

CS 536 Notes: About Science of Programming

Lecture 1, Mon, Jan 9, 2012

A. Why

- Course guidelines are important.
- Active learning is the style we'll be using in the class.
- 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 today, 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 are at <http://www.cs.iit.edu/~cs536>
- The home page includes news and the calendar of lectures, coursework, and tests.
- There's a Course Plan page that discusses the textbook, the course topics, course structure, and grading, collaboration, disability, and ethics policies.

D. Active Learning

- Education research shows that students learn better when they are active participants in class, compared to passive listeners of lectures.
- This active kind of learning includes obvious things like answering questions in class, but it also includes team activities and quick feedback to the instructor.
- You'll need to put more effort into class, but you'll learn more/better/faster.

E. So What Is Science of Programming Anyway?

- Science of Programming is about **program verification**.
- Program verification aims to get reliable programs by proving properties about the program.
 - 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.
- Distinguish program verification from program testing.
 - In testing, we run a program and verify that it behaves correctly.

F. Neither Reasoning or Testing is Better Than the Other

- In real life, we need both testing and reasoning; neither is always better than the other.
 - Testing is a reality check on reasoning: We want to test because our reasoning might be wrong.
 - Reasoning needs to be done to find good test cases (and avoid bad ones).
 - Usually there exist a large/infinite number of test cases; we want to concentrate on important ones.
- Simple example of using reasoning to generate test cases:
 - Take the statement `if (x >= 0) z := z+x;` (“:=” means assignment)
 - Say our specification is “If $z \geq c$ before the statement, then $z > c$ after the statement”. Write this for now as


```
/* z >= c */ if (x >= 0) z := z+x; else ++z; /* z > c */
```
 - What are good test cases? x equal to 0? 1? -1? How about 2, 3, 4, 5, ?

G. Typechecking as a Kind of Program Verification

- Typechecking is an example of program verification.
- In typechecking, we look at a program before we run it and reason about whether it uses types correctly.
 - E.g., declare variables x and y to be integers. Then $x+1$ is an integer, so $y := x+1$ is type-correct, and $x := y/x$ is type-correct, etc.
 - Note y/x might still cause a runtime error, but it’s not a type error.
- A typechecker is a mechanical theorem prover for judgements of the form “this variable or expression has type T ” and “This operation is type-correct.”
 - A **strong typechecker** produces proofs that provide complete evidence for type safety. A **weak typechecker** produces proofs that provide only partial evidence for type safety.
 - E.g. typecheckers for C are weak because they have to assume you know what you’re doing when you cast a pointer.
 - Typecheckers 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.)

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

- In program verification, we often reason simultaneously about many states of memory.
- E.g. take `/* x >= 0 */ x := x + 1 /* x > 0 */`
 - Actual program execution will involve a memory state where x has some specific value, like 17.
 - When reasoning about the program, we work simultaneously with $x = 0$, $x = 1$, $x = 2$, etc. Each value involves one possible memory state.
 - We write predicates like “ $x \geq 0$ ” and “ $x > 0$ ” to stand for sets of memory states.

- Reasoning about this program stands for figuring out all the possible execution paths that the program might take.
- Instead of actually executing the program, we're simulating its execution ahead of time, on sets of states.
- The sets of states are represented using logical predicates, so we have to use rules of logic (plus rules for how programs execute) to do this simulated execution.
- **That's what program verification is:** 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: Syntax of Propositional Logic

- **Propositional logic** is logic over **proposition variables** (i.e., Boolean variables).
 - **Notation:** Typically use p, q, \dots for propositions, T, F for the constants true and false.
 - In propositional logic we study the logical connectives: And (\wedge), Or (\vee), Not (\neg), Implication (\rightarrow), and Equivalence (\leftrightarrow). These are also known as Conjunction (\wedge), Disjunction (\vee), Negation (\neg), Conditional (\rightarrow), and Biconditional (\leftrightarrow).
- **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 $p \vee q$.
 - $p \rightarrow q$ is the **implication** or **conditional** of p and q .
 - p is the **antecedent** or **hypothesis** of $p \rightarrow q$.
 - q is the **consequent** or **conclusion** of $p \rightarrow q$.
 - Implications
 - The **contrapositive** of $p \rightarrow q$ is $\neg q \rightarrow \neg p$; its **converse** is $q \rightarrow p$; its **inverse** is $\neg p \rightarrow \neg q$.
 - " p is **sufficient for** q " means $p \rightarrow q$; " p is **necessary for** q " means $q \rightarrow p$.
 - " p **only if** q " means $p \rightarrow q$; " p **if** q " means $q \rightarrow p$; "if p then q " means $p \rightarrow q$.
 - $p \leftrightarrow q$ is the **equivalence** or **biconditional** of p and q .
 - p is the **antecedent** or **hypothesis** of $p \leftrightarrow q$.
 - q is the **consequent** or **conclusion** of $p \leftrightarrow q$.

- **Precedences** (strong to weak): $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
 - E.g., all parentheses are optional in $((\neg p) \wedge q) \vee r \rightarrow s \leftrightarrow t$.
 - I.e., it's the same as $\neg p \wedge q \vee r \rightarrow s \leftrightarrow t$.
- **Associativity:**
 - \wedge and \vee are associative so it sort of doesn't matter; let's use left associativity.
 - \rightarrow is right associative: $p \rightarrow q \rightarrow r$ is short for $(p \rightarrow (q \rightarrow r))$.
 - Note $(p \rightarrow (q \rightarrow r))$ is different from $(p \rightarrow q) \wedge (q \rightarrow r)$.
 - Treat \leftrightarrow as also being right associative: $p \leftrightarrow q \leftrightarrow r$ is short for $(p \leftrightarrow (q \leftrightarrow r))$
 - But we'll pretty much never use $(p \leftrightarrow (q \leftrightarrow r))$
 - It's different from $(p \leftrightarrow q) \wedge (q \leftrightarrow r)$.
- **Syntactic Equality**
 - Syntactic equality (written \equiv) means equality as text, so $2+2$ is not \equiv to 4 (written $2+2 \neq 4$). We do ignore redundant parentheses when checking for syntactic equality, so $2+2 \equiv ((2)+(2))$.
 - The precedence and associativity rules tell us $p \wedge q \vee r \equiv (p \wedge q) \vee r \neq p \wedge (q \vee r)$.
 - The minimal parenthesization of a syntactic item is the one with the fewest parentheses that is still \equiv to the original. E.g., the minimal parenthesization of $(p \wedge q) \vee r$ is $p \wedge q \vee r$. Also, $p \wedge (q \vee r)$ is its own minimal parenthesization.

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 ”.
 - Treat true, false, \rightarrow as like 1, 0, and \leq , so $F \rightarrow T$ is like $0 \leq 1$.
- Kinds of propositions
 - **Tautology:** Has a truth table with all rows true.
 - **Contradiction:** Has a truth table with all rows false.
 - **Contingency:** Has a truth table with at least 1 row true and at least 1 row false.