is true when one of them is true and the other one is false.
  - The exclusive or is the negation of the biconditional; momentarily writing $p \oplus q$ for the exclusive or of p and q,
    - I.e., $p \oplus q \Leftrightarrow (p \wedge \neg q) \vee (\neg p \wedge q) \Leftrightarrow \neg(p \leftrightarrow q)$
  - The usual "inclusive" or allows one or both of p and q to be true:
    - I.e., $(p \vee q) \Leftrightarrow (\neg p \wedge q) \vee (p \wedge \neg q) \vee (p \wedge q)$.
    - Also, $p \oplus q \Leftrightarrow (p \vee q) \wedge \neg(p \wedge q)$.