1. General Vocabulary for Lexical Analysis

- **Alphabet** a finite set of symbols (characters)
- **String** a finite sequence of symbols from some alphabet
- **Language** a set of strings
- **Token** a pair consisting of a token name (syntactic category) and an optional attribute (semantic information)
- **Pattern** a description of the form that lexemes of a token might take
- **Lexeme** a string that matches the pattern of a token

2. What language is denoted by each of the following regular expressions? Try writing down a few simple strings and give a concise description of the language.

   (a) \[\_a-zA-Z\] \[\_a-zA-Z0-9\]*
   **Answer:** Python (in fact many other languages) identifiers

   (b) (\(\varepsilon\)|a)b*)*
   **Answer:** All possible sequences with characters a and b

   (c) (0*1+)+|(1*0+)+
   **Answer:** Binary strings with different first and last bits.

3. Write a regular expression for the following languages:

   (a) Even binary numbers (read from left to right – i.e., “1101” is 13).
   **Answer:** “01*0” or “101*010”, depending on whether we allow leading zeros.

   (b) All strings of lowercase letters in which the letters are in ascending lexicographic order.
   **Answer:** a*b*c*...y*z*
(c) Java-like comments, consisting of a string surrounded by /* and */ without an intervening */ unless it is inside double quotes ("").

**Answer:** A simple example, only allowing letters in comments, would be:

```
\/(\*[\[\s]?("(\*[\s]?([^*/])")?)*\*\n```

4. Regular expression with backreference:

Now you can reference a matched group within your expression. For example, `(\w)\1` could now match any pair of repeated letters: “aa”, “bb”, etc. As you might guess, \2 refers to the second matched group and so forth.

(a) Suppose now you have a large CSV (Comma Separated Values) file, each row of which has the form `firstname,lastname,email`. You want to find out all uninteresting people who use boring email address of the form `firstname+lastname@xxx.com` and avoid making friends with them. Try to write a normal (without backreference) regular expression that describes this pattern, or state it’s impossible.

**Answer:** Not possible

(b) Now try to write expression for the mentioned pattern using backreference.

**Answer:** `(\w+), (\w+), \1@` 2

(c) (DEMO) If we try to match a string of letter “a”s, e.g. “aaaaaaa”, with this regex `/a+a+a+a+a+a+a+a+a+c/`, which regex engine do you think would run faster? Or do you think they run in comparable speed?

i. The young and rising language *Ruby*

ii. The 30-year-old text processing tool *AWK*