lab session 1: compiler construction

November 21, 2017

The first lab session of the course *Compiler Construction* consists of 6 exercises. Each of these exercise must be solved using Flex (scanner generator). For each exercise, you need to hand in two files to the automated judging system `Themis`: a flex file (i.e. `filename.fl`) and a `Makefile`.

Each solved exercise (i.e. accepted by Themis) is worth 10 points, and you get 10 points for free. The remaining 30 points are awarded by manual inspection by the teaching assistants. Hence, if you solved all problems, your grade is not automatically a 10. For some of the exercises, solutions can easily be found on the internet. Be warned, that copying those is considered plagiarism!

Exercise 1: Caesar cipher

A Caesar cipher is one of the simplest known encryption techniques. The method is named after Julius Caesar, who used it in his private correspondence. It is a substitution cipher in which each letter in a text is replaced by the next letter in the alphabet. For example, with a right rotation of only 1 position, A would be replaced by B, B by C, C by D, etc. The letter Z becomes an A. The same technique is applied to lowercase letters (a becomes b, etc) and digits, (0 becomes 1, 1 becomes 2, ..., and 9 becomes 0). Other characters (like commas, spaces, etc.) are left unchanged. Make a flex file that reads text from `stdin` and outputs the corresponding encrypted text.

Example 1:

input:

```
THE QUICK BROWN FOX JUMPS OVER THE LAZY DOG.
```

output:

```
UIF RVJDL CSPXO GPY KVNQT PWFS UIF MBAZ EPH.
```

Example 2:

input:

```
AMAZINGLY FEW DISCOTHEQUES PROVIDE JUKEBOXES.
```

output:

```
BNBAJOHMZ GFX EJTDPUIFRVFT QSPWJEF KVLFCPYFT.
```

Example 3:

input:

```
The above 2 sentences are examples of pangrams.
These are sentences in which each letter of A..Z is used at least 1 times.
```

output:

```
Uif bcpwf 3 tfoufodft bsf fybnqmft pg qbohsbnt.
Uiftf bsf tfoufodft jo xijdi fbdi mfuufs pg B..A jt vtfe bu mfbtu 2 ujnft.
```

Exercise 2: Morse code

On Nestor, you can find a file with the Morse code alphabet. Make a flex file that reads text from `stdin` and outputs the corresponding text in Morse code. Note that characters which are not in the alphabet should be copied verbatim to the output.

Example 1:

input:

```
THE QUICK BROWN FOX JUMPS.
```

output:

```
-..... --.-..-..-.-.-.- -....-.---.---. ..-.----..- .---..---.--.....
```

Example 2:

input:

```
SOS = Save Our Souls
```

output:

```
...---... -...- ....-...-. ---..-.-. ...---..-.-.....
```

Example 3:

input:

```
Or, SOS = Save Our Ship?
Actually, both are not TRUE!
```

output:

```
---.-.--..-- ...---... -...- ....-...-. ---..-.-. ..........--...--..
.--.-.-..-.-.-...-..-.----..-- -...----.... .-.-.. -.---- -.-...-.!
```

Exercise 3: Roman numerals

Make a flex file that accepts on its input *Roman numerals*. For each Roman numeral, the output should be the numeral, followed by an equals sign, follwed by the decimal representation of the number. You may assume that the numerals on the input are correct (i.e. XIV, and not XIIII).

Example 1:

input:

```
XLII MMXVI
```

output:

```
XLII=42
MMXVI=2016
```

Example 2:

input:

```
MCCXXXIV
```

output:

```
MCCXXXIV=1234
```

Example 3:

input:

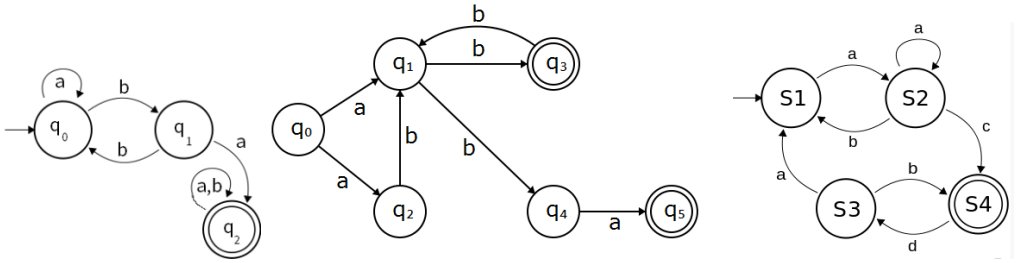
```
MMMMCMXCIX
```

output:

```
MMMMCMXCIX=4999
```

Exercise 4: DFA

In the following figure you see from left to right three DFAs: `DFA1`, `DFA2`, and `DFA3`.



Make a flex file that reads text from `stdin` and outputs longest matches that are accepted by the automata. All non matching characters should be ignored (i.e. skipped).

Example 1:

input:
```
Habbbbaba!
```
output:
```
DFA1 accepted: abbbbaba
```

Example 2:

input:
```
abbbbbbbbbbbaaaaa
```
output:
```
DFA1 accepted: abbbbbbbbbbaaaaa
```

Example 3:

input:
```
abaabaacbaaaac
```
output:
```
DFA3 accepted: abaabaac
DFA1 accepted: baaaa
```

Exercise 5: Accepting a's and b's

Make a flex file that accepts lines of input, for which each line must contain exactly two `a`'s and more than two `b`'s.

Example 1:

input:
```
aabbbb
```
output:
```
accepted: aabbbb
```

Example 2:

input:
```
aaabb
```
output:
```
rejected: aaabb
```

Example 3:

input:
```
babbbab
aba
```
output:
```
accepted: babbbab
rejected: aba
```

Exercise 6: RPN evaluator

In *reverse Polish notation (RPN)* arithmetic expressions are written using post-fix notation, i.e. the operators follow their operands; for instance, to add 3 and 4, one would write `3 4 +` rather than `3 + 4`. Another example, using multiple operations is `3 4 - 5 +`, which in conventional notation is `3 - 4 + 5`. A nice property of RPN is that there is no need for parentheses, e.g. in conventional notation we need parentheses in `(7 - 2)*5`, but in RPN this is simply written as `7 2 - 5 *`.

Write a Flex file that accepts a string in RPN from `stdin` and outputs its evaluation. You may assume that the input string consists of numbers, and the operators `+` (addition), `-` (subtraction), `*` (multiplication), and `/` (division). Moreover, tokens are separated by whitespace.

Example 1:

input:

```
17 2 - 2 * 5 /
```

output:

```
6.000000
```

Example 2:

input:

```
2 1 12 3 / - +
```

output:

```
-1.000000
```

Example 3:

input:

```
-2.0 3.1415925E+1 *
```

output:

```
-62.831850
```