1

Formal Languages

First, a quick refresher on set theory.

A set is a collection of elements. Those elements can be anything, including other sets.

Sets can be described by listing their elements, such as $\{a, b, c, \ldots\}$.

The empty set is denoted $\varnothing$, or $\{\,\}$.

A set A is *finite* if it has a finite number of elements, that is, if there is a natural number $n \in \mathbb{N}$ such that A has n elements. If no such n exists, then A is *infinite*.

The main relation on sets is *set membership*, written $a \in A$: a is an element of set A.

Two sets are equal if they have exactly the same elements.

Another relation on sets is *set inclusion*, written $A \subseteq B$: A is a subset of B, meaning that every element of A is an element of B).

Some properties of $\subseteq$:

$\varnothing \subseteq A$ for every A

$A \subseteq A$ for every A

If $A \subseteq B$ and $B \subseteq C$, then $A \subseteq C$

$A = B$ if and only if $A \subseteq B$ and $B \subseteq A$.

If P is a property, then $\{x \mid P(x)\}$ is the set of all elements satisfying the property. (Technically speaking, there are some restrictions on what makes up an acceptable property in this context — but they won't impact us.)

Common operations on sets:

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

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

$\overline{A} = \{x \in \mathcal{U} \mid x \notin A\}$ (where $\mathcal{U}$ is a universe of elements such that $A \subseteq \mathcal{U}$ — the definition of $^{-}$ therefore depends on the universe under consideration)

$A \times B = \{\langle x, y\rangle \mid x \in A, y \in B\}$, the set of all pairs of elements from A and B. This generalizes in the obvious way to products $A_1 \times A_2 \times \cdots \times A_k$.

A function $f : A \longrightarrow B$ associates (or maps) every element of A to an element of B. Set A is the domain of the function, and B is the codomain. The image of A under f is the subset of B defined by $\{b \in B \mid f(a) = b \text{ for some } a \in A\}$.

If $f : A \longrightarrow B$ and $g : B \longrightarrow C$, then the *composition* $g \circ f : A \longrightarrow C$ defined by $(g \circ f)(x) = g(f(x))$.

A function $f : A \longrightarrow B$ is *one-to-one* if it maps distinct elements of A into distinct elements of B (that is, if $a \neq b$, then $f(a) \neq f(b)$). A function $f : A \longrightarrow B$ is *onto* if every element of B is in the image of A under f (that is, if for every element $b \in B$ there is an element $a \in A$ with $f(a) = b$). A function $f : A \longrightarrow B$ is a *one-to-one correspondance* if it is both one-to-one and onto.

* * *

Intuitively, a computation is a way to "implement" a mathematical function $f : A \longrightarrow B$. A big part of the course is to build an understanding of what that intuition means.

Arbitrary functions between arbitrary sets A and B is too broad a class of functions to work with. Historically, researchers have looked at two classes of functions to study computation:

1. **Natural number functions** of the form

$$f : (\mathbb{N} \times \cdots \times \mathbb{N}) \longrightarrow \mathbb{N}$$

2. **Decision problems** of the form

$$d : D \longrightarrow \{1, 0\}$$

(where 1 can be interpreted as true and 0 as false). Decisison problems represent predicates on the domain D. For example, determining if a graph is planar.

The abacus model we saw last night is an example of computation model that can be used to define what it means for a natural number functions to be computable. For the next several lectures, we will consider decision problems instead.

The domain D of decision problems is usually constrained to be a set of *strings*.

Let Σ be a non-empty finite set we will call the *alphabet*. A *string over* Σ is a (possibly empty) finite sequence of elements of Σ, usually written $a_1 \cdots a_k$, where $a_i \in \Sigma$. For example, aaccabc is a string over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$. It is also a string over alphabet $\{\mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{d}\}$. We will use u, v, w to range over strings.

The length of $u = a_1 \ldots a_k$, written $|u|$, is k.

The empty string is written ϵ. It has length 0.

The set of all strings over Σ is denoted Σ^*. Note that this is an infinite set. (Why?)

If $u = a_1 \ldots a_k$ and $v = b_1 \cdots b_m$ are strings over Σ, then the *concatenation* uv is the string $a_1 \cdots a_k b_1 \cdots b_m$. Note that $\epsilon u = u \epsilon = u$ for every string u.

We define $u^0 = \epsilon$, $u^1 = u$, $u^2 = uu$, $u^3 = uuu$, etc.

One reason why decision problems are an interest class of problems to study is that they can be studied using sets of strings. To every decision problem

$$d : \Sigma^* \longrightarrow \{1, 0\}$$

you can associate a set of strings $S(d)$ defined by

$$S(d) = \{u \in \Sigma^* \mid d(u) = 1\}$$

that completely characterizes d. Indeed, given any set of strings A, you can construct a decision problem $D(A)$ by:

$$D(A)(u) = \begin{cases} 1 & \text{if } u \in A \\ 0 & \text{otherwise} \end{cases}$$

with the property that $D(S(d)) = d$. In other words, given any decision problem d, you can associate with d a set of strings A with the property that you can reconstruct d from A alone.

We will transform the problem of of determining whether a decision problem is computable into the problem of determining whether a set of strings is computable. But of course, what we will mean when we say that a set of strings A is computable is that the corresponding decision problem constructed from A as above is computable. Why do we do this? Because sets of strings are much easier to work with: they are sets, and we can manipulate sets in many different ways.

* * *

Because sets of strings are so important, let's give them a name. A *formal language* (usually called only a language) over alphabet Σ is a set of strings over Σ.

Since languages are sets, we inherit the usual set operations $A \cup B$, $A \cap B$, $\overline{A}$ (where the universe of A is taken to be Σ^*).

Because a language A is a set of strings specifically, we can also define more specific operations.

> $A \cdot B = \{uv \mid u \in A \text{ and } v \in B\}$, that is, the set of all strings obtained by concatenating a string of A and a string of B.
>
> $A^0 = \{\epsilon\}$
>
> $A^1 = A$
>
> $A^2 = A \cdot A$
>
> $A^3 = A \cdot A \cdot A$, etc

$$A^* = A^0 \cup A^1 \cup A^2 \cup A^3 \cup \cdots = \bigcup_{k\geq 0} A^k$$

The * operation is called the *Kleene star*.

Some properties that are easy to verify:

$$\varnothing \cdot A = A \cdot \varnothing = \varnothing$$

$$\{\epsilon\} \cdot A = A \cdot \{\epsilon\} = A$$

Example: $\Sigma = \{\mathtt{a}, \mathtt{b}\}$, and $A = \{\mathtt{aa}, \mathtt{bb}\}$. Then $A^* = \{\epsilon, \mathtt{aaaa}, \mathtt{aabb}, \mathtt{bbaa}, \mathtt{bbbb}, \mathtt{aaaaaa}, \mathtt{aaaabb}, \mathtt{aabbaa}, \mathtt{aabbbb}, \mathtt{bbaaaa}, \mathtt{bbaabb}, \mathtt{bbbbaa}, \mathtt{bbbbbb}, \ldots\}$.

This explains why I wrote Σ^* for the set of all strings: if we consider Σ as a set of strings, each of length 1, then Σ^* according to the above definition indeed gives the set of all strings over alphabet Σ.

The operations $\cup$, $\cdot$, and * are called the *regular operations*.

A language over alphabet $\Sigma = \{a_1, \ldots, a_k\}$ is *regular* if it can be obtained from the sets $\varnothing, \{\epsilon\}, \{a_1\}, \ldots, \{a_k\}$ and finitely many applications of the regular operations.

Example: consider the language of all even-length strings over alphabet $\{\mathtt{a}, \mathtt{b}\}$. It can be obtained as:

$$((\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\}) \cdot (\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\}))^*$$

Therefore, that language is regular.

Example: consider the language of all strings over $\{\mathtt{a}, \mathtt{b}, \mathtt{c}\}$ that start and end with an $\mathtt{a}$:

$$(\{\mathtt{a}\} \cdot (\{\mathtt{a}\} \cup \{\mathtt{b}\} \cup \{\mathtt{c}\})^* \cdot \{\mathtt{a}\}) \cup \{\mathtt{a}\}$$

Therefore, that language is also regular.

Regular languages form a very natural class of languages, and we will see soon that they arise out of an equally natural model of computation.

Some natural ways to form regular languages:

- The empty language is regular, pretty much by definition.
- Every singleton language (language with a single string) is regular. This is easy to see: language $\{a_1 \ldots a_k\}$ is obtained by taking $\{a_1\} \cup \cdots \cup \{a_k\}$.

- Every finite language (i.e., language with a finite number of strings) is regular. Again, this is easy to see. Say $A = \{u_1, \dots, u_k\}$. By the previous statement, each of $\{u_1\}$, ..., $\{u_k\}$ are regular. Therefore, their union is regular.

- if A and B are regular languages, then $A \cup B$ and $A \cdot B$ are regular languages. That follows directly from the definition.

- If A is regular, then A^* is regular. Again, this follows directly from the definition. This means, in particular, that Σ^*, the set of all strings over a given alphabet, is regular.

- If A and B are regular, then $A \cap B$ is regular. We'll show this is the case later, because we don't have enough techniques to show that just yet.

- If A is regular, then $\overline{A} = \Sigma^* - A$ is regular. Again, we don't have enough techniques to show that just yet.

- If A is regular, then *rev*(A), the set of all strings from A but in reverse (where the reverse of $a_1 \dots a_k$ is just $a_k \dots a_1$) is regular. We'll be able to show that next section.

You might be led to believe at this point that every language is regular. That's not true. The most natural non-regular language is probably the language of all *palindrome* strings over an alphabet Σ. A palindrome string is a string with the property that it and its reverse are equal, such as `aaa`, or `abba`. It's not obvious that this is not regular, and we don't have enough techniques to show that yet.

* * *

Regular expressions are a convenient notation for regular languages.

A regular expression over alphabet Σ is defined by the following syntax:

$$
\begin{aligned}
r ::= \quad & \mathtt{1} \\
& \mathtt{0} \\
& a \qquad \text{for every } a \in \Sigma \\
& (r_1\texttt{+}r_2) \\
& (r_1 r_2) \\
& (r_1^*)
\end{aligned}
$$

We usually drop parentheses, under the assumption that r_1^* binds tighter than concatenation $r_1 r_2$ which binds tighter than $r_1\texttt{+}r_2$. For example, `ab+ac` is a regular expression, as is `a(b+c)` and $\mathtt{a}^*\mathtt{(b+c)}^*$.

A regular expression r denotes a language $[\![r]\!]$ over Σ in the following way:

$$
\begin{aligned}
[\![\mathtt{1}]\!] &= \{\epsilon\} \\
[\![\mathtt{0}]\!] &= \varnothing \\
[\![a]\!] &= \{a\} \\
[\![r_1\texttt{+}r_2]\!] &= [\![r_1]\!] \cup [\![r_2]\!] \\
[\![r_1 r_2]\!] &= [\![r_1]\!] \cdot [\![r_2]\!] \\
[\![r_1^*]\!] &= [\![r_1]\!]^*
\end{aligned}
$$

For example:

$$
\begin{aligned}
[\![\mathtt{ab+ac}]\!] &= [\![\mathtt{ab}]\!] \cup [\![\mathtt{ac}]\!] \\
&= ([\![\mathtt{a}]\!] \cdot [\![\mathtt{b}]\!]) \cup ([\![\mathtt{a}]\!] \cdot [\![\mathtt{c}]\!]) \\
&= (\{\mathtt{a}\} \cdot \{\mathtt{b}\}) \cup (\{\mathtt{a}\} \cdot \{\mathtt{c}\}) \\
&= \{\mathtt{ab}\} \cup \{\mathtt{ac}\} \\
&= \{\mathtt{ab}, \mathtt{ac}\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathtt{a(b+c)}]\!] &= [\![\mathtt{a}]\!] \cdot [\![\mathtt{b+c}]\!] \\
&= [\![\mathtt{a}]\!] \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{c}]\!]) \\
&= \{\mathtt{a}\} \cdot (\{\mathtt{b}\} \cup \{\mathtt{c}\}) \\
&= \{\mathtt{a}\} \cdot (\{\mathtt{b}, \mathtt{c}\}) \\
&= \{\mathtt{ab}, \mathtt{ac}\}
\end{aligned}
$$

$$
\begin{aligned}
[\![\mathtt{a}^*\mathtt{(b+c)}^*]\!] &= [\![\mathtt{a}^*]\!] \cdot [\![\mathtt{(b+c)}^*]\!] \\
&= [\![\mathtt{a}]\!]^* \cdot [\![\mathtt{b+c}]\!]^* \\
&= \{\mathtt{a}\}^* \cdot ([\![\mathtt{b}]\!] \cup [\![\mathtt{c}]\!])^* \\
&= \{\mathtt{a}\}^* \cdot (\{\mathtt{b}\} \cup \{\mathtt{c}\})^* \\
&= \{\mathtt{a}\}^* \cdot \{\mathtt{b}, \mathtt{c}\}^*
\end{aligned}
$$

And thinking about this last set (which is difficult to write down), it is basically the set of all strings obtained by concatenating a sequence of `a`s (including none) to a sequence of `b`s and `c`s (in any order, including none). So `aaaaaa` is in this set, as is `aaaab`, `aaaabbbb`, `aaaabbbcbcbc`, etc.

Theorem: A language A is regular exactly if there is a regular expression r such that $[\![r]\!] = A$.

We can use regular expressions to show that if A is regular, then $rev(A) = \{rev(u) \mid u \in A\}$, where $rev(u)$ is the reverse of string u, is regular: since A is regular, there is a regular expression r with $[\![r]\!] = A$. We take this regular expression and transform it into regular expression $\widehat{r}$ as follow:

$$
\begin{aligned}
\widehat{\mathtt{1}} &= \mathtt{1} \\
\widehat{\mathtt{0}} &= \mathtt{0} \\
\widehat{a} &= a \\
\widehat{r_1\mathtt{+}r_2} &= \widehat{r_2}\mathtt{+}\widehat{r_1} \\
\widehat{r_1 r_2} &= \widehat{r_2} \cdot \widehat{r_1} \\
\widehat{r_1^*} &= (\widehat{r_1})^*
\end{aligned}
$$

Now, if $[\![r]\!] = A$, then $[\![\widehat{r}]\!] = rev(A)$, and therefore $rev(A)$ is regular.