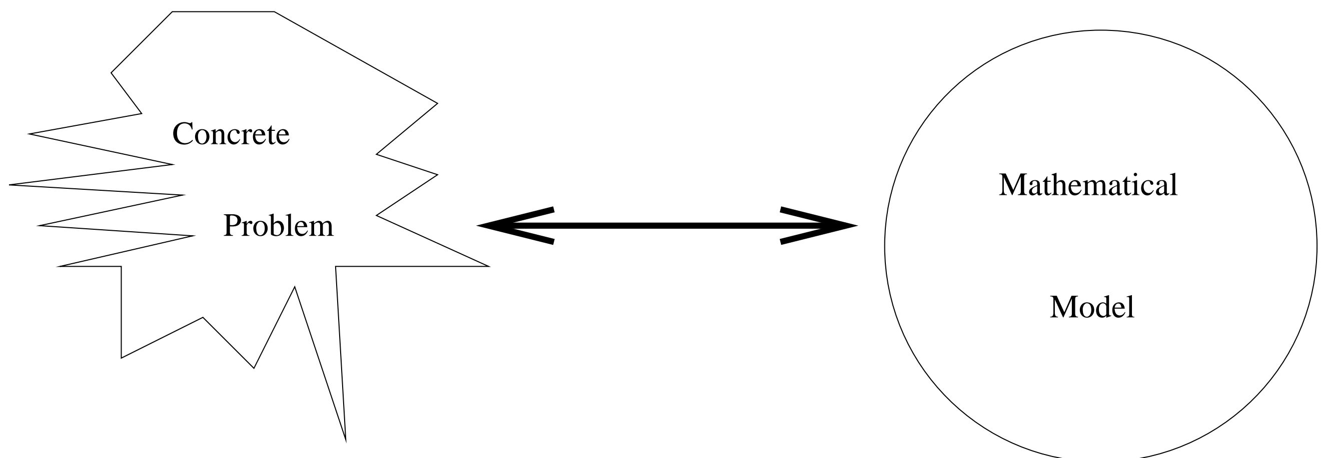
What is the subject matter of this course?

- **Computability**: What can be computed in principle?
- **Logic**: How can we express statements and proofs?
- **Complexity**: What can be computed in practice?



The mathematical method is to form abstractions that capture some important aspects of a real-world phenomenon, then operate on those abstractions using *formal definition*, *proof*, and *mathematical problem-solving*.

Our real-world target is *digital computation*. Our first abstraction is say that we have an *input*, a collection of bits, and want an *output*, often a single bit. The key questions are then:

- How is the input organized?
- What computational operations are allowed?
- Do we have internal memory, and how much?

An answer to these questions gives us a formal model of computation.

Some Formal Models of Computation:

- **Boolean:** Input bits are undifferentiated, we can use boolean operations (AND, OR, NOT) and store the results. We also express properties of the input using propositional logic.
- **Formal Language Theory:** The input bits are arranged in a *string* of letters. We work with one letter at a time. Defining the internal memory gives us models such as the *finite-state machine*, *pushdown automaton*, or *Turing machine*.
- **First-Order Logic:** The input bits form a *structure* made up of *relations*. We express properties of the input using first-order logic (e.g., quantifiers $\forall$ and $\exists$).
- **Recursive Function Theory:** Input bits are formed into non-negative integers, on which we define functions starting from arithmetic operations.
- **Abstract RAM:** The input and internal memory are formed into words in registers, and operations mimic those of real-world sequential computers.

Course Requirements

Texts: available at Jeffery Amherst College Store

[BE:] Jon Barwise and John Etchemendy, *Language, Proof, and Logic* (required, *new copy* needed for homework)

[P]: Christos Papadimitriou, *Computational Complexity* (recommended)

[S]: Michael Sipser, *Introduction to the Theory of Computation* (useful, lower-level reference)

Prerequisites: Mathematical maturity: reason abstractly, understand and write proofs, use big-O notation. CMPSCI 250 needed; CMPSCI 311, 401 helpful. Today's material is a good overview of the sort of stuff we will do, next lecture we start right in on formal language theory.

Graded Work:

- Eight problem sets (35% of grade)
- Midterm (30% of grade), in-class Monday 29 March 2004
- Final (35% of grade)

Cooperation: Students should talk to each other and help each other; but **write up solutions on your own, in your own words.** Sharing or copying a solution could result in a grade of F for the course, even on the first offense. If a significant part of one of your solutions is due to someone else, or something you've read then **you must acknowledge your source!** When the grader then looks at the source, it should be clear from your writeup that you've understood anything you've taken from it. A good heuristic is not to have the source in front of you when you write up.

Electronic Grading: Most of the homework problems from [BE] will be graded electronically. Your interaction with the software provided will result in a file that you send in, and each student must send in a separate file from a separate interaction with the software (they check).

Mathematical Sophistication

- *How to Read and Do Proofs, Second Edition* by Daniel Solow, 1990, John Wiley and Sons.

Review of Regular and Context-Free Languages

- Hopcroft, Motwani, and Jeffrey D. Ullman, *Introduction to Automata Theory, Languages, and Computation,* 2001: Chapters 1–6.
- Lewis and Papadimitriou, *Elements of the Theory of Computation,* 1998: Chapters 1–3.
- Sipser, *Introduction to the Theory of Computation*, 1997: Chapters 1 – 2.

NP Completeness

- Garey and Johnson, *Computers and Intractability,* 1979.

Descriptive Complexity

- Immerman, *Descriptive Complexity,* 1999.

Syllabus will be up soon on the course web site:

- `http://www.cs.umass.edu/~barring/cs601`

There is a pointer there to the Spring 2003 web site, and the syllabus there will be close to what we do here.

Rough guide:

- Formal Languages and Computability (9 lectures)
- Propositional and First-Order Logic (6 lectures)
- Complexity Theory (12 lectures)

- How is the input organized?
- What computational operations are allowed?
- Do we have internal memory, and how much?

An answer to these questions gives us a formal model of computation.

Some Formal Models of Computation:

- **Boolean:**
- **Formal Language Theory:** (starting next lecture)
- **First-Order Logic:**
- **Recursive Function Theory:**
- **Abstract RAM:** (as in an algorithms course)

Boolean Computation:

Data Type: Booleans, true or false, 0 or 1

Operations: AND, OR (inclusive), NOT, can define more

Variables: Inputs $x_1, \ldots, x_n$, others $y_1, \ldots, y_m$

Expressions: $(x_1 \vee \neg x_2) \wedge x_3$, etc.

Straight-Line Program:

```
y_1 = NOT x_1;
y_2 = y_1 OR x_2;
y_3 = x_3 AND x_4;
y_4 = y_2 AND y_3;
y_5 = y_2 OR x_1;
return NOT y_5;
```

A straight-line program is equivalent to a *boolean circuit*, about which more later. Each line of the program (gate of the circuit) has a truth value that is a *function of the input variables*. Given a function of the inputs (a *property*), we will consider among other things *how long a program* is needed to compute it in this way.

Boolean Logic:

Chapters 1-8 of [BE] deal with propositional or boolean logic. The main topics are:

- *Syntax* and *semantics* for boolean expressions
- Rules for proving that an expression is a *tautology* or that one expression *follows logically* from another
- A formal proof system encapsulating these rules and implemented as interactive software, included with the book

The basics of boolean logic should be review, but formal proof may not be. You will be expected to learn the proof system from the book and software, and there will be exercises on it in HW#2. At that point we will consider two important properties of the system:

- Soundness: everything provable is true
- Completeness: everything true is provable

Here “everything” refers only to statements that can be made within the system.

First-Order Logic:

We can think of a sequence of boolean variables $x_0, \ldots, x_{n-1}$ as a single object X, a *function* from the set $\{0, \ldots, n-1\}$ to $\{0, 1\}$. Thus for any number i in the range, $X(i)$ is the boolean variable we used to call x_i.

X is also called a *unary relation*, a *property* that each number in the range either has or doesn't have. We can also have *binary*, *ternary*, or k-ary relations.

For example, in a directed graph where the vertices are named $\{0, \ldots, n-1\}$ we have a binary relation E, where the boolean $E(i, j)$ is 1 iff there is an edge from i to j.

More First-Order Logic:

A *first-order structure*, roughly speaking, is a universe of base objects and a set of relations over them. If one of these relations is the *input*, we can use *first-order logic* to define properties of the input.

For example the first-order sentence

$$\forall x : \exists y : E(x, y)$$

is true for the graph with edge relation E iff every vertex has out-degree at least one.

In Chapters 9-13 of [BE] we'll see syntax and semantics for first-order logic, a set of proof rules for it, and software that lets you play with structures consisting of sets of blocks of different types on a grid. We'll prove (using later chapters of [BE]) that this proof system is also sound and complete.

Recursive Function Theory:

Bits can be used to represent *numbers* (by which I generally mean *non-negative integers* in binary. General computations may be coded as functions from numbers (or tuples of numbers) to numbers.

Kleene's *recursive function theory* defines a set of *partial recursive functions* by induction. There are *base functions* and rules for creating new functions from old ones.

It turns out that "partial recursive" corresponds to "algorithmically computable". (This is a theorem, not a definition.)

Primitive Recursion and Bloop

An important subset of the partial recursive functions is the *primitive recursive functions*. We will define these in terms of a simple programming language called Bloop. (My Bloop is based on the language of the same name in Hofstadter's *Gödel, Escher, Bach* but has a different syntax.

Variables represent numbers (non-negative integers). They are declared and initialized to 0 when they first appear.

Statements of Bloop consist of:

- assignment statements `x = ...`
- the increment operator `++`
- function declarations and calls (call-by-value, no recursion)
- bounded loops `for (x) B`, where `x` is a variable and `B` is a block of code in which `x` is not modified. This code executes `B` exactly `x` times.

Bloop Examples:

```
declare one() {
   x++;
   return x;}

declare add(x,y) {
   z = x;
   for (y) z++;
   return z;}

declare  mult(x,y) {
   for (x) add(z,y);
   return z;}
```

Enriching Bloop:

- Use arithmetic expressions like variables
- Simulate if-then, `if (x) B` means:

```
y = one();
for (x) {
   for (y) B;
   y = 0;}
```

  This executes `B` once if `x` is positive and otherwise does nothing.

A function is *primitive recursive* iff it can be implemented by a Bloop program.

Relations Among The Models:

Let's look at a single problem. Given n input bits, we want to know whether *exactly two* of them are ones. This question can be posed in each of our models:

- **Boolean:** There are various ways to build an SLP or circuit, which we'll explore on HW#1.
- **Finite-State Machine:** Sweep the input string left-to-right, remembering whether we've seen zero, one, two, or more than two ones.
- **First-Order Logic:**

$$\exists x : \exists y : \neg(x = y) \wedge \forall z : I(z) \leftrightarrow (z = x \vee z = y)$$

- **Numerical Input:** Is the input the sum of two distinct powers of two? On HW#1 you'll write a Bloop program to decide this.
- **Abstract RAM:** The problem probably defaults to one of the others once we decide on our data representation.