CSE396: Intro Theory of Computation, Spring 2026

[show course website, Piazza, and TopHat]
[do unofficial test of attendance-taking on TopHat]
[go over syllabus and course rules]
[talk about textbook]

Brief Course Overview

1. Formal Languages as "Math With Symbols" (this week).
2. Finite Automata and Regular Expressions (this month into next, long Sipser ch. 1).
3. Context-Free Grammars and Languages (next month, Sipser ch. 2 but not section 2.4).
4. Computability and Undecidability (April, Sipser chs. 3--5 skimming section 5.2).
5. A bit of Computational Complexity (May, Sipser ch. 7 and one page of ch. 9 for a proof).

The very last lectures will be on the Cook-Levin Theorem for showing **NP**-*hardness* and **NP**-*completeness*. It is possible that I will cut down some of the CFG content and do more with Boolean circuits in the run-up to the Cook-Levin proof.

Formal Languages

Which comes first, the number or the symbol? Let us multiply

```
    47
 x  43
-------
   141
  188_
-------
  2021
```

That year wa the product of two nearly-equal primes, both congruent to 3 modulo 4. This makes 2021 into a Blum Integer, and the importance of this to cryptography will also be touched on in the last weeks. Well, 2025 was a perfect square, 45×45, but $2026 \; = \; 2 \times 1013$ is comparatively boring since 1013 is prime. Staying with 2021, would you say the operations above are *numeric* or *symbolic*? What if we do this in binary notation?

```
          101111
       x  101011
     -----------
          101111
         101111
       101111
```

```
        101111
        -----------
        11111100101
```

Maybe this *feels* more symbolic. In my youth there was more a balance between "analog" and "digital" as monikers for computing, but that's all gone digital. (Music, however, has made a large move back to analog.) In any event, this course handles the symbolic side.

This begins with defining the **alphabet** of symbols used. The alphabet for binary arithmetic, and binary strings in general, is $\{0, 1\}$. In many ways, this is the only alphabet we formally need to consider. The alphabet $\{a, b\}$ will have a different "feel"---I will use it more often than the text because it feels like using words, and some examples will put other letters $c, d, e, \dots$ to good use. But formally, $\{a, b\}$ works just like $\{0, 1\}$ if you think $a = 0, b = 1$ (or vice-versa). Moreover, to a computer, all letters on our keyboards get translated to binary strings, variously by the ASCII code, Unicode, or UTF-8, which is an amalgam of the two.

A motive for making alphabets more general is to incorporate **tokens** as basic units. Tokens are often notated inside angle brackets $\langle \dots \rangle$ and that is exactly what this paper does. For example, on the bottom right of its second page, it refers to the "string"

$$<t_i, \texttt{S}> <t_i, \texttt{S}> <t_i, \texttt{I}> <t_i, \texttt{I}> <t_i, \texttt{R}>$$

where each "t_i" is a person and S,I,R are the epidemiological labels for people who are in the state of being Susceptible, Infected, or Recovered. This is "meta"---we have symbols inside our symbols, but the point is that the (Nondeterministic) Finite Automaton defined in the paper takes these tokens as its basic inputs.

So we want to think abstractly of a general alphabet---and the convention is to use a capital Greek Σ to denote one. This may be confusing---Σ usually stands for a sum, and we may have a few of those too. But we use a limited set of letters in our notation, and Σ took hold---as exemplified in the same paper. Most of the time we will have $\Sigma = \{0, 1\}$ or $\Sigma = \{a, b\}$, however.

Definition: A **string** is a sequence of characters.

In C++ terms, `string = list<char>,` whereas `alphabet = set<char>.`

Definition: A set of strings is called a **language**. `language = set<list<char> >`

This is "barebones"---it's like saying the English language equals the set of words you can play in the game *Scrabble*. Chapter 2 on context-free grammars will be all about defining rules on top of words and the kinds of more-human-like languages you can get as a result.

Here are some pertinent examples:

- **BAL** stands for the language of balanced-parentheses strings. It has $\Sigma = \{ (\,,\,) \}$. Well, OK, the curly braces and comma are part of mathematical notation for sets; the '(' and ')' are the actual characters.
  - Some strings in BAL: ( ), (( )), ( ) (( )), (( ) (( )))
  - Some strings not in BAL:) (, (( ), ( ) ( )).
  - Is the empty string in BAL?
- **PAL** stands for the language of strings that read the same forwards and backwards, i.e., **palindromes**. The idea applies to whatever alphabet you use.
  - Some strings in PAL: $abba$, 10101, "amanaplanacanalpanama"
  - Some strings not in PAL: $baba$, 101001, "A man, a plan, a canal: Panama."
  - Is the empty string in PAL? (Yes)
- The set of prime numbers is not a *language* in our formal sense. In C++ terms it has type `set<int>` not `set<string>`. But if we specify it as the set of standard binary representations of prime numbers without leading zeroes, then we get the formal language **PRIMES** = $\{10, 11, 101, 111, 1011, 1101,\ 10001,\ 10011,\ 10111,\ 11101,\ 11111,\ \ldots\}$.

The *empty language*, like any empty set, is denoted by $\emptyset$. The empty string will be denoted by ϵ (Greek lowercase epsilon) in this course. [Other sources---including the above paper---use λ (Greek lowercase lambda) for the empty string. I will often mention notational variants in sources you may see on the Web. Color codes: **bold black** for notation and terms that are completely standard, but **orange** for notation that varies between sources or names and terms I've made up.]

What's the difference between $\emptyset$ and ϵ? First, the former is a `set`, the other a `string`. Second, we will see the difference is like that between the numerical 0 and 1 as numbers. Observe:

- The concatenation $x \cdot c$ of a `string` x and a `char` c is the string xc. For example, $aab \cdot a = aaba$. An English rendering of $\cdot$ is "*and then*".
- The concatenation $x \cdot y$ of strings x and y is the string xy. E.g., $aab \cdot aba = aababa$.
- This is the same as what you get by "catting on" to x the chars in y one at a time.
- If $y = \epsilon$, then y has no chars, so the last point is a no-op. So: $x \cdot \epsilon = x$ is a general rule, for all strings x. Likewise, $\epsilon \cdot x = x$ is a general rule. That's how ϵ is like 1. (Well, this makes $\cdot$ analogous to multiplication, but it's not commutative: $aab \cdot aba \neq aba \cdot aab$.)

To really compare it with $\emptyset$, we need to involve ϵ in a language. So consider: $\{\epsilon\}$. This is a *set* whose only member is a *string*, so it is a `set<string>`, which is a `language`. Next we need to "lift" the concatenation operation up to work between languages. This needs a definition:

Definition 1: Given any two languages A and B (their being "over" the same alphabet Σ is understood here), their **concatenation** is the language $A \cdot B$ defined by

$$A \cdot B = \{x \cdot y : x \in A \wedge y \in B\}.$$

An intuition for this is that strings are like streams of data from sensors, and languages $A, B, \dots$ are tests telling whether chunks of data meet respective conditions for being OK. So a string z passes the $A \cdot B$ test if it consists of a portion x that passes the A test *and then* a portion y that passes the B test. Here's a little swervy test of notation: Does $A \cdot A = \{x \cdot x : x \in A\}$? The answer is that this is too narrow. Suppose $A = \{0,1,2,3,4,5,6,7,8,9\}$ represents the condition of being a digit character (`\d` if you've done string-matching). Then $A \cdot A$ should allow any two digits, not just the doubled cases $00, 11, \dots, 99$. Instead, $A \cdot A = \{x \cdot y : x, y \in A\}$.

Having understood that about Definition 1, let us try the "edge cases" $B = \emptyset$ and $B = \{\epsilon\}$:

- $\boldsymbol{A \cdot \emptyset} = \{x \cdot y : x \in A \wedge y \in \emptyset\} = \{x \cdot y : x \in A \wedge \mathit{false}\} = \{x \cdot y : \mathit{false}\} = \emptyset$
- $\boldsymbol{A \cdot \{\epsilon\}} = \{x \cdot y : x \in A \wedge y \in \{\epsilon\}\} = \{x \cdot \epsilon : x \in A\} = \{x : x \in A\} = \boldsymbol{A}.$

Likewise, $\emptyset \cdot A = \emptyset$ for any language A, whereas $\{\epsilon\} \cdot A = A$ always. Intuitively, $A \cdot \emptyset = \emptyset$ says that if a sensor at a required stage fails then the whole test series fails. Whereas, $A \cdot \{\epsilon\}$ means that the second condition passes automatically on the heels of the first, without needing (or allowing) any more data to be taken.

If `language = set<string>` isn't "up there" enough, there's also the term that a **class** is a set of languages. The first major example will be the class **REG** of **regular languages**.

Regular languages are those defined by **regular expressions**. Here is just a little preview of what those involve---as part of a generally important notice to bear in mind:

Lectures in this course are held on `T(hur+ue)sdays`.

The '+' symbol means "or" here. Well, if you've already gotten and looked at the textbook, it writes $\cup$ instead of $+$ in regular expressions. When we hit grammars, we will write $|$ instead, and you've already used one or two of these vertical bars to mean **or** in a programming language. But in this case, those could get confused with the letters **u** and **i**, respectively. So I wrote $+$. Which can get confused with addition---but hey, "or" is a kind of Boolean addition.

There's also an invisible symbol here: $\cdot$ for **concatenation** again. This is analogous to numerical multiplication, except that it isn't commutative: $a \cdot b$ is a different *string* from $b \cdot a$, even though it would be equal as numbers. (Well hey, multiplying *matrices* isn't commutative either.) To be super-pedantic, the regular expression in light purple is `T·(h·u·r + u·e)·s·d·a·y·s` . The main reason to write $+$ and $\cdot$ is that the distributive law holds:

$$a \cdot (b+c) \; = \; a \cdot b \; + \; a \cdot c, \text{ and on the right, } (a+b) \cdot c \; = \; a \cdot c \; + \; b \cdot c.$$

It holds with languages too: $A(B \cup C) \; = \; AB \cup AC$ and $(A \cup B)C \; = \; AC \cup BC$. With sets, we really do want you to write $\cup$ not $+$. Later we will use letters $r, s, t, \ldots$ and also Greek letters α, β, γ to stand for regular expressions, which stand for languages...and then you can use either symbol.

You can also involve the empty string ϵ in regular expressions. We could try to be even more clever and write

$$\mathtt{T} \cdot (\mathtt{h}+\epsilon) \cdot \mathtt{u} \cdot (\mathtt{r}+\mathtt{e}) \cdot \mathtt{s} \cdot \mathtt{d} \cdot \mathtt{a} \cdot \mathtt{y} \cdot \mathtt{s}.$$

This matches the words "Tuesdays" and "Thursdays" like before. But the defect is that it also matches the excess strings "Tursdays" and "Thuesdays". Because it allows excess strings that don't conform to the target concept of "days of the week", we will say this regular expression is **unsound**. Well, if "days of the week" is the exact specification we want to capture, the original regular expression fails that in a different way: it fails to match the other five days of the week. We will say it is **not comprehensive** for the target concept. We will define the positive sides of these terms, **sound** and **comprehensive**, later.

[If even more time allows, tell the story at https://rjlipton.wordpress.com/2015/02/23/the-right-stuff-of-emptiness/ . Wherever the break comes, the rest will be part of notes for the week 2 recitations.