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, like *abc* 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* just look for a particular string, but for string patterns. For example in program code we need to identify what are the keywords, what are the identifiers etc. A pattern for identifiers could be stated as: they start with a letter, followed by zero or more letters, numbers and underscores. 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—be it an email address, for example. In this way we can exclude user input that would otherwise have nasty effects on our program (crashing it or making it go 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 simple 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 and hyphens. The `+` at the end of the brackets 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.org
very.common@example.co.uk
a.little.lengthy.but.fine@dept.example.ac.uk
other.email-with-dash@example.edu
```

But for example the following two do not

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

according to the regular expression we specified in (1). Whether this is intended or not is a different question (the second email above is actually an acceptable email address acording to the RFC 5322 standard for email addresses).

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

As mentioned above, identifiers, or variables, in program code are often required to satisfy the constraints that they start with a letter and then can be followed by zero or more letters or numbers and also can include underscores, but not as the first character. Such identifiers can be recognised with the regular expression

```
[a-zA-Z] [a-zA-Z0-9_]*
```

Possible identifiers that match this regular expression are `x`, `foo`, `foo_bar_1`, `A_very_42_long_object_name`, but not `_i` and also not `4you`.

Many programming languages offer libraries that can be used to validate such strings against regular expressions. Also there are some common, and I am sure very familiar, ways of how to construct regular expressions. For example in Scala we have a library implementing the following regular expressions:

`re*`	matches 0 or more occurrences of preceding expression
`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 of preceding expression
`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]`
`.`	matches every character except newline
`(re)`	groups regular expressions and remembers matched text

With this table you can figure out the purpose of the regular expressions in the web-crawlers shown Figures 1, 2 and 3.[2] Note, however, the regular expression for http-addresses in web-pages in Figure 1, Line 15, is intended to be

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

It specifies that web-addresses need to start with a double quote, then comes `http` followed by an optional `s` and so on until the closing double quote comes. 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
```

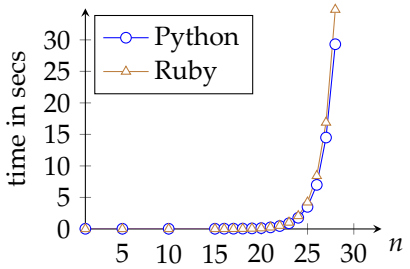
Note also that the convention in Scala is that `.r` converts a string into a regular expression. I leave it to you to ponder whether this regular expression really captures all possible web-addresses.

[2] There is an interesting twist in the web-scraper where `re*?` is used instead of `re*`.

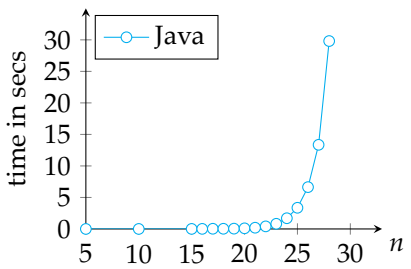
Why Study Regular Expressions?

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. There are many libraries implementing regular expressions. I am sure you have come across them before (remember PRA?). Why on earth then is there any interest in studying them again in depth in this module? Well, one answer is in the following two graphs about regular expression matching in Python, Ruby and Java.

Graph: `a?{n}a{n}` and strings $\underbrace{\texttt{a}\ldots\texttt{a}}_{n}$



Graph: `(a*)*b` and strings $\underbrace{\texttt{a}\ldots\texttt{a}}_{n}$



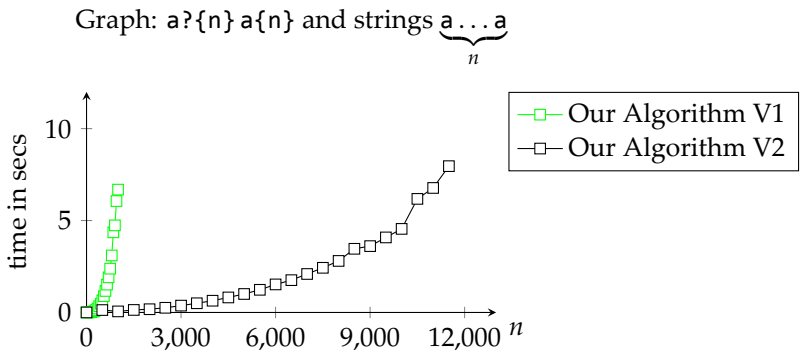
This first graph shows that Python needs approximately 29 seconds for finding out whether a string of 28 as matches the regular expression `a?{28}a{28}`. Ruby is even slightly worse.[3] Simlarly, Java needs approximately 30 seconds to find out that the regular expression `(a*)*b` does not match strings of 28 as. Admittedly, these regular expressions are carefully chosen to exhibit this exponential behaviour, but similar ones occur more often than one wants in "real life". For example, on 20 July 2016 a similar regular expression brought the webpage Stack Exchange to its knees:

`http://stackstatus.net/post/147710624694/outage-postmortem-july-20-2016`

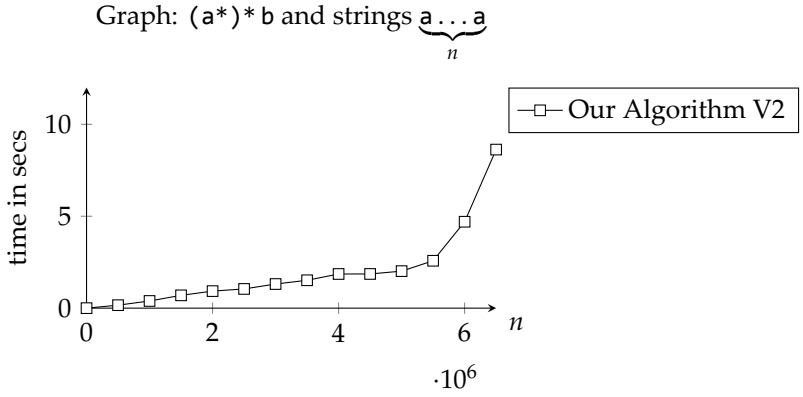
Such troublesome regular expressions are sometimes called *evil regular expressions* because they have the potential to make regular expression matching engines to topple over, like in Python, Ruby and Java. This 'toopling over' is also sometimes called *catastrophic backtracking*. The problem with evil regular expressions is that they can have some serious consequences, for example, if you use them in your web-application. The reason is that hackers can look for these instances where the matching engine behaves badly and then mount a nice DoS-attack against your application. These attacks are already have their own name: *Regular Expression Denial of Service Attacks (ReDoS)*.

[3] 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. More such test cases can be found at `http://www.computerbytesman.com/redos/`.

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 as 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:



And in the case of the regular expression (a*)*b and strings of as we will beat Java by factor of approximately 1,000,000 (note the scale on the x-axis).



Basic Regular Expressions

The regular expressions shown earlier for Scala, we will call *extended regular expressions*. The ones we will mainly study in this module 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 ones are just syntactic sugar. (Basic) regular expressions are defined by the following grammar:

$$
\begin{array}{lll}
r ::= & \mathbf{0} & \text{null language} \\
| & \mathbf{1} & \text{empty string / "" / []} \\
| & c & \text{single character} \\
| & r_1 + r_2 & \text{alternative / choice} \\
| & r_1 \cdot r_2 & \text{sequence} \\
| & r^* & \text{star (zero or more)}
\end{array}
$$

Because we overload our notation, there are some subtleties you should be aware of. When regular expressions are referred to, then **0** (in bold font) does not stand for the number zero: rather it is a particular pattern that does not match any string. Similarly, in the context of regular expressions, **1** does not stand for the number one but for a regular expression that matches the empty string. 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 character. You should also be careful with 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. Here r^* stands for a regular expression, which is different from the operation on sets is defined as

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

Note that this expands to

$$A\star \stackrel{\text{def}}{=} A^0 \cup A^1 \cup A^2 \cup A^3 \cup A^4 \cup \ldots$$

which is equivalent to

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

Remember that A^0 is always the set containing the empty string.

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 of regular expressions 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 to be two different patterns, 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, often our **0** and **1** are written $\varnothing$ and ϵ, respectively. 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

$$\texttt{[...]+} \cdot \texttt{@} \cdot \texttt{[...]+} \cdot \texttt{.} \cdot \texttt{[...]\{2,6\}}$$

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[4]

$$\begin{array}{rcl} \mathbf{0} & \mapsto & \texttt{ZERO} \\ \mathbf{1} & \mapsto & \texttt{ONE} \\ c & \mapsto & \texttt{CHAR(c)} \\ r_1 + r_2 & \mapsto & \texttt{ALT(r1, r2)} \\ r_1 \cdot r_2 & \mapsto & \texttt{SEQ(r1, r2)} \\ r^* & \mapsto & \texttt{STAR(r)} \end{array}$$

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 in Scala. If runtime is not an issue, then the latter can be seen as syntactic sugar of the former. For example we could replace

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

The Meaning of Regular Expressions

So far we have only considered informally what the *meaning* of a regular expression is. This is not good enough for specifications of what algorithms are supposed to do or which problems they are supposed to solve.

To define the meaning of a regular expression we will associate with every regular expression a language, or set of strings. This language contains all the strings the regular expression is supposed to match. To understand what is going on here it is crucial that you have read the handout about basic mathematical notations.

The ***meaning of a regular expression*** can be defined by a recursive function called L (for language), which is defined as follows

[4] More about Scala is in the handout about *A Crash-Course on Scala*.

$$
\begin{array}{rcll}
L(\mathbf{0}) & \stackrel{\text{def}}{=} & \{\} & \\
L(\mathbf{1}) & \stackrel{\text{def}}{=} & \{[]\} & \\
L(c) & \stackrel{\text{def}}{=} & \{"c"\} & \text{or equivalently } \stackrel{\text{def}}{=} \{[c]\} \\
L(r_1 + r_2) & \stackrel{\text{def}}{=} & L(r_1) \cup L(r_2) & \\
L(r_1 \cdot r_2) & \stackrel{\text{def}}{=} & L(r_1) \mathbin{@} L(r_2) & \\
L(r^*) & \stackrel{\text{def}}{=} & (L(r))\star &
\end{array}
$$

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

$$L(h \cdot e \cdot l \cdot l \cdot o) = \{"hello"\}$$

This is expected because this regular expression is only supposed to match the "*hello*"-string. Similarly if we have the choice-regular-expression $a + b$, its meaning is

$$L(a + b) = \{"a", "b"\}$$

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 they have the same meaning. For example you can do the following calculation which shows they have the same meaning:

$$
\begin{array}{rcl}
L((r_1 + r_2) + r_3) & = & L(r_1 + r_2) \cup L(r_3) \\
& = & L(r_1) \cup L(r_2) \cup L(r_3) \\
& = & L(r_1) \cup L(r_2 + r_3) \\
& = & L(r_1 + (r_2 + r_3))
\end{array}
$$

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 if and only if $s \in L(r)$. 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.

There is one more feature of regular expressions that is worth mentioning. Given some strings, there are in general many different regular expressions that can recognise these strings. This is obvious with the regular expression $a + b$ which can match the strings a and b. But also the regular expression $b + a$ would match the same strings. However, sometimes it is not so obvious whether two regular expressions match the same strings: for example do r^* and $\mathbf{1} + r \cdot r^*$ match the same strings? What about $\mathbf{0}^*$ and $\mathbf{1}^*$? This suggests the following relation between ***equivalent regular expressions***:

$$r_1 \equiv r_2 \stackrel{\text{def}}{=} L(r_1) = L(r_2)$$

That means two regular expressions are said to be equivalent if they match the same set of strings. Therefore we do not really distinguish between the different regular expression $(r_1 + r_2) + r_3$ and $r_1 + (r_2 + r_3)$, because they are equivalent. I leave you to the question whether

$$\mathbf{0}^* \equiv \mathbf{1}^*$$

holds or not? Such equivalences will be important for our matching algorithm, because we can use them to simplify regular expressions, which will mean we can speed up the calculations.

My Fascination for Regular Expressions

Up until a few years ago I was not really interested in regular expressions. They have been studied for the last 60 years (by smarter people than me)—surely nothing new can be found out about them. I even have the vague recollection that I did not quite understand them during my study. If I remember correctly,[5] I got utterly confused about $\mathbf{1}$ (which my lecturer wrote as ϵ) and the empty string.[6] Since my study, I have used regular expressions for implementing lexers and parsers as I have always been interested in all kinds of programming languages and compilers, which invariably need regular expression in some form or shape.

To understand my fascination *nowadays* with regular expressions, you need to know that my main scientific interest for the last 14 years has been with theorem provers. I am a core developer of one of them.[7] Theorem provers are systems in which you can formally reason about mathematical concepts, but also about programs. In this way they can help with writing bug-free code. Theorem provers have proved already their value in a number of systems (even in terms of hard cash), but they are still clunky and difficult to use by average programmers.

Anyway, in about 2011 I came across the notion of ***derivatives of regular expressions***. This notion allows one to do almost all calculations in regular language theory on the level of regular expressions, not needing any automata. This is important because automata are graphs and it is rather difficult to reason about graphs in theorem provers. In contrast, to reason about regular expressions is easy-peasy in theorem provers. Is this important? I think yes, because according to Kuklewicz nearly all POSIX-based regular expression matchers are buggy.[8] With my PhD student Fahad Ausaf I am currently working on proving the correctness for one such matcher that was proposed by Sulzmann and Lu in 2014.[9] This would be an attractive results since we will be able to prove that the algorithm is really correct, but also that the machine code(!) that

[5] That was really a long time ago.
[6] Obviously the lecturer must have been bad.
[7] http://isabelle.in.tum.de
[8] http://www.haskell.org/haskellwiki/Regex_Posix
[9] http://goo.gl/bz0eHp

implements this code efficiently is correct. Writing programs in this way does not leave any room for potential errors or bugs. How nice!

What also helped with my fascination with regular expressions is that we could indeed find out new things about them that have surprised some experts in the field of regular expressions. Together with two colleagues from China, I was able to prove the Myhill-Nerode theorem by only using regular expressions and the notion of derivatives. Earlier versions of this theorem used always automata in the proof. Using this theorem we can show that regular languages are closed under complementation, something which Gasarch in his blog[10] assumed can only be shown via automata. Even sombody who has written a 700+-page book[11] on regular exprssions did not know better. Well, we showed it can also be done with regular expressions only.[12] What a feeling if you are an outsider to the subject!

To conclude: Despite my early ignorance about regular expressions, I find them now very interesting. They have a beautiful mathematical theory behind them, which can be sometimes quite deep and contain hidden snares. They have practical importance (remember the shocking runtime of the regular expression matchers in Python and Ruby in some instances). People who are not very familiar with the mathematical background of regular expressions get them consistently wrong. The hope is that we can do better in the future—for example by proving that the algorithms actually satisfy their specification and that the corresponding implementations do not contain any bugs. We are close, but not yet quite there.

Notwithstanding my fascination, I am also happy to admit that regular expressions have their shortcomings. There are some well-known "theoretical" shortcomings, for example recognising strings of the form $a^n b^n$. I am not so bothered by them. What I am bothered about is when regular expressions are in the way of practical programming. For example, it turns out that the regular expression for email addresses shown in (1) is hopelessly inadequate for recognising all of them (despite being touted as something every computer scientist should know about). The W3 Consortium (which standardises the Web) proposes to use the following, already more complicated regular expressions for email addresses:

```
[a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+(?:\.[a-zA-Z0-9-]+)*
```

But they admit that by using this regular expression they wilfully violate the RFC 5322 standard which specifies the syntax of email addresses. With their proposed regular expression they are too strict in some cases and too lax in others. Not a good situation to be in. A regular expression that is claimed to be closer to the standard is shown in Figure 4. Whether this claim is true or not, I would not know—the only thing I can say to this regular expression is it is a monstrosity. However, this might actually be an argument against the RFC standard, rather than against regular expressions. A similar argument is made

[10] http://goo.gl/2R11Fw
[11] http://goo.gl/fD0eHx
[12] http://www.inf.kcl.ac.uk/staff/urbanc/Publications/rexp.pdf

in

https://elliot.land/validating-an-email-address

which explains some of the crazier parts of email addresses. Still it is good to know that some tasks in text processing just cannot be achieved by using regular expressions.

```
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.

```
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 def crawl(url: String, n: Int) : Unit = {
24   if (n == 0) ()
25   else if (my_urls.findFirstIn(url) == None) {
26     println(s"Visiting: $n $url")
27     get_page(url); ()
28   }
29   else {
30     println(s"Visiting: $n $url")
31     for (u <- get_all_URLs(get_page(url))) crawl(u, n - 1)
32   }
33 }
34
35 // starting URL for the crawler
36 val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc"""
37
38
39 // can now deal with depth 3 and beyond
40 crawl(startURL, 2)
```

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 24–28 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 email_pattern = """([a-z0-9_\.-]+)@([\da-z\.-]+)\.([a-z\.]{2,6})""".r
16
17 def unquote(s: String) = s.drop(1).dropRight(1)
18
19 def get_all_URLs(page: String) : Set[String] =
20   http_pattern.findAllIn(page).map(unquote).toSet
21
22 def print_str(s: String) =
23   if (s == "") () else println(s)
24
25 def crawl(url: String, n: Int) : Unit = {
26   if (n == 0) ()
27   else {
28     println(s"Visiting: $n $url")
29     val page = get_page(url)
30     print_str(email_pattern.findAllIn(page).mkString("\n"))
31     for (u <- get_all_URLs(page).par) crawl(u, n - 1)
32   }
33 }
34
35 // staring URL for the crawler
36 val startURL = """http://www.inf.kcl.ac.uk/staff/urbanc"""
37
38 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 15. The main change is in Line 30 where all email addresses that can be found in a page are printed.

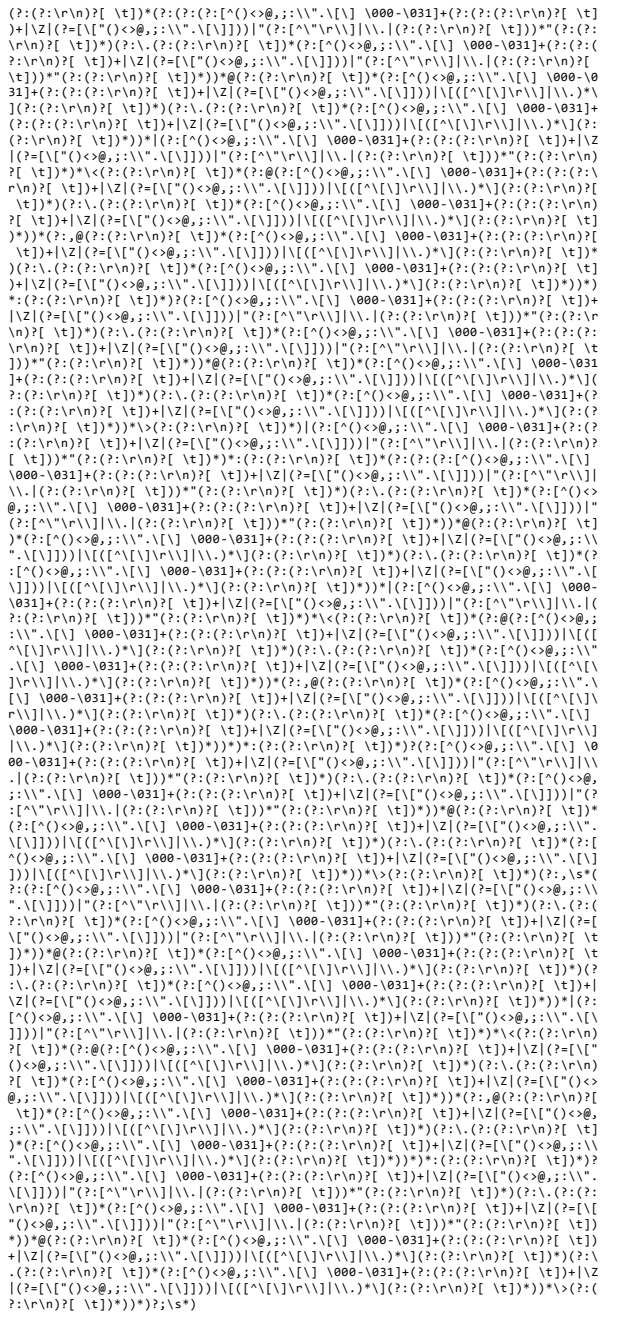


Figure 4: Nothing that can be said about this regular expression…