Informatics 2A: Tutorial Sheet 2 - SOLUTIONS

Mary Cryan

1. (a) Let $\underline{D}$ (for decimal digits) be the language:

$$0+1+2+3+4+5+6+7+8+9.$$

The required regular expression is then:

$$\underline{D}^*(\underline{D}\ .+.\ \underline{D})\underline{D}^*(\epsilon+((E+e)(\epsilon+\text{`}+\text{'}+\text{`}-\text{'})\underline{D}\ \underline{D}^*))$$

(many variations are possible). Note that in the above, the . is representing the point that gets written between the whole-part of the number and the trailing digits - it's not an operator, it is part of the resultant expression.

(b) For egrep, we have to take the alternative notation into consideration. For example '.' indicates "any single character" in an egrep pattern, so to describe the decimal point we need to write `\.` instead. Also, to take the union of two patterns, we use `|` not +. And finally, for unions of characters we can use square brackets (eg `[eE]` indicates either of `e` or `E`). The egrep command provided by John Longley (last year's lecturer) is `egrep` *pattern* `foo.java`, where *pattern* is:

```
"[0-9]*(([0-9]\.)|(\.[0-9]))[0-9]*([Ee][+-]?[0-9]+)?"
```

Or alternatively the following pattern (my own solution) will work

```
"([0-9]+\.|[0-9]*\.[0-9]+)(([Ee][-+]?[0-9]+)?)"
```

It will take quite a bit of hacking to get this right, and it is helpful to be logged-in to DICE (or any Linux environment) so you can test against the tutorial sheet .tex file (the strings that should be returned have commented-lines saying that, and it's reassuring to see those as you tweak things).

In many versions of the machine syntax, `[0-9]` can be abbreviated e.g. to `[\d]` or `:digits:`, however `\d` also matches a collection of decimal digits in other languages.

(c) An example can be seen with the strings `7.` and `7.e5`, These are both accepted (or should be), but `7.e` should not. The maximum number of consecutive non-accepting states between accepting states is two, as seen with the string `7.e-5`. (This has implications for longest-match lexing: we cannot simply stop as soon as we pass from an accepting state to a non-accepting one, as we might enter an accepting state again later.)

2. (a) Let $\underline{S}$ be the language of string characters, i.e. all input characters except `"`(double quote) and \\(backslash).
Then the required regular expression is:

$$\texttt{"}\,(\underline{S}+\backslash\,(\,\texttt{"}+\backslash+\texttt{b}+\texttt{t}+\texttt{n}+\texttt{f}+\texttt{r}))^*\,\texttt{"}$$

(b) Translating this into machine syntax with suitable escapes for `"` and \\, John Longley's solution is:

```
egrep "\"([^\"\\]|([\\][\"\\btnfr]))*\"" foo.java
```

Note that there are two levels of escape sequences involved in the pattern above where we want to replace the special `\` options after excluding free choice of the `\` and `"` characters. It's actually quite difficult to get the details of exclusions and re-introductions exactly right (having to use `\\` to just replace a single `\` inside a string. In my own workings (before checking John's solution) I had trouble combining the `\` and `"` inside the square brackets (especially with extra things) and for me it's clearer (and more reassuring) to rewrite John's solution as

```
egrep "\"([^\"\\]|([\\]([\"]|[\\]|[btnfr])))*\"" foo.java
```

I like this because it separates the `\\` and `\"` away from the individual characters `b`, `t` etc (recombining with the `|` which does union.

(c) A possible `egrep` command to achieve this is:

```
egrep "^[^\"]*\"[^\"]*(\"[^\"]*\"[^\"]*)*$" foo.java
```

Note that `^` and `$` match the start and end of line respectively. This use of `^` is different to when it's used inside a square bracket with a set of characters (there it indicates the complement of that set of characters).

This is a tricky pattern to decipher (or to come up with) but it actually relates to the example r.ex. from Lecture 7 where we were checking different ways of specifying binary strings with an even number of 0s. But in this case it's odd not even, and it's `"` not 0, that we are trying to specify.

3. In each of (a)-(d) below, let L denote the language in question.

(a) Not regular.
Given $k \geq 0$, consider $x = \epsilon$, $y = a^k$, $z = ba^k$.
We have $xyz \in L$ and the number of states of y is greater than k, so we are required to test it against the pumping lemma parameters.
Given *any* splitting of y as uvw where $v \neq \epsilon$ (this is not our choice, the adversary can split y however it wants), we take $i = 0$ (this is our choice, we can consider any i we want).
Then $uv^iw = uw = a^l$ for some $l < k$, so $xuv^iwz = a^lba^k \notin L$.

(b) Not regular.
Given $k \geq 0$, let $x = a^kb$, $y = a^k$, $z = b$, so that $xyz \in L$, and since y is longer than k, we must test it against the pumping lemma.
Given any splitting of y as uvw with $v \neq \epsilon$, we may take $i = 2$ (just for a change).
Then $xuv^iwz = a^kba^lb$ for some $l > k$, whence $xuv^iwz \notin L$.

(c) Regular. The trick is to note that the strings of L are exactly those that switch between a's and b's an even number of times, i.e. those that start and end with the same letter. This is because we get an ab

every time we switch from a's to b's, and a ba every time we switch from b's to a's.
So L corresponds to the following regular expression:

$$\epsilon \,+\, a \,+\, b \,+\, a(a{+}b)^*a \,+\, b(a{+}b)^*b \ .$$

(d) Not regular.
Given $k \geq 0$, let $p \geq k$ be prime (there must be such a p, using Euclid's theorem that there are infinitely many prime numbers).
Consider $x = \epsilon$, $y = a^p$, $z = \epsilon$. Then $xyz = a^p \in L$.
Given any splitting of y as uvw where $v \neq \epsilon$, take $i = p + 1$.
Note that $|xuwz| = p - |v|$ and $|v^i| = (p + 1)|v|$, so $|xuv^iwz| = p - |v| + (p+1)|v| = p(|v| + 1)$, which is not prime since both factors (p and $|v| + 1$) are ≥ 2. Thus $xuv^iwz \notin L$.

Note that as i varies, the possible lengths of the string xuv^iwz form an *arithmetic progression*. In the above, we are essentially exploiting the idea that there are no infinite arithmetic progressions of prime numbers.

As a side remark: the Green-Tao Theorem (2004), a celebrated result in number theory, states that, even though the prime numbers do not contain any infinite arithmetic progressions, they do contain arbitrarily long ones. So one can't place any fixed upper bound on the i needed in the above proof in advance.