FLA 2026 Lecture Notes

Brendan Griffiths

2026-02-13

Table of contents

Preface

This book is a collection of lecture notes for COMS3003A and COMS3021A.

1 An Introduction to Theoretical Computer Science

Maybe one day I will write something for this page...

2 Data, Representations, and Problems

2.1 Data

2.1.1 Primitive Data

Symbols: The most basic unit - may be anything that can be Read & Written

- Numbers: $0, 1$
- Latin Letters: $A, B, C, ..., Y, Z$
- Greek Letters: α, β
- Electricity: $1.5V$, $3.0V$
- Sound

Alphabet: A finite set of symbols

- Common notation Σ, Γ, Δ

Strings / Words:

- A finite sequence of symbols
- ε is the empty word
- Words have a length $|w| = n \iff$ there are n symbols in w
  - $|\varepsilon| = 0$

String Operators:

- Concatenation - connecting two symbols or strings
  - $w_1 \circ w_2 = w_1 w_2$
- Repeat Concatenation - concatenate the same string the specified number of times
  - $w^n = w \circ w \circ w \circ ... \circ w$

2.1.2 Describing Strings

String Generators: Describe a set of strings

- Union - Either string
  - $w \cup v = \{ w, v \}$
- Kleene Star - Repeat the string any numbers of times
  - $w^* = \{ \varepsilon, w, ww, www, wwww, ... \}$
- Kleene Plus - repeat the string at least once
  - $w^+ = ww^* = \{ w, ww, www, ... \}$
- Otherwise called a regular expression

The set of all strings over an alphabet is

$$\Sigma^*$$

> **Note**
>
> Let $\Sigma = \{ a, b \}$
>
> $$\Sigma^* = \left\{ \begin{array}{c} \varepsilon, \\ a, aa, aaa, ... \\ b, bb, bbb, ... \\ ab, aba, abaa, ... \\ abba, abbaa, abbaaa, ... \end{array} \right\}$$

2.1.3 Languages

A language L is a set of strings (i.e. a subset of Σ^*)

> **Notation**
>
> As shorthand, we may specify a set L is a language by indicating that it is a subset of all strings.
>
> $$L \subseteq \Sigma^*$$

Language Examples:

- The set of all strings with two symbols

$$\{w \in \Sigma^* \mid |w| = 2\}$$

- The set of all strings that end in 0

$$\{w \in \Sigma^* \mid w = v \circ u \text{ and } u = 0\}$$

2.2 Representations

A representation is an interpretation of objects into strings

If I give you 5 objects, how would you label them? How about 10 objects? What about an infinite number of objects?

Formally, a representation is a mapping from a set of objects X to a string

$$r : X \mapsto \Sigma^*$$

We also want to add the following properties to our representation:

- Every input has some output

$$\forall x \in X, \exists w \in \Sigma^*, r(x) = w$$

- Every output has at least one input

$$\forall w \in \Sigma^*, \exists x \in X, r(x) = w$$

- If two inputs are different, their outputs must be different

$$\forall x, y \in X, x \neq y \implies r(x) \neq r(y)$$

> **i** A brief interlude on relations
>
> A relation $\alpha : X \mapsto Y$ is the more general idea of a function. A function $f : X \mapsto Y$ requires that each $x \in X$ maps to only one $y \in Y$. Relations are a relaxation of that to-one property (called functional). Our first property of all $x \in X$ having at least one $y \in Y$ is called serial.
> Another way to think of relations is using sets. A binary Relation $R : X \mapsto Y$ is the subset $R \subseteq X \times Y$. Notationally, we say $x \, R \, y$ (x relates to y by R) if $(x, y) \in R$
> An example is equality of numbers $=: \mathbb{R} \mapsto \mathbb{R}$
>
> $$\forall x, y \in \mathbb{R}, x = y \iff (x, y) \in \{ \ (a, a) \mid a \in \mathbb{R} \ \}$$

Notation

Instead of writing $r(x)$ we will write $\langle x \rangle_r$ to mark it as a string
If it is clear what representation we are using, then we can drop the r

A representation allows us to take random objects and 'insert' them into strings, but we also want to be able to 'extract' an object out of a string.

So we add the additional requirement that our representation must be invertible. We define some default value for X. If the string cannot be identified, we just map it to this default value.

Warning

We are hand-waving away a fairly important problem. Two strings may represent the same object, for example in decimal numbers: $0.99999\ldots = 1$
As an exercise, try write a more rigorous definition of a representation using a tool you were taught in abstract maths (hint: equivalence)

2.2.1 Natural Numbers $\mathbb{N}$ in an alphabet of k symbols

$$\forall n \in \mathbb{N}, \langle n \rangle_{\mathbb{N},k} = \text{The base-}k\text{ encoding of } n$$

By default, we will use $k = 2$. If the k is left off, then assume it is 2

2.2.2 Integers $\mathbb{Z}$ in an alphabet of k symbols:

$$\forall z \in \mathbb{Z}, \langle z \rangle_{\mathbb{Z},k} = \begin{cases} 0 \circ \langle |z| \rangle_{\mathbb{N},k} & \text{if } z \geq 0 \\ 1 \circ \langle |z| \rangle_{\mathbb{N},k} & \text{if } z < 0 \end{cases}$$

Warning

What does the string 00 mean in this representation?
This is where our "default" value for a "garbage" string comes into play. We cannot recogize 00 according to our rules, so we instead just map it to the integer 0.

2.2.3 Rationals $\mathbb{Q}$ in an alphabet of k symbols

$$\forall \frac{x}{y} \in \mathbb{Q}, \left\langle \frac{x}{y} \right\rangle_{\mathbb{Q},k} = \langle x \rangle_{\mathbb{Z},k} \# \langle y \rangle_{\mathbb{Z},k}$$

We've had to add the symbol # to separate our integers in the fraction. We can simplify this down into just our alphabet of k symbols.

To pull a rational number out of a string, we use the reverse of the above, but if we cannot recognize the string, we instead default to 0.

Binary simplifier example:

$$e_2(x) = \begin{cases} 00 & \text{if } x = 0 \\ 01 & \text{if } x = 1 \\ 10 & \text{if } x = \# \\ 11 & \text{otherwise} \end{cases}$$

2.2.4 Booleans

$$\langle b \rangle_{\mathbb{B},k} = \begin{cases} 1 & \text{if } b = \text{True} \\ 0 & \text{if } b = \text{False} \end{cases}$$

2.2.5 Complex Objects

We know have a fairly robust set of primitive representations. We can encode natural numbers, integers, rationals, and booleans, but how can we encode a more complex object? One that is made up of several sub-objects?

If our object has 2 components (imagine a 2D vector), we can employ the same separation as our rationals.

$$\left\langle \begin{pmatrix} x \\ y \end{pmatrix} \right\rangle := \langle x \rangle \# \langle y \rangle$$

For m components?

$$\left\langle \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{pmatrix} \right\rangle := \langle a_1 \rangle \# \langle a_2 \rangle \# ... \# \langle a_m \rangle$$

⚠ What if we have a vector of rationals?

We use # as a separator for both our vector components and our numerator and denominator. This is a collision of meaning. How can we write something like

$$\begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix}$$

Our current encoding of this just becomes

$$\left\langle \begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix} \right\rangle = \langle a \rangle \# \langle b \rangle \# \langle c \rangle \# \langle d \rangle$$

which is either a 4D integer vector, a 3D vector in $\mathbb{Z} \times \mathbb{Q} \times \mathbb{Z}$, or a 2D vector of rationals. We can replace the vector separator # with either a new symbol ; or an additional #

$$\left\langle \begin{pmatrix} \frac{a}{b} \\ \frac{c}{d} \end{pmatrix} \right\rangle = \langle a \rangle \# \langle b \rangle ; \langle c \rangle \# \langle d \rangle$$

To get back to Binary alphabet we can use the following encoding:

$$e(s) = \begin{cases} 00 & s = 0 \\ 01 & s = 1 \\ 10 & s = \# \\ 11 & s = ; \end{cases}$$

⚠ What is the smallest alphabet?

What is the smallest possible alphabet we can construct? How might this alphabet represent different objects?

2.3 Encodings

Encodings are the technique we use to write symbols in a different alphabet. We apply them on a semi-regular basis in computer science.

An encoding is a mapping that converts a symbol to a string in another alphabet

$$e : \Sigma \mapsto L \subseteq \Gamma^*$$

$$s \in \Sigma \implies e(s) \in L \subseteq \Gamma^*$$

Notation

We can also "encode" strings

$$\text{Let } w = s_1 s_2 ... s_n \in \Sigma^* \text{ and } e : \Sigma \mapsto L \subseteq \Gamma^*$$

$$e(w) = \langle w \rangle_e = e(s_1)e(s_2)...e(s_n) \in \Gamma^*$$

There are two types of encodings:

- Fixed Width: All encoded strings are the same length
  - A new symbol is indicated by reaching the end of the fixed length
  - Example: ASCII
    $$e : \Sigma \mapsto \Gamma^n$$
- Variable Width: All encoded strings have different lengths
  $$e : \Sigma \mapsto \{w \in \Gamma^* \mid |w| \leq n\}$$
  - A new symbol is indicated in some fashion
    - All symbols start the same way, but feature different symbols for the rest
    - A space between symbols
  - Example: Morse Code
    - Morse Code was originally designed as an auditory alphabet. It has three symbols, a *short sound* (dot), a *long sound* (dash), and *no sound* (seperator). However, we can simplify it into a binary alphabet of sound, no sound by defining a dash as the repetition of sound over a fixed time interval (the standard is 3 times the length of a dot).

Warning

All encodings are representations, just from an alphabet of symbols Σ to strings Γ^* instead of a arbitrary set X. Can we make an encoding from a set of strings to another set of strings?

$$e : \Sigma^* \mapsto \Gamma^*$$

2.4 Problems

We want to draw a distinction between an algorithm and a problem.

We define a problem, as the class of inputs, and class of outputs for which we are interested in.

$$P : X \mapsto Y$$

An algorithm is the set of instructions by which we manipulate our inputs into our outputs.

A good intuition for the distinction is that algorithms can call other algorithms, but they cannot call a problem.

> **Note**
>
> To rely on some prior programming experience, you can imagine that a problem is a "declaration" in code, while the algorithm is the actual "definition".
> For example, in C++, in some header (.h) file we write
>
> ```
> unsigned int compute_square(unsigned int n);
> ```
>
> which is our "Problem" COMPUTE_SQUARE : $\mathbb{N} \mapsto \mathbb{N}$.
> and then in a .cpp file we have
>
> ```
> unsigned int compute_square(unsigned int n) {
>     unsigned int n_square = n * n;
>     return n_square;
> }
> ```
>
> as our algorithm.
> This is not a perfect one-to-one relationship. Our header definition lacks the description of what our output should look like as the full definition of COMPUTE_SQUARE is
>
> $$\text{COMPUTE\_SQUARE}(n) = n^2$$
>
> and implementing it in code is itself an *almost* verbatim description of the problem. Use it as a starting off position.

Examples:

- Prime Factorisation: Input any natural, output: list of prime numbers

$$P : \mathbb{N} \mapsto [\mathbb{N}]$$

- Derivative: Input a differentiable function, output the derivative function

$$P : F \mapsto F'$$

- Swap Two Numbers: Input two naturals, output two naturals swapped

$$P : \mathbb{N} \times \mathbb{N} \mapsto \mathbb{N} \times \mathbb{N}$$

We want to "compute" the answer, so we turn the domain and co-domain into a string with representations r_X and r_Y that we can manipulate

$$P : r_X(X) \mapsto r_Y(Y)$$

A **Computational Problem** is therefore a function f from one set of strings Σ^* to another set of strings Γ^*

$$f : \Sigma^* \mapsto \Gamma^*$$

The most common form however is

$$f : \{ 0, 1 \}^* \mapsto \{ 0, 1 \}^*$$

Sometimes, we just want to identify if a given object has some property attached to it

- Is this number even?
- Is it a rational?
- Can I take the derivative?

We call these kinds of a problems a **Decision Problem**

$$f : \Sigma^* \mapsto \{ 0, 1 \}$$

In fact, we can restructure any computational problem into a decision problem.

2.4.1 Decision Problem and Computational Problem Equivalence

As an example: Lets take the computation problem of dividing a natural number

$$\text{Let } n, m \in \mathbb{N}, \text{ and } \frac{n}{m} \in \mathbb{Q}$$
$$\text{COMPUTE\_NATURAL\_DIV}(n, m) = \frac{n}{m}$$

Let assume an algorithm for COMPUTE_NATURAL_DIV exists, say A.

If we give A a string natural number pair $\langle n, m \rangle$ it will output the string $\left\langle \frac{n}{m} \right\rangle$

We define the Decision problem with $n, m, q \in \mathbb{N}$

$$\text{IS\_NATURAL\_DIV}(n, m, q) = \begin{cases} 1 \text{ if } \frac{n}{m} = q \\ 0 \text{ if } \frac{n}{m} \neq q \end{cases}$$

Let A be an algorithm that solves COMPUTE_NATURAL_DIV.

We write a new algorithm B on that solves IS_NATURAL_DIV

define B on input $\langle n, m, q \rangle$

1. If $A(\langle n, m \rangle) = q$
    - True => Return 1
    - False => Return 0

And now for the reverse:

Let B be an algorithm that solves IS_NATURAL_DIV

define A on input $\langle n, m \rangle$:

1. for q in $[1, n]$
    1. if $B(\langle n, m, q \rangle) = 1$
        - True => return q
        - False => do nothing
2. return $\langle 0 \rangle$

Algorithm A solves COMPUTE_NATURAL_DIV as:

1. If $n < m$ then $\frac{n}{m} \notin \mathbb{N}$ and no answer exists or $n = 0$ therefore the loop will never have a q with $B(\langle n, m, q \rangle) = 1$, thus we return $\langle 0 \rangle$
2. If $n = m$ then $\frac{n}{m} = 1$, therefore for $q = 1$, we will return q
3. If $n > m$ then $n = qm$
    - The smallest possible q is 1 and $m = n$
    - The largest possible q value is $q = n$ and occurs when $m = 1$
    - If $q > n$ then $m < 1$, thus $m \notin \mathbb{N}$, therefore B does not apply

> **Warning**
>
> Try to do the same for a computational problem P. Given $P : \{0,1\}^* \mapsto \{0,1\}^*$ and algorithm A_P, show there exists a decision problem $P' : \{0,1\}^* \mapsto \{0,1\}$

2.4.2 Language and Decision Problem Equivalence

Languages are equivalent to decision problems. We can construct a decision problem out of language membership, and we can construct language membership out of a decision problem.

$$w \in L \iff D(w) = 1$$

Note

Show $D(w) = 1 \implies w \in L$

Let $D : \Sigma^* \mapsto \{0, 1\}$ be a decision problem.

Define the language L as

$$L = \{w \in \Sigma^* \mid D(w) = 1\}$$

Show $w \in L \implies D(w) = 1$

Let $L \subseteq \Sigma^*$ be a language.

Define the decision problem D as

$$D(w) = \begin{cases} 1 & w \in L \\ 0 & w \notin L \end{cases}$$