Imperial College London

Language Processors

Lab 2 – Regex

Max Cattafi (m.cattafi@imperial.ac.uk)

Regex

WHENEVER I LEARN A NEW SKILL I CONCOCT ELABORATE FANTASY SCENARIOS WHERE IT LETS ME SAVE THE DAY.
OH NO! THE KILLER MUST HAVE FOLLOWED HER ON VACATION!
BUT TO FIND THEM WE'D HAVE TO SEARCH THROUGH 200 MB OF EMAILS LOOKING FOR SOMETHING FORMATTED LIKE AN ADDRESS!
IT'S HOPELESS!
EVERYBODY STAND BACK.
I KNOW REGULAR EXPRESSIONS.
PERL!
TAP TAP

Besides being associated to type-3 (regular) grammars, regular expressions are very used as practical pattern matching tools, for instance for log analysis, usually in implementations which pragmatically extend the theoretical expressiveness.

Scripting languages as Perl, and Linux command line tools as `grep` have popularized this use. The C++11 standard has introduced regex also in C++. However for the time being not all compiler implementations include this feature (for instance the version of g++ currently installed in the labs doesn't include a full implementation). When this is the case one needs to use external libraries such as Boost.Regex (see also below).

Useful online interactive tools for learning regex: Rubular (also contains a very handy quick reference table at the bottom of the page) and Regexr.

Consider file `prog1.c` from the previous lab. Let's try

```
grep // prog1.c
```

The output is:

```
// we just declare a few variables...
int num1=0, /* int num1 is initialized */ num2; // int num2 is not
```

Usually `grep` works on lines, in this case the output includes all the lines containing a double slash (that is, those including a double slash comment).

We can use option `-o` to only output the matched portion:

```
grep -o // prog1.c
```

```
//
//
```

If we want to include the comment text we have to specify a regex pattern (double quotes are necessary to enclose the regex in this case because of the presence of the `*` which could be read as a command line wildcard rather than a regex operator):

```
grep -o "//.*" prog1.c
```

```
// we just declare a few variables...
// int num2 is not
```

Now our match includes anything (the dot symbol `.` matches any character, although usually the newline is excluded, and adding the `*` symbol we specify that there can be from 0 up to infinite of these characters) between the `//` and the end of the line.

Experiment with regex on source files trying to match lines with various criteria. For instance: lines containing only double slash comments (that is, beginning with a double slash: use the `^` symbol for the start of line and remember to take the presence of spaces, becase of indentation, into account), variable declarations, function declarations, instructions etc.

For multi-line patterns try using Rubular (and the option `m`). When Rubular requires you to 'escape' something, prefix it with a backslash. For instance instead of the `//` pattern seen above you would have to use \/\/.

C++ regex

As mentioned above Boost.Regex provides a possible implementation of regex for C++.

In `credit_card_example.cpp` you can find an example of how Boost.Regex works. Read the source code and the added comments, then compile and run the example to see it in action.

In order to compile source which uses Boost.Regex (in our lab it is installed only on Linux) you need to link the library, for instance from the command line:

```
 g++ credit_card_example.cpp -o creditex -L/usr/lib/x86_64-linux-gnu/ -lboost_re
```

Experiment with various regular expressions (e.g. to match or remove comments from C source files) as in the previous lab.

Notice that regex variables (see explanation in `credit_card_example.cpp`) can be used also in matching regex. Imagine for instance we encode numbers in strings, so that each number is represented by a string of `n` whose length is equal to the number (e.g. 5 is `nnnnn`) and consider the following regex:

```
boost::regex e("^n?$|^(nn+?)\\1+$");
```

The question mark in `(nn+?)` is not the usual question mark meaning "0 or 1 occurrences of this": after a + (Kleene cross) or a * (Kleene star) the question mark means that the match is 'lazy'. For instance if the string is `aaaaaa` and the regex is `^a+`, the match is `aaaaaa`, while with `^a+?` the match is `a`.

Can you see what the regex does?

Answer:

It matches only with strings (of n) whose length is not a prime number.