Instructor: Hudson Turner  
Teaching Assistant: Rashmi Kankeria


This course introduces elements of the theory of computation, an active research area involving the formulation of precise questions and answers concerning

- what is computable
- by what means
- in what amount of space and time.

Remarkably, such work does not depend essentially on any particular technology or programming language. Instead, machines and computations are expressed and studied as mathematical objects.

Fundamental ideas and results predate modern digital computers, and remain important.

Course goals

1. Familiarity with some fundamental results in the theory of computation.

These are lovely, and are part of the core of computer science.

Theory of computation is mathematical. We'll need to understand and use mathematical definitions — this involves stating precise claims about them, and proving those claims.

So a second course goal is:

2. More experience working with precise mathematical statements!

Mathematical sophistication is useful in almost all (all?) areas of computer science.

Moreover, the precision and even some of the methodology required in work with mathematical definitions is rather similar to that required in software design and development.

Course organization

Details in syllabus.

Almost weekly homework. You can't learn this stuff without doing it!!

Homework worth about 15% of course grade.

Most of the background material you'll need can be found in the course notes — find at [www.d.umn.edu/~hudson/4811](http://www.d.umn.edu/~hudson/4811).

Use textbook primarily for enrichment and additional detail.

Again, you learn this stuff by doing it: most lectures will begin with a discussion period in which we review earlier material.

I'll bring questions for you, and will rely on your participation. It won't work without your help.

Many lectures will (also) begin with a pop quiz. No makeup on these, but lowest three for each student will be dropped.

Approximately 15% of course grade is based on these pop quizzes.

About 80% of course grade based on three exams plus a final. All are open-book/open-note. (Emphasis on accurate understanding!)

For next time

We'll start with a brief overview of the subject of the course: theory of computation.

We will then begin a rather exhaustive review of the material in chapters 1 and 2 (excluding the treatment of logic in Section 1.2).

Sets, functions, relations, languages, and proofs!

The first exam will cover this material. (You'll need to be comfortable with it in order to keep up later in the course.)

For next time, read Introduction and Sections 1.1, 1.3, 1.4.

Class notes, homework assignments, etc. will be available via the web page at [www.d.umn.edu/~hudson/4811](http://www.d.umn.edu/~hudson/4811)
in the form of postscript files. If you need help viewing and printing these files, ask fellow students or the TA.
Languages
(as understood in the study of theory of computation)

A unifying idea in theory of computation: what we study are "languages."

What is a language?

- Start with set $\Sigma$ of symbols.
- A string over $\Sigma$ is a finite sequence of symbols from $\Sigma$ (possibly the sequence of length zero, that is, the empty string, written $\Lambda$).
- By $\Sigma^*$ we denote the set of all strings over $\Sigma$.
- A language over $\Sigma$ is simply a subset of $\Sigma^*$.

Notice that computer programs, along with their inputs and outputs, are in fact strings (essentially strings over the alphabet $\{0, 1\}$).

We'll study a few main classes of languages...

A second characterization of regular languages:

Regular languages are "recognized" by regular expressions.

Example $\{0 + 1\}$ is a regular expression.
It generates two strings over $\{0, 1\}$: 00, 11.

Example $(0 + 1)(0 + 1)$ is a regular expression.
It generates four strings over $\{0, 1\}$: 00, 01, 10, 11.

Example $(0 + 1)(0 + 1)(0 + 1)$ is a regular expression.

How many different strings does it generate?

Example $0^*$ is a regular expression.
Generates all strings over $\{0, 1\}$ that consist of some number of 0's (possibly none) followed by a 1.

Example $(0 + 01 + 10 + 11)^*$
What language over $\{0, 1\}$ does this regular expression generate?

Regular languages

We'll carefully study two rather different characterizations of the class of regular languages, and eventually show that they are indeed equivalent.

First, regular languages are "recognized" by finite state automata.

Rough description of a finite state automaton:

- finite alphabet $\Sigma$ (finite set of symbols)
- finite set of states, including an "initial state"
- input is a string over $\Sigma$
- at each step, read next symbol in string
- change state (based on input symbol and current state)
- string is "accepted" if machine is in an "accepting state" after reading last symbol in input string

We'll soon make this description precise, so we can really work with it!

(We'll also be able to draw nice pictures of these.)

Some languages are not regular

Example
The language pal of palindromes over $\{0, 1\}$ is not regular.

(A palindrome is a string that is the same read forward or backward.)

Intuitive reason pal is not regular. Each string over $\{0, 1\}$ has a finite length. On the other hand, there is no upper bound on the length of strings over $\{0, 1\}$. Therefore, in order for a finite automaton $M$ to decide whether a given string $x$ is a palindrome, $M$ would need an arbitrarily large number of states — in order to "remember" the (arbitrarily long) first half of $x$.

Also, most programming languages are not regular.
But the set of all identifiers in a programming language is a regular language.

This fact is important in the design of programming language compilers.
Context-free languages

Context-free languages are recognized by pushdown automata.

Rough description:

- finite set of states, unbounded stack
- again, input is a string
- at each step,
  - read next symbol in string
  - change state and manipulate stack based on:
    - input symbol,
    - current state, and
    - symbol at top of stack
- machine starts in a distinguished “initial state” with the stack “empty”
- again, an input string is “accepted” if the machine is in an “accepting state” after reading the last symbol in the input string

Recall that a primary limitation of finite state automata is that they have only a finite set of states for “remembering.”

Pushdown automata also have only a finite set of states, but, in addition, they have an unbounded stack.

Context-free languages are generated by context-free grammars (CFG’s).

Example: A CFG for the language of palindromes over \{0, 1\}.

\[ S \rightarrow 0 \mid 1 \mid A \mid 0S0 \mid 1S1 \]
\[ S \Rightarrow 0S0 \Rightarrow 01S10 \Rightarrow 0110 \]

Programming languages are typically context-free languages (or, more precisely, they are very nearly context-free).

Hence, the theory of context-free languages underlies compilers and interpreters of programming languages.

Much of the syntax of natural language is context-free.

Some languages are not context-free…

Recursive languages, recursively enumerable languages

Generated and recognized by Turing machines (TM’s).

[Subtle distinction between the two classes of languages.]

Rough description of TM:

- finite set of states (including a distinguished initial state), unbounded tape, finite set of tape symbols
- (for recognition) input is the string that is on the tape at the start of the computation
- at each step,
  - read tape symbol at current position on tape
  - based on current tape symbol and current state:
    - change state,
    - change tape symbol at current position, and possibly move left or right
- input string is “accepted” if TM reaches the “accepting state” during the computation

Church-Turing thesis:

Whatever can be computed can be computed by a TM.