YOUR NAME PLEASE:

*** SOLUTIONS ***

(corrected)  DA  4/10/17

Computer Science 201
Second Exam (corrected version)

April 7, 2015
7-8:30 pm

Closed book and closed notes. Show ALL work you want graded on the test itself.

For problems that do not ask you to justify the answer, an answer alone is sufficient. However, if the answer is wrong and no derivation or supporting reasoning is given, there will be no partial credit.

GOOD LUCK!
1. We define an "ordered tree" data structure as follows:
Base case: a Racket number or symbol is an ordered tree.
Recursive case: if T1 and T2 are ordered trees, then there
is an ordered tree T whose left subtree is T1 and whose right
subtree is T2.

This is reflected in the following Racket struct definition:

(struct ot (left right) #:transparent)

Recall that this defines a constructor ot, a type predicate ot?,
and selectors ot-left and ot-right, for the left and right subtrees.
Examples of ordered trees:
(define t1 'hi!)
(define t2 137)
(define t3 (ot t1 t2))
(define t4 (ot (ot 13 4) (ot (ot 'a 'b) 52)))

1.(a) (6 points)
Write a *single* Racket procedure (ordered-tree? value)
that takes an arbitrary Racket value and returns #t
if it is an ordered tree according to the definition
above, and #f otherwise.

Examples:
(ordered-tree? t1) => #t (and similarly for t2, t3, t4 above)
(ordered-tree? #t) => #f
(ordered-tree? (ot (ot 'leaf1 '(a list)) 'leaf3)) => #f

(define (ordered-tree? value)
  (or
   (symbol? value)
   (number? value)
   (and
    (ot? value)
    (ordered-tree? (ot-left value))
    (ordered-tree? (ot-right value)))
))
1.(b) (3 points)
Write a *single* Racket procedure (leaves tree) that takes one ordered tree and returns a list of all "leaves" of the tree, that is, all the number or symbol values in the tree, in the left-to-right order in which they occur.

Examples:
(leaves 'hi!) => '(hi!)
(leaves 137) => '(137)
(leaves (ot 'x 'y)) => '(x y)
(leaves (ot 1 (ot (ot 2 1) (ot 'b 'a)))) => '(1 2 1 b a)

(define (leaves tree)
  (if (ot? tree)
      (append (leaves (ot-left tree))
              (leaves (ot-right tree)))
      (list tree)))

1(c) (3 points)
Write a *single* Racket procedure (rt tree) that takes an ordered tree and returns an ordered tree that is the reverse of the given tree, that is, the tree that would be obtained by "flipping over" the given tree.

Examples:
(rt 137) => 137
(rt 'hi!) => 'hi!
(rt (ot 'a 'b)) => (ot 'b 'a)
(rt (ot (ot 1 (ot 2 3)) (ot (ot 4 5) 6))) =>
  (ot (ot 6 (ot 5 4)) (ot (ot 3 2) 1))

(define (rt tree)
  (if (ot? tree)
      (rt (ot-left tree))
      (rt (ot-right tree)))
      tree))
2. (a) (6 points)
Construct a combinational (that is, loop-free) circuit to compute the outputs c and d from the inputs a and b, given the following truth table:

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

\[ c = a' \cdot b' + a \cdot b \]
\[ d = b' \]

Use only the following types of gates: NOT, 2-input AND, 2-input OR. You may use rectangles (labeled with gate types) for gates. Make sure your input and output wires are labeled.

State how many gate delays your circuit uses, with a justification.

3 gate delays
2. (b) (4 points)
Consider the sequential (that is, "loopy") circuit with wires
s, t, u, v, and two gates arranged so that

\[ u = \text{NAND}(s, v) \]
\[ v = \text{OR}(u, t) \]

Draw a diagram of this circuit (rectangles with labels indicating
the gate type are fine.) Also, find all the stable configurations
of its four wires, and justify your answer.

\[
\begin{array}{cccc}
S & T & U & V \\
0 & 0 & 1 & 1 \\
0 & 1 & 1 & 1 \\
1 & 1 & 0 & 1 \\
\end{array}
\]

If \( s = 0 \) then \( u = 1 \) and \( v = 1 \), regardless of \( t \), so
\( s = 0, t = 1, u = 1, v = 1 \) is stable
\( s = 0, t = 1, u = 1, v = 1 \) is stable

If \( s = 1 \) and \( v = 1 \) then \( u = 0 \), so the only stable
possibility is \( t = 1 \) (otherwise, \( v = 0 \))

If \( s = 1 \) and \( v = 0 \) then \( u = 1 \), and this is
not stable, because \( v \) would become 1.

2. (c) (2 points)
Briefly explain what it means for a set of types of gates to be
a Boolean basis, and give two different examples of Boolean bases.

A set of types of gates is a Boolean basis
if for any Boolean function \( f \), it is
possible to design a circuit for \( f \) using
only the types of gates in the set.
Examples of Boolean bases: \{\text{AND, OR, NOT}\},
\{\text{AND, OR, NOT}\}, \{\text{AND, NOT}\}, \{\text{OR, NOT}\}, \{\text{NAND}\}, \{\text{NOR}\}.
3. Recall that the TC-201 assembly-language instructions are:
   halt, load address, store address, add address, sub address,
   input, output, jump address, skipzero, skippos, skiperr,
   loadi address, storei address

Also the directive: data number
reserves one memory location and stores the number in it.

3.(a) (8 points)

Write a symbolic assembly-language program to read in a zero-terminated
sequence of numbers in the range -1000 to 1000, then print out the
**smallest** non-zero value, and halt. Please use symbolic opcodes
and symbolic (not numeric) addresses. You may assume that there will
be at least one nonzero number in the sequence.

An example of the behavior of your program:

```
input = 17
input = -4
input = -2
input = 0
output = -4

Next-input:
   input
   skipzero
   jump compare
   jump done

Compare:
   store num
   load min
   sub num
   skippos
   jump next-input
   load num
   store min
   jump next-input
   done:
   load min
   output
   halt

Num: data 0
Min: data 1001
```
3.(b) (4 points)
Briefly describe each of the following:

(i) 8-bit sign/magnitude representation of numbers.
   The first bit gives the sign: 0 for +, 1 for -.
   The other 7 bits give the magnitude (absolute value) in 7-bit unsigned binary.

(ii) random access memory in the TC-201.
   The random access memory (RAM) of the TC-201 is 4096 registers, addressed from 0 to 4095,
   each of which contains 16 bits.

(iii) what the store and storei instructions do in the TC-201.
   The instruction "store address" copies the contents of the accumulator to the memory register at the given address. The instruction "storei address" copies the contents of the accumulator to the memory register whose address is given by the low-order 12 bits of the contents of the memory register at the given address.

(iv) why the TC-201 cannot simulate a Turing machine.
   The TC-201 has a finite amount of memory (4096 x 16 + 16 + (2+2) bits suffice to describe a configuration completely.) A Turing machine may use an unbounded amount of memory.
4. (a) (6 points)
Consider the alphabet of symbols \( \{x, y, z\} \).
Write a "basic regular expression" for each of the following sets of strings over this alphabet. Recall that in a basic regular expression we have operations for union (\( | \)), repetition 0 or more times (\( * \)), and concatenation (centered dot).

(i) The set of strings that have at least one \( z \).
\[
(x|y)^* z (x|y|z)^*
\]

(ii) The set containing just the four strings: \( xy, yz, zx, zz \).
\[
x y \mid y z \mid z x \mid z z
\]

(iii) The set of strings of length at least 3 that have a \( y \) as the third-to-last symbol.
\[
(x|y|z)^* y (x|y|z)(x|y|z)
\]

(iv) The set of strings that consist either of an even number of \( x \)'s followed by an even number of \( y \)'s or an odd number of \( x \)'s followed by an odd number of \( z \)'s. (Note that 0 is an even number.)
\[
(x x)^* (y y)^* \mid x (x x)^* z (z z)^*
\]

(v) The set of strings that contain at most 2 of the 3 symbols \( x, y, z \).
\[
(x|y)^* \mid (y|z)^* \mid (x|z)^*
\]

(vi) The set of strings that do not contain two consecutive \( x \)'s.
\[
(y \mid z \mid x (y|z))^* (x \mid \varepsilon)
\]
4.(b) (6 points)
Let the alphabet of symbols be \{x, y, z\}. Construct a deterministic finite state acceptor for the set \( L \) of all strings over this alphabet that do not contain \( xy, yz, \) or \( zx \) as a contiguous substring.

Examples of strings in \( L \): empty string, \( z, xx, yyyx, yxz, xxxzzyyyy \)
Examples of strings not in \( L \): \( xy, xxxzzx, yxy, zzzxyyyzzx \)

Recall that a finite state acceptor is deterministic if it has at most one transition defined for every state and symbol. Your finite state acceptor may be incomplete -- that is, it may omit definitions of transitions for some state and symbol pairs. A diagram is fine. Be sure to indicate the start state and the accepting state(s).
5. Give brief answers to the following questions (3 points each):

5.(a) Give the 16-bit sign/magnitude representations of two numbers that would cause the arithmetic error bit to be set if they are added using the TC-201 add instruction.

\[
\begin{align*}
0 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \\
0 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \quad 1 & \\
\end{align*}
\]

These are both = 32,767 (in sign/magnitude or two's complement), and their sum cannot be represented in 16-bit sign/magnitude or 16-bit two's complement.

5.(b) What is the result of the Linux command

grep -E "th(is|at)" temp.txt

This lists all the lines of the file temp.txt that contain "this" or "that" as a contiguous substring.
5. (c) Let L be a regular language over an alphabet A. Define the complement of L, denoted L', to be the set of strings over A that are not in L. Must L' be regular? Why or why not?

Yes, L' is regular. Because L is regular, there is a complete DFA M that recognizes L. If we make all the accepting states of M rejecting, and all the rejecting states of M accepting, we get a new DFA M' that accepts all the strings M rejects and rejects all the strings that M accepts. Thus, \( L(M') = L' \), so L' is regular.

5. (d) Consider the gate G with 3 inputs a, b, c and one output d whose truth table is shown below.

Is G by itself a Boolean basis? Why or why not?

<table>
<thead>
<tr>
<th>a</th>
<th>b</th>
<th>c</th>
<th>d</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

Yes, \( y = G(x, x, x) \) gives \( \text{NOT} \):

\[
\begin{array}{c|c|c}
    x & G(x, x, x) & 0 \\
    \hline
    0 & 1 & 1 \\
    1 & 0 & 1 \\
\end{array}
\]

And \( z = G(x, y, x) \) gives \( \text{NAND} \):

\[
\begin{array}{c|c|c|c}
    x & y & G(x, y, x) & 0 \\
    \hline
    0 & 0 & 1 & 1 \\
    0 & 1 & 1 & 1 \\
    1 & 0 & 1 & 1 \\
    1 & 1 & 0 & 1 \\
\end{array}
\]

\( \text{NAND?} \) is a Boolean basis.