A Crash-Course on Notation

Characters and Strings

In this module we will often use ***characters***. While they are surely familiar, we will make one subtle distinction. If we want to refer to concrete characters, like `a`, `b` and so on, we use a typewriter font. So if we want to refer to the concrete characters of my email address we shall write

`christian.urban@kcl.ac.uk`

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

`hello␣world`

But often we do not care about which characters we use. In such cases we us the italic font and write a, b and so on. So if we need a representative string, we might write

$$abracadabra \qquad (1)$$

We do not really care what the characters stand for, except we do care about is that for example the character a is not equal to b.

An ***alphabet*** is a finite set of characters. Often the letter Σ 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. If nothing else is specified, we usually assume the alphabet consists of just the lower-case letters a, b, ..., z. Sometimes, however, we explicitly restrict strings to contain, for example, only the letters a and b. In this case we say 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 as "$hello$" where the double quotes indicate that we dealing with a string. But since, strings are lists of characters we could also write this string as

$$[h, e, l, l, o]$$

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 some subtleties with the empty string, sometimes written as "" but also as the empty list of characters $[\,]$. Two strings, for example s_1 and s_2, can be *concatenated*, which we write as $s_1 @ s_2$. Suppose we are given two strings "foo" and "bar", then their concatenation, writen "foo" @ "bar", gives "$foobar$". Often we will simplify our life and just drop the double quotes whenever it is clear we are talking about strings, writing as already in (1) just foo, bar, $foobar$ or $foo @ bar$.

Some simple properties of string concatenation hold. For example the concatenation operation is *associative*, 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

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

While for us strings are just lists of characters, programming languages often differentiate between the two concepts. In Scala, for example, there is the type of `String` and the type of lists of characters, `List[Char]`. They are not the same and we need to explicitly coerce elements between the two types, for example

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

Sets and Languages

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

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

The notation $\in$ means *element of*, so $1 \in \{1, 2, 3\}$ is true and $3 \in \{1, 2, 3\}$ is false. Sets can potentially have infinitely many elements. For example the set of all natural numbers $\{0, 1, 2, \ldots\}$ is infinite. This set is often also abbreviated as $\mathbb{N}$. We can define sets by giving all elements, like $\{0, 1\}$, but also by ***set comprehensions***. For example the set of all even natural numbers can be defined as

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

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

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

Notice that set comprehensions could be used to define set union, intersection and difference:

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

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

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

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 (2) 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.

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

For languages we define the operation of ***language concatenation***, written $A@B$:

$$A@B \stackrel{\text{def}}{=} \{s_1@s_2 \mid s_1 \in A \land s_2 \in B\} \tag{3}$$

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

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

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: the empty set behaves like 0 for multiplication and the set $\{[]\}$ like 1 for multiplication.

Following the 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 induction 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 and what is x^0?)

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

$$A^* \stackrel{\text{def}}{=} \bigcup_{0 \leq n} A^n$$

Unfolding this definition

$$A^0 \cup A^1 \cup A^2 \cup A^3 \cup \ldots$$

which is equal to

$$\{[]\} \,\cup\, A \,\cup\, A@A \,\cup\, A@A@A \,\cup\, \ldots$$

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

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

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

or in other words all strings containing as and bs only.