

CS 370

Midterm Exam

March 9, 2007

The following exam is open book and open notes. You may feel free to use whatever additional reference material you wish, but **no electronic aids** are allowed. Please note the following instructions. There will be a ten point deduction for failure to comply with them:

- start each problem on a new sheet of paper
- write your Banner ID number, but not your name, on each sheet of paper you turn in

Also, please note the following:

- show your work whenever appropriate. There can be no partial credit unless you show how you derived your answers
- be succinct. You may lose points for facts that, while true, are not relevant to the question at hand

You have until 2:20 to finish the exam.

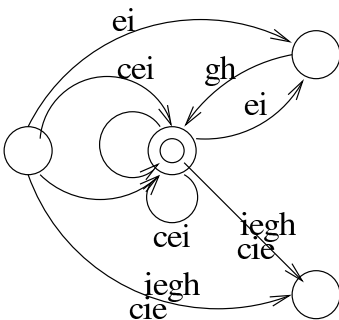
1. (20 points) Draw a picture of a FSM that implements “I before E except after C and when said as A as in ‘neighbor’ and ‘weigh’”.

More precisely, the FSM should accept a token made up solely of the letters a-z, such that an e can only be followed by an i when it’s immediately preceded by a c or followed by gh, and the substrings cie and iegh cannot appear.

This is the only spelling rule that should be enforced; nonsensical “words” like *xyzzy* are perfectly valid while totally reasonable words like *pfeiffer* are not. Here are some valid and some invalid strings:

String	Valid	Note
xyzzy	Yes	No e or i at all
aiex	Yes	i before e
aeix	No	e before i
ceix	Yes	e before i after c
ciex	No	i before e after c
eight	Yes	igh combination
wiegh	No	iegh combination

I really have to apologize for this one: I saw the question as a simple one, in terms of a finite state machine using the shortcuts we talked about it. Without those, things are a whole lot harder, and in the first bunch of solutions I saw, nobody did it that way. So, I’ll be grading with a lot of slack...



<i>Points</i>	<i>Notes</i>
20	<i>Major components of language covered</i>
15	<i>Group such as "iegh" sequence missed</i>
15	<i>Lots of missing final states</i>
10	<i>Major troubles, such as no loop.</i>

2. (20 points) An email address consists of a user name, followed by an @, followed by a domain name.

A user name consists of an arbitrary string of length at least one, taken from the characters a-z and 0-9.

A domain name consists of one or more domain name components. Each component except the last one is an arbitrary string of length at least one, taken from the characters a-z and 0-9, followed by a period. The last domain name component is a string of length two to four, taken from the characters a-z.

Write a regular expression for email addresses.

Note: I realize this is a simplified description of email addresses. Trying to do a regular expression for real email addresses would be a bit ambitious for a midterm question!

<i>Points</i>	<i>Notes</i>
-1	<i>* instead of + after domain name components</i>
-2	<i>only one level of domain name before top level</i>
-2	<i>requires names to start with [a-z]</i>
-5	<i>only username, top-level</i>
-3	<i>only allow fixed-length top-level</i>
-2	<i>allow 0-9 in top-level</i>
-2	<i>allow upper case</i>
-1	<i>allow underscore</i>
-2	<i>allow local domain</i>
-1	<i>allow 2-6 characters in toplevel</i>

3. (40 points) BIND is the most common domain name server on the internet (and quite a few other programs have configuration files with a very similar file format). The BIND configuration file format can be described like this:

The configuration file has a list of zero or more entries.

An entry can be either a key-value pair or a nested complex type.

A key-value pair has a string for the key, an equal sign, a value (which can be either a string enclosed in quotes or an integer), and a semicolon.

A nested complex type has a string for a key, a left brace, a list of zero or more entries, and a right brace.

Spaces, newlines, etc. are not significant.

Here's an example:

```
options {
    foo = "bar";
    bar = -1;
    baz {
        what = "hum";
        hum = "foo" ;
    }
}
darnit = "bar";
```

- (a) (20 points) Write a context free grammar for BIND configuration files. Assume tokens have been defined for all the relevant single-character punctuation ({ , ; = , etc), for strings, and for integers.

```
entrylist : entry entrylist
|
;

entry : kvpair ';'
| STRING '{' entrylist '}'
```

```

;
kvpair : STRING '=' value

value : integer
      | '"' STRING '"'
;

```

Points	Comments
-5	No recursion (lists have to be length 0 or 1)
-1	No quotes around string

(b) (20 points) Give an abstract syntax tree for the example configuration file in the question.

```

list
  entry
    options
    list
      entry
        foo
        "bar"
      entry
        bar
        -1
      entry
        baz
        entry
          what
          "hum"
        entry
          hum
          "foo"
    entry
      darnit
      "bar"

```

4. (20 points) Give a PDA that will recognize the set of all strings consisting of the letters a and b, such that the string starts with a, ends with b, and contains twice as many b's as a's. You can use any representation of the PDA that's clear and unambiguous.

I'm sorry... the problem is conceptually a pretty simple combination of a couple of other concepts, but working out the details is pretty awful. I'm putting the answer down here, but I won't be grading this one...

This one just needs a straightforward combination of a counter PDA to make sure there are twice as many b's as a's, with a FSM in the states to make sure we start with an a and end with a b. First, let's build the counter – I'll emphasize that I don't care about the order of a's and b's by leaving that part of the productions blank (for now).

$T(a, _ \epsilon) = (_, XX)$

$T(b, _ X) = (_, \epsilon)$

$T(b, _ \epsilon) = (_, Y)$

$T(a, _ YY) = (_, \epsilon)$

$T(a, _ Y) = (_, X)$

Probably worth noting a few things here: the first two rules handle the cases where we've seen more a's than b's: the first one pushes two X's every time it sees an a, so it knows it needs to see two b's. the second one knocks one off that imbalance every time it sees a b.

Then next two rules handle the cases where there have been more b's than a's: if we see a b, we push a Y on the stack to mark our b imbalance, and when we see an a we knock two Y's off the stack.

The last rule handles a case where we've only seen an odd number of b's, by changing a single-a deficit to a single-b deficit.

Now we need a FSM to enforce the "starts with a, ends with b". Hmmm, my intent was that there must be at least one character, but I don't see that in the question... but here goes:

$$T(a, s_0) = s_a$$

$$T(a, s_a) = s_a$$

$$T(b, s_a) = s_b$$

$$T(b, s_b) = s_b$$

$$T(a, s_b) = s_a$$

$$S = s_0$$

$$A = \{s_b\}$$

All that's left is to combine them:

$$T(a, s_0, \epsilon) = (s_a, XX)$$

$$T(a, s_a, \epsilon) = (s_a, XX)$$

$$T(a, s_b, \epsilon) = (s_a, XX)$$

$$T(b, s_a, X) = (s_b, \epsilon)$$

$$T(b, s_b, X) = (s_b, \epsilon)$$

$$T(b, s_a, \epsilon) = (s_a, Y)$$

$$T(b, s_b, \epsilon) = (s_b, Y)$$

$$T(a, s_a, YY) = (s_a, \epsilon)$$

$$T(a, s_b, YY) = (s_a, \epsilon)$$

$$T(a, s_a, Y) = (s_a, Y)$$

$$T(a, s_b, Y) = (s_a, Y)$$

What we wound up with here is just every possible combination of stack rules and transition rules from before.