NAME: _____

CS 330 Midterm

Spring 2016

Doodle here.

On the following pages, write things that are true and relevant.

1. *Matchmaker*

Consider this terse summary of regular expression parts and pieces:

match what?	how many times?	where?
`(pat)` - group and capture	`pat?` - 0 or 1, greedy	`^` - at beginning of line
`pat1|pat2` - `pat1` or `pat2`	`pat*` - 0 or more, greedy	`$` - at end of line
`[abc]` - `a`, `b`, or `c` (complement: `[^abc]`)	`pat*?` - 0 or more, lazy	`\b` - at word boundary
`\d` - any digit (complement: `\D`)	`pat+` - 1 or more, greedy	`(?<=pat)` - after `pat`
`\w` - any alphanumeric (complement: `\W`)	`pat+?` - 1 or more, lazy	`(?=pat)` - before `pat`
`\s` - any whitespace (complement: `\S`)	`pat{n}` - `n`	`(?!pat)` - not before `pat`
`.` - any non-newline character	`pat{n,}` - `n` or more, greedy	`(?<!pat)` - not after `pat`

Write regular expressions to match the following criteria and only the following criteria. Write regular expressions only, not Ruby code.

(a) Strings of binary numbers, e.g. 1001001.

(b) Numbers in the interval [10, 25]. Do not match sequences embedded in larger numbers, like 1025.

(c) Words containing a sequence of three vowels—because such words are beautiful and righteous.

(d) Real number literals, which may be positive or negative and which always have a decimal point with at least one number after it.

(e) The third field in a line of comma-separated values. Use capturing or lookaround assertions to home in on the third field. Assume commas do not appear in the fields themselves.

2. *Genotype*

   We refer to the base datatypes supported by an architecture as *primitives*. In addition to these primitives, many high-level languages allow for the construction of new, user-defined types. Identify three different languages that you are familiar with and briefly describe how each allows you to define new types. Provide an example of a new type in each language.

3. *Taking Names*

   You are creating an abstract syntax tree of a language where everything is an expression. You want to be able to query this tree for a set or list of all *unique* variable identifiers. Add a `collectIDs` method to this hierarchy to generate such a list. Assume that constructors and other necessary methods already exist. They are not relevant to this problem.

```
public abstract class Expression {









}

public class ExpressionIdentifier extends Expression {
  private String id;









}

public class ExpressionAdd extends Expression  {
  private Expression a;
  private Expression b;









}
```

```
public class ExpressionBlock extends Expression {
  private ArrayList<Expression> statements;









}

public class ExpressionRepeat extends Expression {
  private Expression count;
  private ExpressionBlock body;









}
```

4. *Place for Everything*

   What kind of data/information goes in each of the following areas of memory and the CPU?

   (a) Register `%esp`.

   (b) Register `%eax`.

   (c) Register `%ebp`.

   (d) Register `%eip`.

   (e) The stack. (Be thorough.)

   (f) The `.data` section.

   (g) The `.text` section.

   (h) Address 0.

5. *Class List*

   You have a directory containing a bunch of C++ source code, both header and implementation files. The source files were written by a variety of developers who refused to adopt a consistent naming convention. So, the files end in a variety of extensions: `hxx`, `hh`, `cpp`, `cxx`, `plusplusc`, and so on. You have neither the time nor the inclination to generate an exhaustive list of all the extensions used. You do know, however, that each implementation file represents a single class and is accompanied by a similarly named header file. Write a single line of shell code to generate the names of all the classes in this directory. Each name should appear only once. For example, suppose the directory contains:

```
Character.cxx
Character.h
NPC.plusplusc
NPC.hpp
Powerup.cxx
Powerup.h
PegasusBoots.cpp
PegasusBoots.hh
```

   Running your line of shell code produces this list:

```
Character
NPC
Powerup
PegasusBoots
```

   The following utilities may be helpful to you. Or they may not.

   - `grep PATTERN`: prints all lines matching a given pattern
   - `head -nNUMBER`: prints the first NUMBER lines of input
   - `ls`: list all the files in the current directory
   - `tail -nNUMBER`: prints the last NUMBER lines of input
   - `uniq`: filters out repeated lines of input; assumes input is sorted
   - `cut -fNUMBER -dSEPARATOR`: separates lines of text by `SEPARATOR` and yields the `NUMBER`th field of each input line; `NUMBER` is 1-based
   - `sort [-r]`: sorts the input [in reverse]