CSE491/596 Fall 2023, Lecture 1 Sept.: "The Formal Languages Ladder"

The basic object in this course is a **character**. The slogan "It From Bit" expresses that the act of distinguishing between two (or more) symbols is more fundamental than the notion of number or geometry. Computing arithmetical operations $+ * $ etc. employs rules for manipulating the symbols standing for individual bits or digits. We will call the symbol type `char` and its members (however big a set of characters we allow) are "**type 0**" objects.

- An **alphabet** is a set of characters. In C++/Java terms, `set<char>`.
- A **string** is a sequence of characters. In C++/Java terms, a string has type `list<char>`.

We will presume that alphabets and strings are finite. (But infinite alphabets and infinite strings over any kind of alphabet are topics studied in other areas of CS theory.) We use a capital Greek Σ to denote one. This may be confusing---Σ usually stands for a sum, and we will have those too. Much of the time we will have $\Sigma = \{0,1\}$ or $\Sigma = \{a,b\}$. The set of all possible *finite* strings **over** an alphabet Σ is denoted by Σ^*, where the star stands for zero-or-more characters.

Finite binary strings can denote integers, either in *standard notation* 0, 1, 10, 11, 100, 101, 110, 111 ... or in "economical" notation

$$\epsilon = 1,\ \text{"0"} = 2,\ \text{"1"} = 3,\ 00 = 4,\ 01 = 5,\ 10 = 6,\ 11 = 7,\ 000 = 8,\ \ldots$$

The latter gives a bijection between $\{0,1\}^*$ and the set $\mathbb{N}^+$ of *positive* natural numbers. It simply rubs out the initial '1' from the standard binary representation. There is also **2-adic** notation which shifts things one place so that ϵ stands for 0, thus giving a bijection between $\{0,1\}^*$ and $\mathbb{N}$, but I don't encourage it because it is better to think of ϵ as like the number 1. This furthers an analogy between the **concatenation** $x \cdot y$ of two strings and multiplication, in which $x \cdot \epsilon = \epsilon \cdot x = x$. The main difference is that concatenation is not commutative, e.g.

$$011 \cdot 10 = 01110 \neq 10 \cdot 011 = 10011.$$

(But hey, matrix multiplication $AB = C$ usually $\neq BA$ is not commutative either, and we still call it "multiplication." In the quantum section we will see strong analogies between concatenation and matrix operations, even more particularly the matrix **tensor product** $A \otimes B$.) Strings and integers are collectively called "**type 1**" objects.

- A **language** is a set of strings: `set<string>`.

This is "barebones"---it's like saying the English language equals the set of words in some English dictionary. The languages of interest will most often be *infinite*. Note that we already defined Σ^* which is the *infinite* language of all *finite* strings. Languages are "**type 2**". So are functions defined on (whole numbers or) strings. This is because a function f is identified by its **graph** R_f which is the set

$\{(x, y) : f(x) = y\}$. The pairs can be coded up as individual strings---with or without the help of some extra characters such as writing $x\#y$ or taking the parens and commas as literal symbols---so the relation R_f becomes a language.

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*. This is "**type 3**". We will also have **hierarchies** of classes, which gets into "**type 4**", but our "**ladder of abstraction**" will stop there.

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.]

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 $A, B, \ldots$ 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 $00, 11, \ldots, 99$. Instead, $A \cdot A = \{x \cdot y : x, y \in A\}$.

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

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

Likewise, $\varnothing \cdot A = \varnothing$ for any language A, whereas $\{\epsilon\} \cdot A = A$ always. Intuitively, $A \cdot \varnothing = \varnothing$ 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.

Now let us abbreviate $A \cdot A$ as A^2, $A \cdot A \cdot A = A^3$, and so on. This is OK even though concatenation isn't commutative on languages either---hey, neither is matrix multiplication, but A^3 is like raising a matrix to the third power, and that's fine. Just like with numbers and matrices, strings and languages obey the additive law of powers: $A^i \cdot A^j = A^{i+j}$.

We have $A^1 = A$, of course, but what is A^0? In particular, what is $\varnothing^0$? Well, suppose we wrote a program loop to fetch and test a mandated number n of chunks of sensor data?

```
for (int i = 0; i < n; i++) {
   string xi = getNewSensorData();
   if (!A(xi)) { throw Failure; }
}
```

If we invoke this loop with a given number n then it will perform n tests and allow processing to continue without exception only if all n of the tests pass.

- If $A = \varnothing$ and we enter the loop, then the body surely fails, *finito*.
- If $n = 0$ then what happens? The loop is a fall-through. Do we die? No: processing continues undisturbed.
- So what happens if $A = \varnothing$ and $n = 0$? *The same as the second case*: we never get put to the death test, and processing continues undisturbed.

In the third case, nor is any data taken. Thus this is exactly the same situation as when concatenating with $\{\epsilon\}$. It is a "free pass". So $A^0 = \{\epsilon\}$ for any language A, and in particular:

$$\varnothing^0 = \{\epsilon\}.$$

Well, this is just like the numerical convention $0^0 = 1$. In all cases, this is needed to make the additive power law $A^i \cdot A^j = A^{i+j}$ work even when $j = 0$.

[If time allows, tell story at https://rjlipton.wordpress.com/2015/02/23/the-right-stuff-of-emptiness/ .]