# A Crash-Course on Notation

There are innumerable books available about compilers, automata theory and formal languages. Unfortunately, they often use their own notational conventions and their own symbols. This handout is meant to clarify some of the notation I will use. I apologise in advance that sometimes I will be a bit fuzzy…the problem is that often we want to have convenience in our mathematical definitions (to make them readable and understandable), but other times we need pedantic precision for actual programs.

### Characters and Strings

The most important concept in this module are strings. Strings are composed of ***characters***. While characters are surely a familiar concept, we will make one subtle distinction in this module. If we want to refer to concrete characters, like `a`, `b`, `c` and so on, we use a typewriter font. Accordingly if we want to refer to the concrete characters of my email address we shall write

`christian.urban@kcl.ac.uk`

If we also need to explicitly indicate the "space" character, we write ␣. For example

`hello␣world`

But often we do not care which particular characters we use. In such cases we use the italic font and write $a$, $b$, $c$ and so on for characters. Therefore if we need a representative string, we might write

$$abracadabra$$

In this string, we do not really care what the characters stand for, except we do care about the fact that for example the character $a$ is not equal to $b$ and so on. Why do I make this distinction? Because we often need to define functions using variables ranging over characters. We need to somehow say this is a variable, say $c$, ranging over characters, while this is the actual character `c`.

An ***alphabet*** is a (non-empty) finite set of characters. Often the letter $\Sigma$ is used to refer to an alphabet. For example the ASCII characters `a` to `z` form an alphabet. The digits 0 to 9 are another alphabet. The Greek letters $\alpha$ to $\omega$ also form an alphabet. If nothing else is specified, we usually assume the alphabet consists of just the lower-case letters $a$, $b$, …, $z$. Sometimes, however, we explicitly want to restrict strings to contain only the letters $a$ and $b$, for example. In this case we will state that the alphabet is the set $\{a, b\}$.

***Strings*** are lists of characters. Unfortunately, there are many ways how we can write down strings. In programming languages, they are usually written

© Christian Urban, King's College London, 2014, 2015, 2016, 2017, 2018

as `"hello"` where the double quotes indicate that we are dealing with a string. In typed programming languages, such as Scala, strings have a special type—namely `String` which is different from the type for lists of characters. This is because strings can be efficiently represented in memory, unlike lists. Since `String` and the type of lists of characters (`List[Char]`) are not the same, we need to explicitly coerce elements between the two types, for example

```
scala> "abc".toList
res01: List[Char] = List(a, b, c)
```

However, we do not want to do this kind of explicit coercion in our pencil-and-paper, everyday arguments. So in our (mathematical) definitions we regard strings as lists of characters and we will also write "$hello$" as list

$$[h, e, l, l, o] \qquad \text{or simply} \qquad hello$$

The important point is that we can always decompose such strings. For example, we will often consider the first character of a string, say $h$, and the "rest" of a string say "*ello*" when making definitions about strings. There are also some subtleties with the empty string, sometimes written as "" but also as the empty list of characters $[\,]$.[1]

Two strings, say $s_1$ and $s_2$, can be ***concatenated***, which we write as $s_1 @ s_2$. If we regard $s_1$ and $s_2$ as lists of characters, then @ is the list-append function. Suppose we are given two strings "*foo*" and "*bar*", then their concatenation, written "*foo*" @ "*bar*", gives "*foobar*". But as said above, we will often simplify our life and just drop the double quotes whenever it is clear we are talking about strings. So we will often just write *foo*, *bar*, *foobar* or *foo* @ *bar*.

Occasionally we will use the notation $a^n$ for strings, which stands for the string of $n$ repeated $a$s. So $a^n b^n$ is a string that has some number of $a$s followed by the same number of $b$s. A simple property of string concatenation is *associativity*, meaning

$$(s_1 @ s_2) @ s_3 = s_1 @ (s_2 @ s_3)$$

are always equal strings. The empty string behaves like a *unit element*, therefore

$$s \,@\, [] = [] \,@\, s = s$$

**Sets and Languages**

We will use the familiar operations $\cup$, $\cap$, $\subset$ and $\subseteq$ for sets. For the empty set we will either write $\varnothing$ or $\{\,\}$. The set containing the natural numbers 1, 2 and 3, for example, we will write with curly braces as

$$\{1, 2, 3\}$$

[1] In the literature you can also often find that $\varepsilon$ or $\lambda$ is used to represent the empty string.

The notation $\in$ means *element of*, so $1 \in \{1, 2, 3\}$ is true and $4 \in \{1, 2, 3\}$ is false. Note that the *list* $[1, 2, 3]$ is something different from the *set* $\{1, 2, 3\}$: in the former we care about the order and potentially several occurrences of a number; while with the latter we do not. Also sets can potentially have infinitely many elements, whereas lists cannot. For example the set of all natural numbers $\{0, 1, 2, \ldots\}$ is infinite. This set is often also abbreviated as $\mathbb{N}$. Lists can be very large, but they cannot contain infinitely many elements.

We can define sets by giving all their elements, for example $\{0, 1\}$ for the set containing just 0 and 1. But often we need to use ***set comprehensions*** to define sets. For example the set of all *even* natural numbers can be defined as

$$\{n \mid n \in \mathbb{N} \wedge n \text{ is even}\}$$

Set comprehensions consist of a "base set" (in this case all the natural numbers) and a predicate (here eveness).

Though silly, but the set $\{0, 1, 2\}$ could also be defined by the following set comprehension

$$\{n \mid n \in \mathbb{N} \wedge n^2 < 9\}$$

Can you see why this defines the set $\{0, 1, 2\}$? Notice that set comprehensions are quite powerful constructions. For example they could be used to define set union, set intersection and set difference:

$$\begin{aligned} A \cup B &\stackrel{\text{def}}{=} \{x \mid x \in A \vee x \in B\} \\ A \cap B &\stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \in B\} \\ A \backslash B &\stackrel{\text{def}}{=} \{x \mid x \in A \wedge x \notin B\} \end{aligned}$$

In general set comprehensions are of the form $\{a \mid P\}$ which stands for the set of all elements $a$ (from some set) for which some property $P$ holds.

For defining sets, we will also often use the notion of the "big union". An example is as follows:

$$\bigcup_{0 \leq n} \{n^2, n^2 + 1\} \tag{1}$$

which is the set of all squares and their immediate successors, so

$$\{0, 1, 2, 4, 5, 9, 10, 16, 17, \ldots\}$$

A big union is a sequence of unions which are indexed typically by a natural number. So the big union in (1) could equally be written as

$$\{0, 1\} \cup \{1, 2\} \cup \{4, 5\} \cup \{9, 10\} \cup \ldots$$

but using the big union notation is more concise.

As an aside: While this stuff about sets might all look trivial or even needlessly pedantic, *Nature* is never simple. If you want to be amazed how complicated sets can get, watch out for the last lecture just before Christmas where I want to convince you of the fact that some sets are more infinite than other sets. Yes, you read correctly, there can be sets that are "more infinite" than others. If you think this is obvious: say you have the infinite set $\mathbb{N}\backslash\{0\} = \{1,2,3,4,\ldots\}$ which is all the natural numbers except 0, and then compare it to the set $\{0,1,2,3,4,\ldots\}$ which contains the 0 and all other numbers. If you think, the second must be more infinite… well, then think again. Because the two infinite sets

$$\{1,2,3,4,\ldots\} \text{ and } \{0,1,2,3,4,\ldots\}$$

contain actually the same amount of elements. Does this make sense? Though this might all look strange, infinite sets will be a topic that is very relevant to the material of this module. It tells us what we can compute with a computer (actually algorithm) and what we cannot. But during the first 9 lectures we can go by without this "weird" stuff. End of aside.

Another important notion in this module are ***languages***, which are sets of strings. One of the main goals for us will be how to (formally) specify languages and to find out whether a string is in a language or not.[2] Note that the language containing the empty string `{""}` is not equal to $\varnothing$, the empty language (or empty set): The former contains one element, namely `""` (also written $[\,]$), but the latter does not contain any element.

For languages we define the operation of ***language concatenation***, written like in the string case as $A@B$:

$$A@B \stackrel{\text{def}}{=} \{s_1@s_2 \mid s_1 \in A \wedge s_2 \in B\} \tag{2}$$

Be careful to understand the difference: the @ in $s_1@s_2$ is string concatenation, while $A@B$ refers to the concatenation of two languages (or sets of strings). As an example suppose $A = \{ab, ac\}$ and $B = \{zzz, qq, r\}$, then $A\,@\,B$ is the language

$$\{abzzz, abqq, abr, aczzz, acqq, acr\}$$

Recall the properties for string concatenation. For language concatenation we have the following properties

| | |
|---|---|
| associativity: | $(A@B)@C = A@(B@C)$ |
| unit element: | $A\,@\,\{[]\} = \{[]\}\,@\,A = A$ |
| zero element: | $A\,@\,\varnothing = \varnothing\,@\,A = \varnothing$ |

Note the difference in the last two lines: the empty set behaves like 0 for multiplication and the set $\{[]\}$ like 1 for multiplication ($n * 1 = n$ and $n * 0 = 0$).

[2] You might wish to ponder whether this is in general a hard or easy problem, where hardness is meant in terms of Turing decidable, for example.

Using the operation of language concatenation, we can define a ***language power*** operation as follows:

$$
\begin{aligned}
A^0 &\stackrel{\text{def}}{=} \{[]\} \\
A^{n+1} &\stackrel{\text{def}}{=} A \,@\, A^n
\end{aligned}
$$

This definition is by recursion on natural numbers. Note carefully that the zero-case is not defined as the empty set, but the set containing the empty string. So no matter what the set $A$ is, $A^0$ will always be $\{[]\}$. (There is another hint about a connection between the @-operation and multiplication: How is $x^n$ defined in mathematics and what is $x^0$?)

Next we can define the ***star operation*** for languages: $A\star$ is the union of all powers of $A$, or short

$$A\star \stackrel{\text{def}}{=} \bigcup_{0\leq n} A^n \qquad (3)$$

This star operation is often also called *Kleene-star*. Unfolding the definition in (3) gives

$$A\star \stackrel{\text{def}}{=} A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots$$

which is equal to

$$A\star \stackrel{\text{def}}{=} \{[]\} \;\cup\; A \;\cup\; A@A \;\cup\; A@A@A \;\cup\; \ldots$$

We can see that the empty string is always in $A\star$, no matter what $A$ is. This is because $[] \in A^0$. To make sure you understand these definitions, I leave you to answer what $\{[]\}\star$ and $\varnothing\star$ are?

Recall that an alphabet is often referred to by the letter $\Sigma$. We can now write for the set of *all* strings over this alphabet as $\Sigma\star$. In doing so we also include the empty string as a possible string over $\Sigma$. So if $\Sigma = \{a, b\}$, then $\Sigma\star$ is

$$\{[], a, b, aa, ab, ba, bb, aaa, aab, aba, abb, baa, bab, \ldots\}$$

or in other words all strings containing $a$s and $b$s only, plus the empty string.