Welcome to CS 245

Instructors: Andrew Malton (SE 112), Prabhakar Ragde (CS 245).

Web page:
`http://www.student.cs.uwaterloo.ca/~cs245`

Read everything on the Web page carefully as soon as possible, especially the academic offenses page.

Newsgroup: `uw.cs.cs245`

Read the newsgroup at least daily for fast-breaking items and discussions of common interest.

Overview of course

This course is about formal mathematical logic and its application to computer science.

In your previous CS and math courses, you have been exposed to some aspects of logic, in varying degrees of formality.

Math 135 introduced you to varying styles of mathematical proof using the topic of elementary number theory.

Some “Math 135”-style examples

Thm: Every even natural number is the sum of two odd natural numbers whose difference is at most 2.

Proof: An even natural number is of the form $4k + 2$ or $4k + 4$, for $k \geq 0$. If it is of the form $4k + 2$, it can be expressed as $(2k + 1) + (2k + 1)$, and if it is of the form $4k + 4$, it can be expressed as $(2k + 1) + (2k + 3)$. $\square$

This is an example of what we will call a mathematical proof. Our goal is to formalize the notions of proof and truth, and to explore the implications of such formalizations.

Note that in this example, we have used the Math 135 definition of the natural numbers which begins at 1, excluding 0. The textbook for this course also implicitly assumes this, in its discussion of induction.

However, when creating a precise and formal definition of the natural numbers (as we often have to do as computer scientists, and as we will do later in this course), it is more convenient and useful to include 0 as a natural number.

As a compromise, we will use the Math 135 definition in mathematical discussion, and the CS-style definition in formalization.

We showed a short mathematical proof of “Every even natural number is the sum of two odd natural numbers whose difference is at most 2”.

There are statements of this form that we believe true but do not know how to prove, such as “Every even natural number greater than 2 is the sum of two prime numbers”.

There are also statements of this form that are false, such as “Every odd natural number is the sum of two even natural numbers.”

In fact, this is not true of any odd natural number, but we need only provide a single counterexample. 1 is an odd natural number that is not the sum of two even natural numbers. If $1 = e + f$, for e, f even, then $e \geq 2$, $f \geq 2$, and $1 \geq 4$, a contradiction.

We are clearly working in a system with rules, though most of us learned the rules gradually, through examples, and would have a hard time listing exactly what is permitted and what is forbidden.

The history of logic goes back more than 2500 years to the ancient Greeks, but these rules were not made precise until the late 1800s and early 1900s.

The tools of mathematics that we have learned, such as calculus and algebra, provide us with ways of solving problems and understanding situations through symbolic and algebraic manipulation. We can apply the same notions to the idea of proof.

The mathematical statements we’ve looked at were about infinite sets (e.g. all even natural numbers), but the proof itself was of fixed, finite length.

Computer programs also are of fixed, finite size, but can operate on unbounded data.

```
(define (sum-list lst)
  (cond
    [(empty? lst) 0]
    [else (+ (first lst) (sum-list (rest lst)))]))
```

You have already used logic in what you know of computer science so far.

You used it in an informal sense when writing a program, reasoning about what language constructs can be applied to solve a problem.

You have also used logical constructs (e.g. if/cond statements, Boolean variables) in your programs.

You used logic in a slightly more formal sense to reason about the correctness of your programs, because you usually could not test them on every possible set of data.

Logic is more than a minor part of the CS toolkit. As we will see, the early foundations of CS spring from the attempt to formalize mathematical logic.

A proof is supposed to be easy to verify. Though the formalization of logic predates computers, one obvious definition of “easy” is “can be checked by a program or by other mechanical means, without the need for human interpretation”.

This naturally leads us to the question of whether proofs can be found by computers, and whether a computer program can be proved correct.

Syntax and semantics

The ideas of syntax and semantics are central to computer science.

A computer program is just a sequence of symbols. There are syntactic rules that determine whether a program is well-formed or not (e.g. parentheses must balance).

The semantics of a program (its meaning or interpretation) describe what happens when that program is run. Usually some additional context is needed (e.g. the data on which the program is to be run). This is also true for some things which are not programs (e.g. Web pages in HTML, database queries in SQL).

Formalizing logic

We will define a logical statement as a sequence of symbols that obey certain syntactic rules.

A proof of a logical statement will be defined as a sequence of syntactic productions of logical statements, resulting in the statement we wish to prove.

We will thus define a formal notion of “proof” without any attached semantics or meaning. It will just involve manipulation of symbols.

Independently, we will define semantics for logical statements, which is a way of interpreting them to determine if they are true or false. Thus we will have a notion of “truth” that is separate from the notion of “proof”.

Our goal will be to show that these two notions coincide: that every statement we can prove is true, and vice-versa. This justifies the utility of proofs and the benefits of looking for them.

We will start with a small system (incapable of expressing the theorems we looked at) but one which is familiar to you from Math 135. Once we show that provable equals true for this system, we will expand it.

Throughout, we will be working in the style of mathematical proof you're used to, even as we attempt to formalize it in stages.

Ideally, we would like to reach the point where proofs about aspects of our formal systems can be expressed in those systems themselves. This is unfortunately not always possible, and we will briefly examine the reasons.

Our focus will be on connections to CS and applications in CS, finishing up with modules on how to formally reason about programs and how to use the vocabulary of logic to describe and model systems.

This course extends ideas you may have seen briefly in Math 135, 136, 137, 138, CS 134, 135, 136, 240, and 251.

It will be relevant to ideas you will see in CS 240 and 251 (if you have not taken them), CS 341, 360, 365, 442, 445, 446, 447 (and the SE equivalents), 462, 466, 480, 486, and just about any CS course where you have to either write programs or reason about algorithms.

Some of these courses may not directly depend on CS 245, but the exposure to concepts and practice in techniques here will strengthen your ability to deal with these courses. It will also help in just about any other Math Faculty course you take. If you really like the notions presented here, you might consider taking PMath 432, which goes further than we can (in ways to be alluded to later on in the course).

Course logistics

The two lecture sections (one CS, one SE) will be sharing slides, assignments, and exams. The slide modules are not divided into individual lectures, so the sections may be slightly out-of-sync.

Assignments (six to eight in total) will be worth roughly 25%, the midterm 25%, and the final exam 50%.

One or more graduate student TAs will be delivering weekly tutorials, with some reinforcement of lecture material and some practice in solving problems.

The required textbook is "Logic in computer science: modelling and reasoning about systems" (second edition), by Michael Huth and Mark Ryan. This text was first used in Winter 2006, though parts had been used in earlier offerings. Other textbooks have been used in this course (and may be used in the future).

We will use Chapters 1, 2, and 4 (out of 6). If time permits, we will look at some of the other relevant topics in the book.

This offering may be different from previous offerings of the course, though it follows the same broad outline.