NAME: _____

CS 330 Midterm

Spring 2014

Doodle here.

1. *Patternity*

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 text. Write regex only, not Ruby code.

(a) All Java method names starting with `get`.

(b) All array indexing of elements 1, 2, or 3.

(c) The words `east` or `west`. Do not match text where these words appear inside another word, like in `beast` or `Qwest`.

(d) Each line beginning with any amount of whitespace, a number, and a period.

2. *Heapology*

Assume you manage the heap using the `malloc`/`free` implementation we wrote in lecture, whose salient points follow:

- You maintain a list of free blocks, with nodes stored in the heaps' free blocks. Each node stores at its beginning its capacity and the address of the next node.
- When a `malloc` request comes in, the free list is scanned for the first block that can satisfy the request. The size of the allocation is stored in the first `sizeof(int)` bytes and an address pointing after the size is returned to the caller.
- When an allocation is freed, a new node starting `sizeof(int)` bytes back from the passed address is prepended onto the free list.

Given this implementation, what terrible things might happen in the following situations?

(a) You `malloc` but you don't free.

(b) You write beyond the bounds of your allocation.

(c) You never `malloc`.

(d) You free a pointer twice.

(e) You free an address that wasn't previously `malloc`'d.

(f) You `malloc` many small requests and consume all available heap space. You then `free` all allocations. You then try to `malloc` a very large allocation.

3. *Record-breaking*

   In JavaScript, which doesn't have static typing, all objects are like `structs` whose fields can be added dynamically:

   ```
   var o = new Object();
   o.hitPoints = 100;
   ```

   Suppose you wanted all objects in Java to likewise have support for the *dynamic* addition of properties. What would you change about the language?

4. *Pound Include*

   C compilers rely on explicit function declarations (often in headers) to ensure that functions are called properly. How could we remove our dependence on these declarations? What would have to change to make them unnecessary?

5. *Javacide*

   Run `ps` at a Unix command prompt, and you see output with the following format:

```
  PID TTY           TIME CMD
  770 ttys000    0:00.58 -zsh
96945 ttys000    0:00.00 tmux
96950 ttys001    0:00.21 java
99642 ttys002    0:01.57 vim foo.c
```

   Write one or two reusable lines of shell script that kill any process mentioning "java". You might find the following useful:

   - `grep <PATTERN>` - echo to STDOUT all lines in STDIN that match pattern
   - `kill <PID>` - kill the process with the given ID
   - `cut -f# -d'<DELIMITER>'` - extract the #th delimited field from STDIN

6. *Pushing Up Daisies*

   Programs with an exorbitant amount of recursion often crash with a stack overflow exception. What specifically is filling up the stack?

7. *Befortran*

   You are a programmer in the 1950s who writes assembly code everyday. You see some ways to make writing code easier, safer, and faster for future generations. What are they?