# Handout 1

This module is about text processing, be it for web-crawlers, compilers, dictionaries, DNA-data and so on. When looking for a particular string in a large text we can use the Knuth-Morris-Pratt algorithm, which is currently the most efficient general string search algorithm. But often we do *not* look for just a particular string, but for string patterns. For example in programming code we need to identify what are the keywords, what are the identifiers etc. Also often we face the problem that we are given a string (for example some user input) and want to know whether it matches a particular pattern. For example for excluding some user input that would otherwise have nasty effects on our program (crashing or going into an infinite loop, if not worse). ***Regular expressions*** help with conveniently specifying such patterns.

The idea behind regular expressions is that they are a simple method for describing languages (or sets of strings)...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 for example email addresses.[1] Consider the following regular expression

```
[a-z0-9_.-]+ @ [a-z0-9.-]+ . [a-z.]{2,6}          (1)
```

where the first part matches one or more lowercase letters (`a-z`), digits (`0-9`), underscores, dots or hyphens. The `+` ensures the "one or more". Then comes the `@`-sign, followed by the domain name which must be one or more lowercase letters, digits, underscores, dots or hyphens. Note there cannot be an underscore in the domain name. Finally there must be a dot followed by the toplevel domain. This toplevel domain must be 2 to 6 lowercase letters including the dot. Example strings which follow this pattern are:

```
niceandsimple@example.com
very.common@example.org
a.little.lengthy.but.fine@dept.example.co.uk
other.email-with-dash@example.ac.uk
```

But for example the following two do not:

```
user@localserver
disposable.style.email.with+symbol@example.com
```

Many programming language offer libraries that can be used to validate such strings against regular expressions, like the one for email addresses in (1). There are some common, and I am sure very familiar, ways how to construct regular expressions. For example in Scala we have

| | |
|---|---|
| `re*` | matches 0 or more occurrences of preceding expression |

[1] See "8 Regular Expressions You Should Know" `http://goo.gl/5LoVX7`

| | |
|---|---|
| `re+` | matches 1 or more occurrences of preceding expression |
| `re?` | matches 0 or 1 occurrence of preceding expression |
| `re{n}` | matches exactly `n` number of occurrences |
| `re{n,m}` | matches at least `n` and at most `m` occurences of the preceding expression |
| `[...]` | matches any single character inside the brackets |
| `[^...]` | matches any single character not inside the brackets |
| `..-..` | character ranges |
| `\d` | matches digits; equivalent to `[0-9]` |

With this you can figure out the purpose of the regular expressions in the web-crawlers shown Figures 1, 2 and 3. Note the regular expression for http-addresses in web-pages:

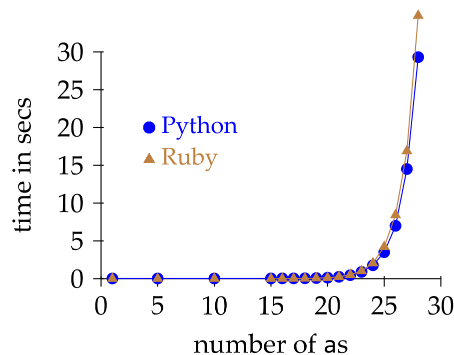
```
"https?://[^"]*"
```

It specifies that web-addresses need to start with a double quote, then comes `http` followed by an optional `s` and so on. Usually we would have to escape the double quotes in order to make sure we interpret the double quote as character, not as double quote for a string. But Scala's trick with triple quotes allows us to omit this kind of escaping. As a result we can just write:

```
""""https?://[^"]*"""".r
```

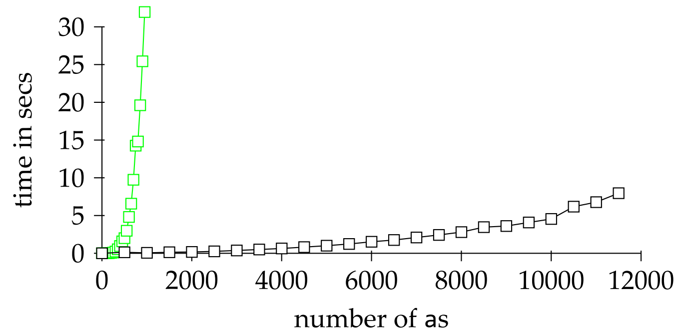
Not also that the convention in Scala is that `.r` converts a string into a regular expression. I leave it to you to ponder whether this regular expression really captures all possible web-addresses.

Regular expressions were introduced by Kleene in the 1950ies and they have been object of intense study since then. They are nowadays pretty much ubiquitous in computer science. I am sure you have come across them before. Why on earth then is there any interest in studying them again in depth in this module? Well, one answer is in the following graph about regular expression matching in Python and in Ruby.



This graph shows that Python needs approximately 29 seconds in order to find out that a string of 28 `a`s matches the regular expression `[a?]{28}[a]{28}`. Ruby is even slightly worse.[2] Admittedly, this regular expression is carefully chosen to exhibit this exponential behaviour, but similar ones occur more often than one wants in "real life". They are sometimes called *evil regular expressions* because they have the potential to make regular expression matching engines topple over, like in Python and Ruby. The problem is that this can have some serious consequences, for example, if you use them in your web-application, because hackers can look for these instances where the matching engine behaves badly and mount a nice DoS-attack against your application.

It will be instructive to look behind the "scenes"to find out why Python and Ruby (and others) behave so badly when matching with evil regular expressions. But we will also look at a relatively simple algorithm that solves this problem much better than Python and Ruby do…actually it will be two versions of the algorithm: the first one will be able to process strings of approximately 1,000 `a`s in 30 seconds, while the second version will even be able to process up to 12,000 in less than 10(!) seconds, see the graph below:




## Basic Regular Expressions

The regular expressions shown above we will call ***extended regular expressions***. The ones we will mainly study are *basic regular expressions*, which by convention we will just call regular expressions, if it is clear what we mean. The attraction of (basic) regular expressions is that many features of the extended one are just syntactic suggar. (Basic) regular expressions are defined by the following grammar:

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

[2] In this example Ruby uses the slightly different regular expression `a?a?a?...a?a?aaa...aa`, where the `a?` and `a` each occur $n$ times.

Because we overload our notation, there are some subtleties you should be aware of. First, when regular expressions are referred to then $\varnothing$ does not stand for the empty set: it is a particular pattern that does not match any string. Similarly, in the context of regular expressions, $\epsilon$ does not stand for the empty string (as in many places in the literature) but for a pattern that matches the empty string. Second, the letter $c$ stands for any character from the alphabet at hand. Again in the context of regular expressions, it is a particular pattern that can match the specified string. Third, you should also be careful with the our overloading of the star: assuming you have read the handout about our basic mathematical notation, you will see that in the context of languages (sets of strings) the star stands for an operation on languages. While $r^*$ stands for a regular expression, the operation on sets is defined as

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

We will use parentheses to disambiguate regular expressions. Parentheses are not really part of a regular expression, and indeed we do not need them in our code because there the tree structure is always clear. But for writing them down in a more mathematical fashion, parentheses will be helpful. 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 $r_2$. This will turn out are two different pattern, which match in general different strings. 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 $r_1 \mid r_2$ or $r_1 \mid\mid r_2$. Also following the convention in the literature, we will often omit the $\cdot$ all together. This is to make some concrete regular expressions more readable. For example the regular expression for email addresses shown in (1) would look like

```
[...]+ · @ · [...]+ · . · [...]{2,6}
```

which is much less readable than (1). Similarly for the regular expression that matches the string *hello* we should write

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

but often just write *hello*.

If you prefer to think in terms of the implementation of regular expressions in Scala, the constructors and classes relate as follows

| | | |
|---|---|---|
| $\varnothing$ | $\mapsto$ | `NULL` |
| $\epsilon$ | $\mapsto$ | `EMPTY` |
| $c$ | $\mapsto$ | `CHAR(c)` |
| $r_1 + r_2$ | $\mapsto$ | `ALT(r1, r2)` |
| $r_1 \cdot r_2$ | $\mapsto$ | `SEQ(r1, r2)` |
| $r^*$ | $\mapsto$ | `STAR(r)` |

A source of confusion might arise from the fact that we use the term *basic regular expression* for the regular expressions used in "theory" and defined above, and *extended regular expression* for the ones used in "practice", for example Scala. If runtime is not of an issue, then the latter can be seen as some syntactic sugar of the former. Fo example we could replace

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

## The Meaning of Regular Expressions

So far we have only considered informally what the *meaning* of a regular expression is. This is no good for specifications of what algorithms are supposed to do or which problems they are supposed to solve.

To do so more formally we will associate with every regular expression a language, or set of strings, that is supposed to be matched by this regular expression. To understand what is going on here it is crucial that you have also read the handout about our basic mathematical notations.

The meaning of a regular expression can be defined recursively as follows

$$\begin{array}{rcl} L(\varnothing) & \stackrel{\text{def}}{=} & \varnothing \\ L(\epsilon) & \stackrel{\text{def}}{=} & \{[]\} \\ L(c) & \stackrel{\text{def}}{=} & \{\text{"}c\text{"}\} \\ L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) \\ L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) @ L(r_2) \\ L(r^*) & \stackrel{\text{def}}{=} & (L(r))^* \end{array}$$

As a result 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 $L(h \cdot e \cdot l \cdot l \cdot o) = \{\text{"}hello\text{"}\}$...as expected. Similarly if we have the choice-regular-expression $a + b$, its meaning is $L(a + b) = \{\text{"}a\text{"}, \text{"}b\text{"}\}$, namely the only two strings which can possibly be matched by this choice. You can now also see 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 use it to 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 are
2  // dead 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 of crawl - 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 // some starting URLs for the crawler
35 val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc"""
36 //val startURL = """http://www.inf.kcl.ac.uk/staff/mcburney"""
37
38 crawl(startURL, 2)
```

Figure 1: The Scala code for a simple web-crawler that checks for 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.

```scala
// This version of the crawler only
// checks links in the "domain" urbanc

import io.Source
import scala.util.matching.Regex
import scala.util._

// gets the first 10K of a web-page
def get_page(url: String) : String = {
  Try(Source.fromURL(url).take(10000).mkString) getOrElse
    { println(s"  Problem with: $url"); ""}
}

// regexes for URLs and "my" domain
val http_pattern = """"https?://[^"]*"""".r
val my_urls = """urbanc""".r

def unquote(s: String) = s.drop(1).dropRight(1)

def get_all_URLs(page: String) : Set[String] = {
  http_pattern.findAllIn(page).map(unquote).toSet
}

def crawl(url: String, n: Int) : Unit = {
  if (n == 0) ()
  else if (my_urls.findFirstIn(url) == None) {
    println(s"Visiting: $n $url")
    get_page(url); ()
  }
  else {
    println(s"Visiting: $n $url")
    for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
  }
}

// starting URL for the crawler
val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc"""

// can now deal with depth 3 and beyond
crawl(startURL, 3)
```

Figure 2: A version of the web-crawler that 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 to check for my name in the links. The main change is in Lines 26–29 where there is a test whether URL is in “my” domain or not.

```scala
// This version of the crawler that also
// "harvests" email addresses from webpages

import io.Source
import scala.util.matching.Regex
import scala.util._

def get_page(url: String) : String = {
  Try(Source.fromURL(url).take(10000).mkString) getOrElse
    { println(s"  Problem with: $url"); ""}
}

// regexes for URLs, for "my" domain and for email addresses
val http_pattern = """"https?://[^"]*"""".r
val my_urls = """urbanc""".r
val email_pattern = """([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})""".r

def unquote(s: String) = s.drop(1).dropRight(1)

def get_all_URLs(page: String) : Set[String] = {
  http_pattern.findAllIn(page).map(unquote).toSet
}

def print_str(s: String) =
  if (s == "") () else println(s)

def crawl(url: String, n: Int) : Unit = {
  if (n == 0) ()
  else {
    println(s"Visiting: $n $url")
    val page = get_page(url)
    print_str(email_pattern.findAllIn(page).mkString("\n"))
    for (u <- get_all_URLs(page).par) crawl(u, n - 1)
  }
}

// staring URL for the crawler
val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc"""

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 16. The main change is in Line 32 where all email addresses that can be found in a page are printed.

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 $a$, $b$, ..., $z$. 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 $\{a, \ldots z\}$ is

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

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

$$\{\text{"the"}, \text{"of"}, \text{"milk"}, \text{"name"}, \text{"antidisestablishmentarianism"}, \ldots\}$$

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 $\varnothing$ or $\{\,\}$. Note that there is a difference between the empty set, or empty language, and the set that contains only the empty string $\{\text{""}\}$: the former has no elements, whereas the latter has one element.

Before we expand on the topic of regular expressions, let us review some operations on sets. We will use capital letters $A, B, \ldots$ 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

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

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 of strings, written as $A^n$ with $n$ being a natural number. This is defined inductively as follows

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

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

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

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

$$\begin{array}{ccc} A \cup \varnothing = A & A @ B \neq B @ A & \varnothing^* = \{""\} \\ A \cup A = A & A @ \varnothing = \varnothing @ A = \varnothing & \{""\}^* = \{""\} \\ A \cup B = B \cup A & A @ \{""\} = \{""\} @ A = A & A^\star = \{""\} \cup A \cdot A^* \end{array}$$

## My Fascination for Regular Expressions