PLC: Programming Assignment 1
Finite Automata, Regular Expressions, and Grammars [100 points]

Must be submitted via submission script (instruction below) by 7pm on Friday, Feb. 15th.

Instructions for submitting your solution

We are relying on automatic processing of homeworks to make grading more manageable for this class. Please follow these directions carefully to ensure that our scripts are able to process your submission.

- Put all the files you are submitting in a folder called `prog1` (please call it exactly this, and do not capitalize any letters or include spaces).
- To submit, you must use the `submit` script, provided by the Computer Support Group here at U. Iowa.
  - When you are ready to submit, you need to log on to a CS linux machine (for example, `linux.cs.uiowa.edu`), and type the command `submit`. You should type `submit` in a directory which contains your `prog1` folder as a subdirectory.
  - The `submit` command will ask you to specify the name of the folder you want to submit (so say `prog1`).
  - Then hit return to tell `submit` you are done specifying files.
  - The `submit` program will ask you what class you wish to submit to. You should specify `c_111`.
  - The `submit` program will then give you a list of possibilities for which assignment you are turning in. You should choose `PROG1` and hit return.
  - The `submit` script will then copy over your solution for us to grade.
  - You can submit multiple times, and only the latest submission will be kept.
  - For more on `submit`, see

    `http://www.divms.uiowa.edu/help/msstart/submit.html`
- If you submit after 7pm on the day the assignment is due, we will count your submission as late, and you will be penalized 10%. Therefore, it would be smart to test out the submission process well before your assignment is due. Because we keep only the most recent submission,

you can just submit an empty `prog1` folder if you like, to make sure you are able to follow the above procedure for submitting.

- You cannot submit part of the assignment on time and part late. We will grade the most recently submitted version, and if that is a late submission, that is the version we will use in assigning your grade (with the 10% late penalty in that case).

Office hours and help

Check out the syllabus for updated office hours and locations for Prof. Stump, Amit Jain, and Frank Fu:

`https://svn.divms.uiowa.edu/repos/clc/class/2013/111-pub/syllabus.pdf`

You can also email us with questions or to request an appointment.

Email. We will make every effort to reply to your email questions within 24 hours. We will try to reply faster than that on the Thursday and Friday preceding the deadline for the assignment, but we will probably not be able to reply after usual working hours of 5pm.

Allowed resources. You may ask the TAs and the instructor any question you want, and any help or guidance from course staff may be freely used in your solutions. Any material contained in the svn repository may also be freely used in your solutions (including copying and pasting parts of grammars). You may not look at other people's solutions, or allow them to look at yours. You can discuss the homework in general terms, but should not reveal specific answers or talk through details of code or grammars. Code available online but not written for this class is a bit of a gray area: probably our assignments are idiosyncratic enough that you will not find an exact solution, but neither should you search online for solutions to problems. Reading through code available publicly online to learn more about OCaml or grammars is ok, but you are not allowed to copy it into your solution.

A note about platforms. The Windows lab machines in 301 MacLean Hall have all the software packages on them that you will need, except tools like `gratr` found in the svn repository. We will try to support you if you install software on your own computer, whatever platform you use.

Compiling gratr on Windows. To compile the `gratr` tool (discussed below) on Windows, open a `cmd` shell (by typing "`cmd`" into the "Search programs and files" text box in the Windows start menu). Type "cd x" to change to a directory named x, or "cd .." to go up a directory level. Navigate to the subdirectory of `111-pub` for `gratr`, and type `compile`. This will invoke a `compile.bat` file in the `gratr` subdirectory, to compile `gratr`. You must have OCaml version 3.11 or later installed for this to work (it is already installed on the Windows lab machines in 301 MacLean). To run `gratr` once it is compiled, copy `gratr.exe` to the `prog1` directory.

Compiling gratr on Linux and Mac. Using the terminal, navigate to the `gratr` subdirectory of `111-pub`, and type `make`. This will only work if you have OCaml installed on your machine. To run `gratr` once it is compiled, copy `gratr` to the `prog1` directory. You will probably need to run this as `./gratr`.

1 Understanding regular expressions [40 points]

This problem is not hard, but to be sure you get full credit you will want to run the `gratr` tool (see instructions above).

1. In the file `q1.gr` in this subdirectory, you will see a regular expression. Create a file called `q1.txt` containing any string that matches this regular expression. Do not include any spaces in the file. [10 points]

   You can test your file by asking `gratr` to process it with the automaton it generates from `q1.gr`. To do this, run the following command in a shell (either `cmd` on Windows or the terminal on Mac/Linux):

   ```
   gratr --run q1.txt --lang-only q1.gr
   ```

   This will produce a file called `run.txt`, where you can see if your string was accepted by `gratr`.

   Just for your information: when you run `gratr` as just described, it will also generate several GraphViz (`.gv` files). To run dot (a tool that is part of GraphViz) on `min.gv` to create a JPEG file `min.jpg` which you can then view with any image viewer, execute this command:

   ```
   dot -Tjpeg -o min.jpg min.gv
   ```

   Looking at `min.jpg` can help you understand what is happening with your `q1.txt`.

2. In the file `q2.gr`, you will see another regular expression. Create a file `q2.txt` containing a matching string. You can test your answer as above. [15 points]

3. In the file `q3.gr`, you will find a third regular expression. Note that this example uses notation `~[ '*' ]` to mean any character except `*`; more generally, you can put any list of character or subranges of characters (e.g., `['a'-'d']`, meaning all characters between `a` and `d` inclusive) in square brackets, and possibly complement the entirety with `~`. Create a file `q3.txt` containing a matching string, similarly to the previous two problems. [15 points]

2 Writing Regular Expressions [30 points]

1. Create a file called `q4.gr`, following the format you saw for example files `q1.gr` and others above, where you use regular expressions to describe the set of strings of `a`'s, `b`'s, and `c`'s where we can have any number of `a`'s or `b`'s, then any number of `b`'s or `c`'s, and then 1 or more `c`. You can use `q4.txt`, which I am providing, as a test case with `gratr`. You may need to add `'\n'?`, which optionally matches a newline (line break), at the end of your expression if your text editor inserts a newline at the end of your test files the way mine seems to do. [15 points]

2. For this problem, use the following lexical definition for `id`:

   ```
   id -> ['a'-'z']+ .
   ```

Create a file called `q5.gr`, similarly to the previous problem, where you use a regular expression to define sequences of at least one `id` (just cut and paste the definition above into your `q5.gr` file – the order of definitions does not matter), where `id`s are separated by either commas or semicolons (and where the whole input is optionally followed by `'\n'`, as in previous problems). You can use test it with `q5.txt` which I am providing. You will probably want to try your own tests, too, for corner cases like having just one `id`. [15 points]

3 Writing Grammars [30 points]

1. Write down a grammar in `q6.gr` for a language of simple function definitions, that look like this:

   ```
   f(x,y,z) = x+y+z ;;
   ```

   In more detail, your grammar should allow a sequence of one or more function definitions, where each function definition has a left-hand side with a function name and then a list of at least one argument, then an equals sign, then an addition expression or variable, and finally a double semi-colon. Variables and function names should be `id`s. To allow whitespace (spaces, tabs, and newlines), put

   ```
   Whitespace


   ws
   ```

   right before `Lexical` in your `.gr` file, and then add some rules defining `ws`. A test file is in `q6.txt`. Note that you do not have to have the grammar detect whether variables used on the right-hand side of the function definition are declared on the left-hand side (this is actually not possible). [20 points]

2. Write down a grammar in `q7.gr` for the set of strings using just the characters `a`, `b`, `0`, and `1`, where each `a` is followed somewhere in the string by exactly one matching `b`, and similarly each `0` is followed by exactly one matching `1`. A test file is given in `q7.txt`. Hint: this is similar to the balanced parentheses grammar (see lecture notes for Feb. 5th). You can test this by running `gratr` this way[1]:

   ```
   gratr --parse q7.txt q7.gr
   ```

   Look in the file `parsed.txt` for the result. [10 points]

[1] This is not implemented yet, but will be very soon.