

Lecture 1: Introduction

Prof. David L. Dill

Department of Computer Science

Outline

- What is this course about? What is it good for?
- Course administration.
- Basic concepts: Strings and languages.
- Guidelines for proofs.

Reading: Chapter 1 of the textbook

Representing Sets

Suppose you, as a programmer, need to represent a small, finite, set.

- What does “represent” mean?

Answer: You can answer questions about it, perform computations on it.

Simple common question (membership): Is $x \in S$?

Other questions:

Is $S = \emptyset$?

Is $S \cap T = \emptyset$? Etc.

Ok, suppose you want to a *finite representation* of *infinite* sets. How do you do it?

One view of formal language theory

Automata and complexity theory is concerned with properties of *formal languages*. (A “formal language” is just an impressive name for a set of strings.)

In formal language, automata, and complexity theory, a *language* is just a set of strings.

Like many mathematical definitions, this leaves behind most of what we think of as “languages,” but can be made precise. And it leads to very profound results.

Formal languages are only interesting when they are infinite.

I claim: Something can be represented in a computer iff it can be written as a string (e.g., integers, floating point numbers, graph structures, etc.)

Self-reference

Formal language theory becomes almost surrealistic because of “self reference.”

Finite representation = a string, so we can talk about *languages of other (finitely representable) languages*.

E.g., we can talk about the language of all Turing machines that halt, and then wonder about the properties of *that* language.

This leads to special proof techniques, and can tie your neurons into knots.

What is a representation?

Suppose you have representation that can be stored in a computer.

Can all sets be represented?

No: Compare the number of possible strings (which is countable) with the number of sets of strings (uncountable).

A particular set that cannot be represented is the set of all irrational numbers – there are “too many” irrational numbers.

This raises profound questions: Which sets can be represented on a computer and which can't?

Questions from formal language theory

What (infinite) sets are representable?

What can a computer do with the representations, in theory?

What *cannot* be done with the representations, in theory?

What problems are easy, hard, or impossible to solve computationally?

Another view of formal language theory

For practical purpose, a language is the same thing as a Boolean function. Such a function is also called a *property* or a *predicate*.

For example, the predicate $\text{even}(x)$, which returns “true” iff x is (string representation of) an even number, can be considered to represent the set of even numbers (think of it as an “implicit set lookup”).

So, if we can answer questions about languages, we are also answering questions about properties of objects.

Example: A program that can test whether a Java program halts on all inputs tests membership in the language of all Java programs that halt on all inputs.

“Some languages cannot be represented” = “Some predicates cannot be effectively computed”

“What sets can be represented” = “What can computers do?”

Applications of Automata Theory

This is probably the most theoretical course in the undergraduate curriculum.

But a lot of this material has very important practical applications.

New applications are being discovered all the time – even for things that were originally invented to answer purely theoretical questions.

Value of the course

In 20 years, computers and programming will be vastly different. But this material will be very much the same – and will still be useful.

Provides insight into fundamental questions

- Defines the questions
- Answers some
- Many are open!
- Very close connection with logic, algorithms, linguistics, others.

Provides advance problem-solving tools.

- Springboard for more advanced courses
- Research
- Applications

Practice with mathematics and proofs.

General knowledge (e.g., management, technology and policy).

Practical applications of automata and complexity theory

- Programming languages and compilers.
- Formal verification and computational logic.
- NP-complete and undecidable problems show up in every field in computer science.
- Etc. (covers a *lot*)

Course administration

Textbook: Hopcroft, Motwani, and Ullman, Introduction to Automata Theory, Second Edition.

Grades: Homeworks (50%), Midterm (20%), Final (30%)

Stuff is due on Tuesdays, by the beginning of class. Unless you are sick or injured, no late homeworks will be accepted. However, we will drop your lowest homework grade, which effectively allows one late homework.

Please help reduce the spread of swine flu. Please do not come to class if you are sick. That includes homeworks and exams – let us know and we'll make arrangements as necessary.

Midterm is in class, Tuesday, Nov 3.

Final is Wednesday, Dec 9, 12:15 PM - 3:15 PM.

Do not announce to me that you have made unrefundable plane reservations to leave before the final.

Web page: To appear shortly at <http://www.stanford.edu/class/cs154>

Basic concept

Def An *alphabet* is a non-empty finite set. The members of the alphabet are called *symbols*.

Examples: $\{0, 1\}$, ASCII, or any other character set.

The capital Greek sigma (Σ) is often used to represent an alphabet.

Strings

Informally: A *string* is a finite sequence of symbols from some alphabet.

Examples:

- ϵ – the empty string (the same for every alphabet). (Leaving a blank space for the empty string is confusing, so we use the Greek letter “epsilon”).

ϵ is not a symbol! It is the string with no symbols; the string of zero length.

- 000, 01101 are strings over the binary alphabet

Recursive definition of strings over alphabet Σ .

(This is a bit more formal than the book’s treatment, but equivalent.)

Base: ϵ is a string over Σ

Induction: If x is a string over Σ and a is a symbol from Σ , then xa is a string over Σ .

(Think of xa as appending a symbol to an existing string.)

Notation: The set of all strings over an alphabet Σ is written Σ^* .

Aside: Inductive data types

Structural induction is a widely-used proof method in computer science.

It starts with a definition of a recursive abstract data-type. Values in the datatype are basically trees (called *terms*).

Such a definition has *constructors*, which are functions that build bigger terms out of terms (or no terms). Constants are constructors with no arguments.

In the case of strings, ϵ can be thought of as a constructor with no arguments, and xa as a constructor that takes two arguments, a string and a symbol, and returns another string.

Structural induction is basically a case analysis, proving something for each constructor. A generic proof looks like this:

Theorem: $P(w)$ holds for all strings w over Σ .

Proof: We prove by induction on the structure of w .

Base: $P(\epsilon)$ holds [for some reason].

Induction: Suppose $P(x)$ for any string $x \in \Sigma^*$ and let a be any symbol in Σ . Then $P(xa)$ [for some reason].

Q.E.D.

Length of a string

Many functions are defined recursively on the structure of strings, and many proofs are done by induction on strings.

Informally: The *length* of a string is the number of occurrences of symbols in the string (the number of different positions at which symbols occur).

The length of string x is written $|x|$.

Recursive definition of length (didn't do in lecture)

It is common to define functions recursively on the structure of abstract data types.

Base: $|\epsilon| = 0$.

Induction: $|xa| = |x| + 1$.

Concatenation of strings

Informally: The concatenation of strings x and y over alphabet Σ is the string formed by following x by y . It is written $x \cdot y$, or (more often) xy .

Examples:

- $abc \cdot def = abcdef$
- $\epsilon \cdot abc = abc$

Recursive definition of concatenation: (not in lecture)

The definition is recursive on the structure of the second string:

Base: $x\epsilon = x$ if x is a string over Σ .

Induction: $x(ya) = (xy)a$ if x and y are strings over Σ and $a \in \Sigma$.

Note: The parentheses are not symbols, they are for grouping, so $x(ya)$ is x concatenated with ya .

Languages

Def A language over Σ is a subset of Σ^* .

Note: Of course, this omits almost everything that intuitively think is important about a language, such as meaning. But this definition nevertheless leads to incredibly useful and important results.

Examples:

- \emptyset (the empty language)
- $\{\epsilon\}$ (the language consisting of a single empty string).
- The set of all strings with the same number of *as* as *bs*.
- The set of all prime numbers, written as binary strings.
- The set of all strings representing Java programs that compile without errors or warnings.
- The set of all first-order logic formulas.
- The set of all theorems of number theory, in an appropriate logical notation.
- The set of all input strings for which a given Boolean Java function returns “true.”

Proof Expectations

We don't want to lose sight of the forest because of the trees.

Here are the “forest-level” points with proofs.

- What is the proof strategy?
 - Induction on strings. **What are the base and induction steps?**
 - Induction on expressions. **What are the base and induction steps?**
 - Diagonalization
 - Reduction from another problem. **Which direction is the reduction?**
- What are the key insights in the proof?
- Often this is a construction (often something that can be implemented as a computer program)
 - Translation between regular expressions, various finite automata.
 - Translation from one problem to another.

Make sure your document explain these things clearly in your proofs.

If we can see QUICKLY that you did the right kind of proof and got the major points right, you'll get nearly full credit.

Proof Guidelines

1. State what is being proved precisely and clearly.
2. Start proof with an explanation of the strategy (e.g. “induction on y ”)
3. Provide guideposts (e.g., **Base**, **Induction**)
4. Highlight the interesting key parts of the proof (where did you have to be clever?)
5. Make it easy for the graders to see these things.