

## Handout 1

This course is about the processing of strings. Lets start with what we mean by *strings*. Strings (they are also sometimes referred to as *words*) are lists of characters drawn from an *alphabet*. If nothing else is specified, we usually assume the alphabet consists of just the lower-case letters  $a, b, \dots, z$ . 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  $\{a, b\}$ .

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. Essentially, strings are lists of characters which can be written for example as follows

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

The important point is that we can always decompose 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 gives "foobar".

We often need to talk about sets of strings. For example the set of all strings over the alphabet  $\{a, \dots, z\}$  is

$$\{ "", "a", "b", "c", \dots, "z", "aa", "ab", "ac", \dots, "aaa", \dots \}$$

Any set of strings, not just the set-of-all-strings, is often called a *language*. The idea behind this choice of terminology is that if we enumerate, say, all words/strings from a dictionary, like

$$\{ "the", "of", "milk", "name", "antidisestablishmentarianism", \dots \}$$

then we have essentially described the English language, or more precisely all strings that can be used in a sentence of the English language. French would be a different set of strings, and so on. In the context of this course, a language might not necessarily make sense from a natural language point of view. For example the set of all strings shown above is a language, as is the empty set (of strings). The empty set of strings is often written as  $\emptyset$  or  $\{ \}$ . Note that there is a difference between the empty set, or empty language, and the set that contains only the empty string  $\{ "" \}$ : the former has no elements, whereas the latter has one element.

As seen, there are languages which contain infinitely many strings, like the set of all strings. The "natural" languages like English, French and so on contain many but only finitely many strings (namely the ones listed in a good dictionary). It might be therefore be surprising that the language consisting of

all email addresses is infinite provided we assume it is defined by the regular expression<sup>1</sup>

$$([a-zA-Z0-9_-]+)@([a-zA-Z0-9_-]+)\.([a-z.]{2,6})$$

The reason is that for example before the @-sign there can be any string you want assuming it is made up from letters, digits, underscores, dots and hyphens—clearly there are infinitely many of those. Similarly the string after the @-sign can be any string. However, this does not mean that every string is an email address. For example

*foo@bar.c*

is not, because the top-level-domains must be of length of at least two. (Note that there is the convention that uppercase letters are treated in email-addresses as if they were lower-case.)

Before we expand on the topic of regular expressions, let us review some operations on sets. We will use capital letters  $A$ ,  $B$ , ... to stand for sets of strings. The union of two sets is written as usual as  $A \cup B$ . We also need to define the operation of *concatenating* two sets of strings. This can be defined as

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

which essentially means take the first string from the set  $A$  and concatenate it with every string in the set  $B$ , then take the second string from  $A$  do the same and so on. You might like to think about what this definition means in case  $A$  or  $B$  is the empty set.

We also need to define the power of a set, written as  $A^n$  with  $n$  being a natural number. This is defined inductively as follows

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

Finally we need the *star* of a set of strings, written  $A^*$ . This is defined as the union of every power of  $A^n$  with  $n \geq 0$ . The mathematical notation for this operation is

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

This definition implies that the star of a set  $A$  contains always the empty string (that is  $A^0$ ), one copy of every string in  $A$  (that is  $A^1$ ), two copies in  $A$  (that is  $A^2$ ) and so on. In case  $A = \{ "a" \}$  we therefore have

$$A^* = \{ "", "a", "aa", "aaa", \dots \}$$

---

<sup>1</sup>See <http://goo.gl/5LoVX7>

Be aware that these operations sometimes have quite non-intuitive properties, for example

$$\begin{array}{lll}
 A \cup \emptyset = A & A @ B \neq B @ A & \emptyset^* = \{'''\} \\
 A \cup A = A & A @ \emptyset = \emptyset @ A = \emptyset & \{'''\}^* = \{'''\} \\
 A \cup B = B \cup A & A @ \{'''\} = \{'''\} @ A = A & A^* = \{'''\} \cup A \cdot A^*
 \end{array}$$

*Regular expressions* are meant to conveniently describe languages...at least languages we are interested in in Computer Science. For example there is no convenient regular expression for describing the English language short of enumerating all English words. But they seem useful for describing all permitted email addresses, as seen above.

Regular expressions are given by the following grammar:

$r ::= \emptyset$	null
$  \epsilon$	empty string / "" / []
$  c$	single character
$  r_1 \cdot r_2$	sequence
$  r_1 + r_2$	alternative / choice
$  r^*$	star (zero or more)

Because we overload our notation there are some subtleties you should be aware of. The letter  $c$  stands for any character from the alphabet at hand. Second, we will use parentheses to disambiguate regular expressions. For example we will write  $(r_1 + r_2)^*$ , which is different from, say  $r_1 + (r_2)^*$ . The former means roughly zero or more times  $r_1$  or  $r_2$ , while the latter means  $r_1$  or zero or more times  $r_2$ . We should also write  $(r_1 + r_2) + r_3$ , which is different from the regular expression  $r_1 + (r_2 + r_3)$ , but in case of  $+$  and  $\cdot$  we actually do not care about the order and just write  $r_1 + r_2 + r_3$ , or  $r_1 \cdot r_2 \cdot r_3$ , respectively. The reasons for this will become clear shortly. In the literature you will often find that the choice  $r_1 + r_2$  is written as  $r_1 | r_2$ . Also following the convention in the literature, we will in case of  $\cdot$  even often omit it all together. For example the regular expression for email addresses shown above is meant to be of the form

$$([\dots])^+ \cdot @ \cdot ([\dots])^+ \cdot \dots$$

meaning first comes a name (specified by the regular expression  $([\dots])^+$ ), then an @-sign, then a domain name (specified by the regular expression  $([\dots])^+$ ), then a top-level domain. Similarly if we want to specify the regular expression for the string "hello" we should write

$$h \cdot e \cdot l \cdot l \cdot o$$

but often just write *hello*.

Another source of confusion might arise from the fact that we use the term *regular expressions* for the ones used in "theory" and also the ones in "practice". In this course we refer by default to the regular expressions defined by the

grammar above. In “practice” we often use  $r^+$  to stand for one or more times,  $\backslash d$  to stand for a digit,  $r^?$  to stand for an optional regular expression, or ranges such as  $[a - z]$  to stand for any lower case letter from  $a$  to  $z$ . They are however mere convenience as they can be seen as shorthand for

$$\begin{aligned} r^+ &\mapsto r \cdot r^* \\ r^? &\mapsto \epsilon + r \\ \backslash d &\mapsto 0 + 1 + 2 + \dots + 9 \\ [a - z] &\mapsto a + b + \dots + z \end{aligned}$$

We will see later that the *not*-regular-expression can also be seen as convenience. This regular expression is supposed to stand for every string *not* matched by a regular expression. We will write such not-regular-expressions as  $\sim r$ . While being “convenience” it is often not so clear what the shorthand for these kind of not-regular-expressions is. Try to write down the regular expression which can match any string except “hello” and “world”. It is possible in principle, but often it is easier to just include  $\sim r$  in the definition or regular expressions. Whenever we do so, we will state it explicitly.

So far we have only considered informally what the *meaning* of a regular expression is. To do so more formally we will associate with every regular expression a set of strings that is supposed to be are matched by this regular expression. This can be defined recursively as follows

$$\begin{aligned} L(\emptyset) &\stackrel{\text{def}}{=} \{ \} \\ L(\epsilon) &\stackrel{\text{def}}{=} \{ "" \} \\ L(c) &\stackrel{\text{def}}{=} \{ "c" \} \\ L(r_1 + r_2) &\stackrel{\text{def}}{=} L(r_1) \cup L(r_2) \\ L(r_1 \cdot r_2) &\stackrel{\text{def}}{=} \{ s_1 @ s_2 \mid s_1 \in L(r_1) \wedge s_2 \in L(r_2) \} \\ L(r^*) &\stackrel{\text{def}}{=} \bigcup_{n \geq 0} L(r)^n \end{aligned}$$

This means we can now precisely state what the meaning, for example, of the regular expression  $h \cdot e \cdot l \cdot l \cdot o$  is, namely  $L(h \cdot e \cdot l \cdot l \cdot o) = \{ "hello" \}$ ...as expected. Similarly if we have the choice-regular-expression  $a + b$ , its meaning is  $L(a + b) = \{ "a", "b" \}$ , namely the only two strings which can possibly be matched by this choice. You can now also conclude why we do not make a difference between the different regular expressions  $(r_1 + r_2) + r_3$  and  $r_1 + (r_2 + r_3)$ ....they are not the same regular expression, but have the same meaning.

The point of the definition of  $L$  is that we can now precisely specify when a string  $s$  is matched by a regular expression  $r$ , namely only when  $s \in L(r)$ . In fact we will write a program *match* that takes any string  $s$  and any regular expression  $r$  as argument and returns *yes*, if  $s \in L(r)$  and *no*, if  $s \notin L(r)$ . We leave this for the next lecture.

```

1 // A crawler which checks whether there
2 // are problems with links in web-pages
3
4 import io.Source
5 import scala.util.matching.Regex
6 import scala.util._
7
8 // gets the first 10K of a web-page
9 def get_page(url: String) : String = {
10   Try(Source.fromURL(url).take(10000).mkString) getOrElse
11     { println(s" Problem with: $url"); ""}
12 }
13
14 // regex for URLs
15 val http_pattern = """"https?:/[^\"]*\"""".r
16
17 // drops the first and last character from a string
18 def unquote(s: String) = s.drop(1).dropRight(1)
19
20 def get_all_URLs(page: String) : Set[String] = {
21   http_pattern.findAllIn(page).map(unquote).toSet
22 }
23
24 // naive version - searches until a given depth
25 // visits pages potentially more than once
26 def crawl(url: String, n: Int) : Unit = {
27   if (n == 0) ()
28   else {
29     println(s"Visiting: $n $url")
30     for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
31   }
32 }
33
34 // starting URL for the crawler
35 val startURL = """"http://www.inf.kcl.ac.uk/staff/urbanc/\"""
36 //val startURL = """"http://www.inf.kcl.ac.uk/staff/mml/\"""
37
38 crawl(startURL, 2)

```

Figure 1: Scala code for a web-crawler that can detect broken links in a web-page. It uses the regular expression `http_pattern` in Line 15 for recognising URL-addresses. It finds all links using the library function `findAllIn` in Line 21.

```

1 // This version of the crawler only
2 // checks links in the "domain" urbanc
3
4 import io.Source
5 import scala.util.matching.Regex
6 import scala.util._
7
8 // gets the first 10K of a web-page
9 def get_page(url: String) : String = {
10   Try(Source.fromURL(url).take(10000).mkString) getOrElse
11     { println(s" Problem with: $url"); ""}
12 }
13
14 // regexes for URLs and "my" domain
15 val http_pattern = """"\https?://[^\"]*\.""".r
16 val my_urls = """"urbanc""".r
17
18 def unquote(s: String) = s.drop(1).dropRight(1)
19
20 def get_all_URLs(page: String) : Set[String] = {
21   http_pattern.findAllIn(page).map(unquote).toSet
22 }
23
24 def crawl(url: String, n: Int) : Unit = {
25   if (n == 0) ()
26   else if (my_urls.findFirstIn(url) == None) ()
27   else {
28     println(s"Visiting: $n $url")
29     for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
30   }
31 }
32
33 // staring URL for the crawler
34 val startURL = """"http://www.inf.kcl.ac.uk/staff/urbanc/""""
35
36 // can now deal with depth 3 and beyond
37 crawl(startURL, 4)

```

Figure 2: A version of the web-crawler which only follows links in “my” domain—since these are the ones I am interested in to fix. It uses the regular expression `my_urls` in Line 16. The main change is in Line 26 where there is a test whether URL is in “my” domain or not.

```

1 // This version of the crawler that also
2 // "harvests" email addresses from webpages
3
4 import io.Source
5 import scala.util.matching.Regex
6 import scala.util._
7
8 def get_page(url: String) : String = {
9   Try(Source.fromURL(url).take(10000).mkString) getOrElse
10    { println(s" Problem with: $url"); ""}
11 }
12
13 // regexes for URLs, for "my" domain and for email addresses
14 val http_pattern = """\\"https?://[^\"]*\\"""".r
15 val my_urls = """urbanc""".r
16 val email_pattern = """([a-z0-9_\. -]+)@([\da-z\.-]+)\.([a-z\.] {2,6})""".r
17
18 // The regular expression for emails comes from:
19 //   http://net.tutsplus.com/tutorials/other/8-regular-expressions-you-should-know/
20
21 def unquote(s: String) = s.drop(1).dropRight(1)
22
23 def get_all_URLs(page: String) : Set[String] = {
24   http_pattern.findAllIn(page).map(unquote).toSet
25 }
26
27 def crawl(url: String, n: Int) : Unit = {
28   if (n == 0) ()
29   //else if (my_urls.findFirstIn(url) == None) ()
30   else {
31     println(s"Visiting: $n $url")
32     val page = get_page(url)
33     println(email_pattern.findAllIn(page).mkString("\n"))
34     for (u <- get_all_URLs(page)) crawl(u, n - 1)
35   }
36 }
37
38 // starting URL for the crawler
39 val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc/"""
40
41 crawl(startURL, 3)

```

Figure 3: A small email harvester—whenever we download a web-page, we also check whether it contains any email addresses. For this we use the regular expression `email_pattern` in Line 17. The main change is in Lines 33 and 34 where all email addresses that can be found in a page are printed.