McMaster University Comp Sci 4TB3/6TB3, Winter Term 2017/18 — Lab 3
For the Labs on January 23 - 26,
Due Monday, January 29, 11 pm

Eden Burton, Jenny Wang, Spencer Park

- *Submission is to be done exclusively through Avenue. Submissions via e-mail will **not** be accepted. A **10% penalty** will be accessed for each day the lab is submitted after the due date.*
- This assignment requires access to a Linux, MacOS X, or some other Unix computer. You can log in remotely to either `moore.mcmaster.ca` or to `mills.mcmaster.ca` with ssh. Submissions are tested on `moore.mcmaster.ca`, please check if your submission works there.
- In this lab, you are allowed to work in pairs, provided that you split the work equally and arrive at a common understanding of the solution. However, in that case you must state in your submission the person you worked with, such that similarities in the solution will not be construed as Academic Dishonesty. Working in groups of three or larger is not allowed and will be considered Academic Dishonesty. If you look for someone to work with, we will try to find a match, please contact the TAs.
- You are allowed and encouraged to talk to everyone in the course to get a common understanding of the problem, but you can share a solution only with your collaborator, if you work in a pair.
- Some questions require to read from *standard input* and out write to *standard output*. If you are not familiar with these terms, please familiarize yourself and make sure that you understand command line redirection ($<$, $>$) and pipes ($|$) in the Unix shell, as this is used to test the submissions.
- The Tutorial Exercises will be presented in the tutorials; you need to submit only answers to the Lab Questions. In the labs, the solution to last week's lab questions are discussed and you can get help with this week's lab questions. Attendance at the labs is not checked.

Tutorial Exercise 1 (Extracting Columns)**.** Consider a file with columns separated by tabs. Write `grep` commands to extract the first, second, third column. Use that in a program editor (like TextWrangler on OS X) to replace each line with its first, second, or third column.

Answer. For extracting the first, second, or third column,

replace `(^[^\t]*).*$` with `\1`
replace `^[^\t]*\t([^\t]*).*$` with `\1`
replace `^[^\t]*\t[^\t]*\t([^\t]*).*$` with `\1`

The `sed` scripts:

```
sed -r 's/(^[^\t]*).*$/\1/g'
sed -r 's/^[^\t]*\t([^\t]*).*$/\1/g'
sed -r 's/^[^\t]*\t[^\t]*\t([^\t]*).*$/\1/g'
```

Note that `\t` is a perl extension, which is supported by `sed` under Linux, but not OS X. IntelliJ needs `*1` instead of `\1`. The `vi` (or `vim`) editor requires that parenthesis used for matching are escaped, as in `\(^[^\t]*\).*$`.

Now consider a CSV file with columns separated by commas instead of tabs. However, commas that are enclosed in `"` are not considered separators.

Tutorial Exercise 2 (Lowercase Tags)**.** Consider IMG tags in html files. Write a `grep` command to lowercase SRC values.

Answer.
replace `(<img [^>]*src\s*=\s*")([^"]+)("[^>]*>)` with `\1\L\2\E\3`

Lab Question 1 (Checking Credit Cards, 8 points)**.** Visa card numbers start with a 4. New Visa cards have 16 digits, old cards have 13 digits. MasterCard numbers start with the numbers 51 through 55. All have 16 digits. American Express card numbers start with 34 or 37 and have 15 digits. The last digit is a checksum calculated using Luhn's algorithm, see `http://en.wikipedia.org/wiki/Credit_card_number`

Design a web page that allows a credit card number to be entered and displays if the card is valid or invalid. Use JavaScript regular expressions to check if the credit card is well-formed and of proper length. Implement Luhn's algorithm in JavaScript. Submit a single html file named `Q1.html` that contains the JavaScript function `isValidCreditCard(sText)`. This function shall return true if the card number is valid and false otherwise.

Lab Question 2 (Sanitizing Pathnames, 8 points)**.** In Posix pathnames, components are separated by `/`. Consecutive multiple `/` have the same meaning as a single `/`. A final `/` has no meaning but an initial `/` is significant. Leading and trailing spaces are allowed but have no significance. A component consists of `a-z`, `A-Z`, `0-9`, and `.` (dot), with two special cases: a component with a single `.` component refers to the current directory and a `..` component refers to the parent directory. Note that `.` can also be part of a component. Portable pathnames have the restriction that each component can have at most 14 characters and the whole pathname can have at most 255 characters.

Implement a sanitizer for pathnames using lex (or flex) and C, see

Man page e.g. at `http://dinosaur.compilertools.net/flex/manpage.html`

Manual e.g. at `http://poincare.matf.bg.ac.rs/~aspasic/ppj/literatura/flex.pdf`

Your implementation should read from standard input and produce a sanitized portable pathname on standard output or an error message on standard error. The implementation has to use the regular expression facilities of lex (or flex) to check for the well-formedness of the input. Valid input and corresponding output are:

```
/aaa//bb/c/                          /aaa/bb/c
  aaa/b.b/../cc/./dd                 aaa/cc/dd
```

The sanitizer should read the input line by line from standard input until the end of the file. For each line, the sanitizer should either produce one line with the sanitized pathname on standard output or an error message on standard error and terminate immediately. The error messages to be produced and sample invalid input are:

```
/a//b/#/c                            invalid character
/012345678901234/bb                  component too long
aa/../..                             malformed pathname
/1/2/3/4/5/6/7/...256 characters long...   pathname too long
```

Submit a file `Q2.l` that can be run on `mills.mcmaster.ca` by

```
flex Q2.l; cc -std=c99 lex.yy.c -lfl; ./a.out
```

Hint: use the regular expression features of lex to check for invalid characters, too long components, and to "swallow" leading and trailing spaces, multiple consecutive `/`, and `.` components.

Lab Question 3 (Recursive Decent Parsing, 8 Points)**.** State whether a recursive decent parser can be built for the grammar specified below or not. Justify your response.

$$
\begin{array}{rcl}
Q & \rightarrow & \text{“}w\text{“ “(“ } E \text{ “)“ “\{“ } \{S\} \text{ “\}“} \\
S & \rightarrow & V \text{ “} = \text{“ } E \text{ “;“} \\
V & \rightarrow & I \mid I \text{ “[“ } E \text{ “]“} \\
E & \rightarrow & \text{“(“ } E \text{ “)“} \\
E & \rightarrow & N \text{ (“} + \text{“ } \mid \text{ “} - \text{“) } N \\
E & \rightarrow & B \text{ (“ } \mid \text{ “ } \mid \text{ “\&“) } B \\
N & \rightarrow & D\{D\} \\
I & \rightarrow & L\{L\} \\
D & \rightarrow & \text{“0“} \mid \text{“1“} \mid \text{“2“} \mid .. \mid \text{“9“} \\
L & \rightarrow & \text{“}a\text{“} \mid \text{“}b\text{“} \mid \text{“}c\text{“} \mid \text{“}d\text{“} \mid \text{“}e\text{“} \\
B & \rightarrow & \text{“}t\text{“} \mid \text{“}f\text{“}
\end{array}
$$

Build a recursive decent parser for the grammar (or a modified one that accepts the same language).

You can implement the parser in the language of your choice but you must provide a script called "script.“ which allows your program to run on a Linux server such as `mills.mcmaster.ca` hosted at the university. The program shall read from *standard input* until a newline character and write `‘‘accepted/n’’` or `‘‘rejected/n’’` to *standard output*. Note that the grammar does not include terminals such as spaces or the newline character.

w(1+2){aa=t&f;bb=1+1;}

is an example of a sentence that should be accepted by the parser.