Lecture 0: Introduction, Review, and Languages

Ryan Bernstein

1 Introductory Remarks

1.1 Syllabus Overview

1.1.1 Textbook

The recommended textbook for this course is the third edition of Michael Sipser's *Introduction to the Theory of Computation*. I'll post chapter readings on the course web site, but these won't be graded, and homework problems won't be assigned directly from the book.

This is why the book is recommended, rather than required. It means that if you can get an older edition, that will probably work fine. If you think you can Google your way to success, I won't stop you — as long as you have a way to learn the material, you should be fine.

1.1.2 Grade Breakdown

Midterm Exam: 25%
Final Exam: 25%
Homework: 50%

1.1.3 Homework and Collaboration

It's time to discuss two of the worst-kept secrets in higher education:

1. Students will want to work together on homework problems
2. This is a course that deals mostly in solved problems. Solutions to some problems may be available online.

Rather than continue to pretend that these things don't happen, I'd like to encourage them. I think that both of these can be great ways to learn, provided that they're done right. Where both of these practices are concerned, my philosophy is this: don't copy/paste.

Solutions should be written up in your own words. Differences in wording can reveal how much you understand about the solution that you submit. If submissions are suspiciously similar, I may ask you to verbally defend them.

I recommend that you attempt problems yourself before looking them up, but obviously, this is unenforceable. If you do look up a solution to a difficult problem online, *this is okay*, but please include a link with

your submission. This way I can highlight particularly enlightening resources (or particularly misleading ones) in class, which may help other people.

Please ensure that homework submissions are legible. If you can handwrite legibly, feel free, but typed solutions are encouraged. If I cannot read your submission, I will not give it the benefit of the doubt.

1.2 Attendance

Attendance will be taken per department policy, but isn't formally factored into your grade. That said, if you're somebody I recognize, I'm more likely to give your grade a bump if you end up near a border between letter grades.

To start, I'm going to take attendance out loud so that I can try and learn names. I'm pretty bad with names, so if I forget yours, please don't take it personally. After a week or so, I'll just pass around a sign-in sheet at the beginning of class.

1.3 Course Overview

This course is the study of abstract machines. These aren't machines like MacBook computers or Intel processors. We don't care how these machines are physically implemented — we won't be talking about register width, pointers, or floating-point units in this class. Instead, the process of "constructing" an abstract machine is as simple as providing a full description of its behavior.

The majority of the course will be spent constructing machines that recognize languages. We'll construct a machine to recognize a given language. This machine will take a string as input, and output $TRUE$ or $FALSE$ dependent on whether or not the input string is in the language. Since these are abstract machines, you could emulate their behavior in software, creating a function that takes a string and returns a boolean.

We'll begin with simple machines that aren't very powerful, but over the course of the term, we'll build up to Turing universal machines. A Turing universal machine is capable of computing anything which is computable. You may have heard programming languages referred to as "Turing complete". This means that the language is capable of implementing any Turing machine, and accordingly, computing anything which can be computed. Because of this completeness, constructing the types of machines that we'll see at the end of the course is actually going to be our first steps into constructing algorithms.

Some of the languages that we'll look at are incapable of being recognized, even by Turing universal machines. Another implication of Turing completeness is that these problems are uncomputable. This is the reason that our compilers can't warn us about things like infinite loops or division by zero.

For the majority of the term, we won't care about efficiency at all. All we care about is whether or not we can create a machine that can compute an answer. At the end, though, we'll begin looking at time and space complexity of computations. There are some problems that we actually count on being difficult to compute. This is the basis for things like cryptography, in which we want it to be difficult or impossible for an adversary to compute or guess our keys. Here, at the edge of computer science, we'll encounter some problems that are still open, some of which have million dollar bounties.

2 Review: Collection Types

The first thing we need to review comes from CS 250. We'll be looking at a particular set of related concepts, which in industry, are referred to as *collection types*. More specifically, we'll be looking at sets, sequences, and tuples.

Sometimes, especially in the second half of the course, we'll be proving things about specific collections. More often, though, we'll be using these collections as the basic building blocks with which we can construct computational models.

2.1 Sets

A set is simply an unordered collection of elements. We refer to sets containing no elements using the symbol $\emptyset$. We write set literals between curly braces, like so:

$$\{0, 1, 2, b, 7\}$$

In this course, we may also define sets using English descriptions between braces, like so:

$$\{x \mid x \text{ is a multiple of three}\}$$

The vertical bar is read "such that". By convention, we name sets with capital letters.

The number of elements in a set S is called the *cardinality* of S. We write this $|S|$. For instance, $|\{1, 2, 3, 4\}| = 4$. Some sets, such as the set of all even integers, have infinite cardinality.

Elements of a set can have any type, including other sets. Don't confuse $\emptyset$ with $\{\emptyset\}$. The first is the empty set itself, while the second is a set *containing* the empty set. The second set is nonempty — $|\emptyset| = 0$, while $|\{\emptyset\}| = 1$.

2.1.1 Well-Known Sets

We'll often make reference to some well-known sets from number theory. These include, but are not limited to:

The natural numbers, $\mathbb{N}$

> The natural numbers are the numbers that we use to count things, [0, 1, 2, ...]. Whether or not zero should be included is a topic of some debate. For the purposes of this course, zero is included in the natural numbers.

The integers, $\mathbb{Z}$

> As computer scientists in the upper division, you should be familiar with integers at this point. This is just a reminder that the set of integers is referred to as $\mathbb{Z}$. Sometimes, we may also refer to the strictly positive integers (i.e. $\mathbb{N} - \{0\}$) as $\mathbb{Z}^+$, and the negative integers as $\mathbb{Z}^-$.

The rational numbers, $\mathbb{Q}$

> Rational numbers are those numbers that can be expressed as a fraction of two natural numbers. $\frac{1}{10}$ and $\frac{2}{3}$ are rational numbers; π and e are not.

The real numbers, $\mathbb{R}$.

Real numbers are those that can be expressed in floating-point notation.

2.1.2 Operations on Sets

Element-wise Inclusion

If some element x is in the set S, we write $x \in S$. If x is *not* in S, then we write $x \notin S$. If $A = \{1, 2, 3\}$, then $1 \in A$ and $4 \notin A$.

Equality

Two sets A and B are *equal* iff they contain exactly the same collection of elements.

Remember that sets by definition are blind to order and repetition. This means that $\{1, 2, 3\} = \{3, 2, 1\}$, and that $\{4, 5, 6\} = \{4, 4, 5, 5, 6, 6\}$.

Subsets

If every element of the set A is also in the set B, we write $A \subseteq B$. This means that A is a *subset* of B (or equivalently, that B is a *superset* of A. As an example, we can say that $\mathbb{Q} \subseteq \mathbb{R}$, that $\mathbb{Z} \subseteq \mathbb{Q}$, and that $\mathbb{N} \subseteq \mathbb{Z}$.

If $A \subseteq B$ and $A \neq B$, we say that A is a *proper subset* of B, which is written $A \subset B$. As with equality and inclusion, we can put a slash through the subset operators to indicate that A is *not* a subset of B ($A \not\subseteq B$).

Difference

The *difference* between sets A and B, written $A - B$, is defined as follows:

$$A - B = \{x \mid x \in A \text{ and } x \notin B\}$$

That is to say, all of the elements that appear in A, but not in B.

- $\{1, 2, 3\} - \{2, 3, 4\} = \{1\}$. Note that 4 is not included.
- $\mathbb{Z}^+ - \mathbb{N} = \{0\}$
- $\mathbb{N} - \mathbb{Z}^+ = \emptyset$

Note that $A \subseteq B \rightarrow A - B = \emptyset$.

Complement

The complement of some set S, written $\overline{S}$ or $S^{\complement}$, is the set of elements that do not appear in S. Using the notation we introduced earlier, we could say:

$$\overline{S} = \{x \mid x \notin S\}$$

We'll typically use the overline notation unless it's inconvenient to do so. If we do proofs involving complements of complements, it's often clearer to use the $\complement$ notation with parentheses.

When dealing with complements, we often have an implicit *universe*, which is a set that encompasses both S and $\overline{S}$. For instance, if we say that S is the set of even integers, we may be implying that $\overline{S}$ is the set of odd integers. In this case, the implied universe, U, is $\mathbb{Z}$. Then $S = U - \overline{S}$, and $\overline{S} = U - S$.

Some ambiguity here is a consequent of describing sets in natural language. If you ever feel that the universe for some exercise is unclear, feel free to ask for clarification.

Union

The *union* of two sets A and B is the set of elements that are in either (or both) sets. In other words:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

- $\{1, 2, 3\} \cup \{3, 4, 5\} = \{1, 2, 3, 4, 5\}$
- $\mathbb{N} \cup \mathbb{Z}^- = \mathbb{Z}$

Intersection

The *intersection* of two sets A and B is the set of elements that are in *both* A and B. Formally speaking:

$$A \cap B = \{x \mid x \in A \text{ and } x \in B$$

- $\{1, 2, 3, 4\} \cap \{3, 4, 5, 6\} = \{3, 4\}$
- $\mathbb{Z}^+ \cap \mathbb{N} = \mathbb{N}$
- $\mathbb{Z}^- \cap \mathbb{N} = \emptyset$

Note that the union and intersection operators $\cup$ and $\cap$ resemble the logical connectives $\vee$ and $\wedge$, since $A \cup B = \{x \mid x \in A \vee x \in B\}$ and $A \cap B = \{x \mid x \in A \wedge x \in B\}$.

Cross Products/Cartesian Products

The cross product of two sets A and B, written $A \times B$, is the set of all ordered pairs of elements such that the first element is in A and the second element is in B.

$$A \times B = \{(x, y) \mid x \in A \text{ and } y \in B\}$$

Since these are *ordered* pairs, elements that appear in $A \cap B$ will be paired with themselves.

If we take a Cartesian product of more than two sets (e.g. $A \times B \times C$), the resulting sequences don't nest. This means that $A \times B \times C$ produces a set of ordered triples, rather than a set of ordered pairs in which one element is another ordered pair.

We may write $A \times A$ as A^2, or in general, the set of possible n-tuples of elements of A as A^n.

- $\{1, 2\} \times \{3, 4, 5\} = \{(1, 3), (1, 4), (1, 5), (2, 3), (2, 4), (2, 5)\}$

We'll talk in more detail about ordered pairs when we talk about tuples.

Powersets The *powerset* of a set S, written $\mathbb{P}(S)$, is the set of all possible subsets of S.

$$\mathbb{P}(S) = \{A \mid A \subseteq S\}$$

- $\mathbb{P}(\{1, 2, 3\}) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

$\emptyset$ is a member of the powerset of every set.

A powerset of a set S has cardinality $2^{|S|}$. We'll prove this in a bit when we talk about proof strategies.

2.2 Sequences

A sequence is an ordered list of 0 or more elements. Unlike sets, repetition is allowed.

We talked before about how this course is the study of abstract machines. While they're not strictly machines, sequences are a great illustration of an abstract concept. In the 160 series, you probably talked quite a bit about the differences between arrays and linked lists.

When dealing with abstract sequences, though, we don't actually care how they're implemented. Elements could be stored contiguously, in a linked list, or in some other data structure entirely. All we care about is having a list or sequence of elements.

The semantics of abstract sequences are really pretty intuitive — it's just a list.

We might write sequence literals between parentheses, like so:

$$(a, b, c)$$

In this course, though, we'll often be looking at strings over some alphabet. In that sense, we'll often write strings over the alphabet $\{a, b, c\}$ by simply concatenating the symbols in question, like so:

$$S = ababcad$$

2.3 Tuples

We can think of a tuple as a sequence with a known number of elements. Often, we assign some meaning to each element, so the order is important. A well-known example would be something like the ordered pairs we use to describe graph coordinates. $(2, 4)$ and $(5, 3)$ can be thought of as 2-tuples where each entry is an integer. The first element represents the x coordinate of the point in question, while the second element represents the y coordinate.

The elements need not come from the same set. In CS 250, we learned to describe a graph as a 2-tuple (V, E), where V is a set of vertices and E is a set of edges connecting them. We also learned to describe a partially-ordered set as a 2-tuple (S, R), where S is a set of elements and R is a binary order relation.

Since we can think of tuples as sequences with finite length, we use parentheses to write tuple literals as well. Sorry.

3 Review: Functions

A function is a mapping from some input set D (the *domain*) to some output set R (the *range*). If a function f maps from some set A to some set B, we write it as follows:

$$f : A \to B$$

The intuitive notion of functions that we get from programming will be sufficient for most of the purposes of this course. The important part to remember is this: to be a properly defined function, f must be *deterministic* — that is to say, f should yield the same output for a given input every time it is evaluated.

As computer scientists, we're used to defining functions by writing out a procedure to compute the result from the input. But if a function's domain is finite, we can define the function as a table. We could consider truth tables in propositional calculus or digital logic to be examples of functions mapping $\{T, F\}^k \to \{T, F\}$ for some $k \in \mathbb{N}$:

A	B	$A \wedge B$
F	F	F
F	T	F
T	F	F
T	T	T

$AND : \{T, F\}^2 \to \{T, F\}$

In computer science and programming, we often refer to functions with range $\{T, F\}$ as *predicates*.

4 Review: Proof Strategies

4.1 Proof by Contradiction (Indirect Proof)

We looked at indirect proofs in the context of formal logic in CS 251, but we can use the same strategy in informal proofs. If we want to prove that something is true, we first assume that it is false. We can then follow some series of logical steps to show that this causes a contradiction. This implies that our assumption must have been mistaken, and our proposition is true.

4.1.1 Example: Proof that $\sqrt{2}$ is Irrational

Assume that $\sqrt{2} \in \mathbb{Q}$. This means that there are some $m, n \in \mathbb{Z}$ such that $\frac{m}{n} = \sqrt{2}$.

Reducing $\frac{m}{n}$ to a minimal fraction means that one or both of m and n is odd, since if they were both even, $\frac{m}{n}$ could be further reduced.

Squaring both sides yields:

$$\frac{m^2}{n^2} = 2$$
$$m^2 = 2n^2$$

This implies that m^2 is even, which in turn implies that m is even. This means that $m^2 = 2k$ for some $k \in \mathbb{Z}$. We can substitute this into the equation above.

$$(2k)^2 = 2n^2$$
$$4k^2 = 2n^2$$
$$2k^2 = n^2$$

This implies that n^2 is also even, which means that n itself is even by the same logic as above.

Since we concluded that m or n *must* be odd when we reduced the fraction, this introduces a contradiction. We can therefore conclude that our assumption is false, and $\sqrt{2} \notin \mathbb{Q}$.

4.2 Proof by Induction

I have a serious issue with the way that induction is taught. For some reason, we always introduce students to induction by having them find closed forms of summations. This (understandably) causes students to associate the concept of induction with finnicky algebra, and to overestimate its complexity.

Inherently, induction isn't that outlandish a concept. First, we have to map our domain to some series. This mapping will usually be pretty obvious — in this course, we'll usually be dealing with string lengths, so the next element of the series is a string with length $n + 1$.

Once we've done this, we can perform induction's two basic steps:

The Base Case

> First, we pick a single case near the beginning of our series. In the context of string length, we might show that our property holds for strings of length 0 or 1.
>
> Sometimes, we'll be inductively proving that a property holds for elements above some k. In this case, we'll use k as the base case.

The Inductive Step

> The inductive step has us show that if our property holds for some element k, it will also be true for the next element in our series.

Once we've done both of these things, we've shown by induction that our property should hold for every element of our series above the base case.

4.2.1 Example: Proof that $|\mathbb{P}(S)| = 2^{|S|}$

As an example of induction in a non-algebraic context, we'll prove that the cardinality of the powerset of any set S is $2^{|S|}$.

$|\mathbb{P}(\emptyset)| = |\{\emptyset\}| = 1$. Since $|\emptyset| = 0$ and $2^0 = 1$, we can say that $|\mathbb{P}(S)| = 2^{|S|}$ when $|S| = 0$.

Assume that $|\mathbb{P}(S)| = 2^{|S|}$ for some set S. We can then create a new set A such that $|S'| = |S| + 1$ by letting $S' = S \cup \{x\}$ for some $x \notin S$.

The only sets in $\mathbb{P}(S \cup \{x\}) - \mathbb{P}(S)$ are sets that contains x. We can find all of these sets by adding x to every set $A \in \mathbb{P}(S)$. (Note that this proof incorporates proof by construction as well).

Since we have added a new set A' for each set $A \in \mathbb{P}(S)$, we can say that:

$$\begin{aligned}|\mathbb{P}(S')| &= |\mathbb{P}(S)| + |\mathbb{P}(S)| \\ &= 2 \cdot |\mathbb{P}(S)| \\ &= 2 \cdot 2^{|S|} \\ &= 2^{|S|+1} \\ &= 2^{|S'|}\end{aligned}$$

Since $|\mathbb{P}(S)| = 2^{|S|}$ when $|S| = 0$ and $|\mathbb{P}(S)| = 2^{|S|} \rightarrow |\mathbb{P}(S \cup x)| = 2^{|S|+1}$ for any $x \notin S$, we can conclude by induction that $|\mathbb{P}(X)| = 2^{|X|}$ for any arbitrary set X.

4.3 Proof by Cases

Proof by cases is a pretty simple concept when used in informal arguments. If we want to prove that a predicate is true for every element in some domain, we partition the domain into a set of disjoint cases that encompass it in its entirety. Then, we prove that the predicate is true for the elements in each partition.

For proofs involving integers, we might partition $\mathbb{Z}$ into positive, negative, or zero if our predicate might be sign-dependent. Another common partition is into even integers and odd integers, especially for predicates involving division.

4.3.1 Example: Proof that $\sqrt{x^2}$ is non-negative for any $x \in \mathbb{R}$

Any $x \in \mathbb{R}$ is positive, negative, or zero, so we can use these three cases in our proof.

Case 1: $x > 0$

If $x > 0$, squaring x will be multiplying a positive number by a positive number, which should always yield a positive result. The square root of a positive real number will always be positive.

Case 2: $x < 0$

If $x < 0$, squaring x will be multiplying a negative number by a negative number, which should yield a positive result. The square root of a positive real number will always be positive.

Case 3: $x = 0$

Since we know the value of x, we can prove that this is nonnegative simply using the fact that $\sqrt{0^2} = \sqrt{0} = 0$.

5 Languages

5.1 Alphabets

We define an alphabet as a finite, non-empty set. While sets can usually have sets as members, this doesn't really make sense in the context of an alphabet. More often, our alphabet will be a set of simple tokens, such as single characters or short strings.

Examples of alphabets include:

- The letters `a-z`
- The digits, `0-9`
- The binary alphabet, $\{0, 1\}$
- The morse code alphabet, $\{\cdot, -\}$
- NES controller inputs, $\{\leftarrow, \rightarrow, \uparrow, \downarrow, b, a, start, select\}$

It's relatively rare that we refer to a large number of alphabets in the same context, so we almost always name alphabets Σ or Γ.

5.2 Strings/Words

Strings or *words* over some alphabet are finite sequences over some alphabet. Since these are called strings or words, we often refer to arbitrary strings as s or w, sometimes with a subscript to distinguish between them.

The length of a string s, written $|s|$, is the number of symbols that it contains. Strings of length 0 are allowed — we refer to strings of length 0 as the *empty string*, which is abbreviated as ϵ.

If we have some string s and some string t, the *concatenation* of s and t — that is to say, the sequence of symbols comprising s followed immediately by the sequence of symbols comprising t — is simply written st.

If we concatenate a string s with itself, we can write this as s^2. We can generalize this to any natural number n: s concatenated with itself n times is written s^n. As an example, we could express the string $ababab$ as ab^3.

5.3 Languages

A language is a set of strings. Examples of languages include:

- $\{1001\}$
- The set of all strings composed of zero or more as
- $\{w \mid w \text{ ends with a } 1\}$
- The set of binary strings that contain `010` as a substring
- Any programming language. Part of the compilation process involves checking to ensure that the programs you write are valid, using techniques like the ones we'll be using in this class

You should have learned in CS 201 about how we use binary to encode different types of data, including numbers, characters, programs, and images. In fact, we can represent anything that we can discretize using just the alphabet $\{0, 1\}$. This means that other examples of languages include:

- The set of all valid bitmap images
- The set of all valid bitmap images that depict cats

Generally speaking, if you have a set of discretizable things — that is to say, things that can be represented with finite precision — you can create a language out of this set, although it may be difficult (or impossible) to create a machine that recognizes it.

5.4 Operations on Alphabets and Languages

Since languages and alphabets are fundamentally sets, all of the set operations that we discussed previously still apply. However, there are some additional operations that we often perform on languages, which we'll introduce here.

5.4.1 Concatenation

We've already talked briefly about what concatenation means in terms of strings. We can also concatenate languages, though. The concatenation of two languages A and B, written $A \circ B$, is defined as follows:

$$A \circ B = \{st \mid s \in A \text{ and } t \in B\}$$

In English, the language $A \circ B$ is the set of all strings that can be created by taking a string from A and concatenating it with a string from B.

Note that if $\epsilon \in A$, then $B \subseteq A \circ B$, and if $\epsilon \in B$ then $A \subseteq A \circ B$.

5.4.2 The Kleene Star

As we've seen, we can build strings from repeated substrings using exponent notation. We can generalize this into a language fairly easily: given a string s, we can generate a language $S = s^*$. Then S is the set of all strings that are zero or more concatenated repetitions of s.

An exponent star used to express the concept of "zero or more repetitions" is known as the Kleene star or Kleene closure after its creator, Stephen Cole Kleene.

We can also generalize this notation and apply it to other, related items. The Kleene closure over an alphabet, written Σ^* (or Γ^*, depending on the name of the alphabet) is the language containing every string that can be generated from zero or more tokens from that alphabet. This gives us an alternative definition of a language:

A language over the alphabet Σ is a subset of Σ^*.

We can also apply the Kleene closure to languages themselves. For some language L, L^* is the language created by concatenating zero or more strings in L.

Note that because the Kleene closure represents *zero* or more concatenated elements, any Kleene closure includes the empty string ϵ, regardless of whether the star is applied to a string, alphabet, or language.