COS 126	General Computer Science	Fall 1998

Midterm 2 - Solutions

We could have answered all of these exam questions correctly without writing very much. Instead, we will do just the opposite here! For many of these problems, we will provide detailed solutions. In some cases, we will even raise and answer questions that were not asked for. Even if you aced the exam, you may still benefit from reading these solutions.

Question 1

Any Boolean function can be represented using only AND, OR, and NOT gates. In fact, there are always infinitely many to do it. The sum-of-products form is a mechanical way to convert any Boolean function into a circuit using AND, OR, and NOT gates. This method may not produce the circuit with the fewest number of gates, but it always works. The sum-of-products form for this expression is $x'yz + xy'z' + xyz'$. This can be expressed pictorially using three AND gates, one OR gate, and 4 NOT gates.

Note also that $x'yz + xy'z' + xyz' = x'yz + x(y' + y)z' = x'yz + xz'$, so this circuit could have been more efficiently described using two AND gates, one OR gate and two NOT gate. This was just meant to illustrate how we can build circuits more efficiently, there is no need to do such circuit optimization on an exam.

Question 2

The circuit can be expressed as the Boolean function $f = (xy)'(y + z)$. The second term says that either y or z or both must be 1. The first term says that either x or y or both are 0.

x	y	z	f
0	0	0	0
0	0	1	1
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	0

Question 3

A

This question is easy, assuming you know how a decoder, adder, and multiplexer work. Otherwise, it is hopelessly difficult. So let's review these fundamental circuits.

A *decoder* has n input wires and 2^n output wires, where n is some positive integer. Interpret the input wires as an n-bit binary number (read from left to right). E.g., for $n = 4$, if the input wires are 1 1 0 1, then we can interpret this as representing the number 13. The decoder outputs 1 on output wire 13 and 0 on the remaining 15 wires (the output wires are numbered 0 to $2^n - 1$ from top to bottom). A decoder is a very useful circuit: for example if the outputs were memory, then inputting 1 1 0 1 puts a 1 into memory location 13.

A multiplexer has n input wires on the top and 2^n on the left. It has a single output. As with a decoder, we interpret the top input wires as an n-bit binary number. We select the left input wire corresponding to

this binary number. The on/off state of this wire becomes the output of the multiplexer. E.g., if the top inputs are 1 1 0 1, then we look at the contents of left input wire number 13 (where the wires are numbered from 0 to $2^n - 1$). If wire 13 is on, the single output wire is on; if wire 13 is off, the output wire is off. This is a very useful circuit. It allows us to retrieve a single bit from the contents of the left wires (which could be memory).

An n-bit adder has $2n$ inputs and n outputs. We interpret the first n bits as a binary number which we denote by x, and the second n bits as the binary number which we denote by y. The adder outputs the number $x + y$ expressed in binary. E.g., if the input wires are 0 0 1 0 0 0 1 0, then $x = y = 2$. The output wire will be the binary representation of $x + y = 4$ or 0 1 0 0. (The adder handles overflow much like a real computer - the input 1 1 0 1 1 0 0 1 would yield 0 1 1 0. Adding the binary numbers 1101 and 1001 gives 10110 but the adder chops of the leading 1.)

Here's the truth table for the relevant part of this Boolean function of 5 variables.

s	t	u	v	w	f
0	0	0	0	0	1
0	0	0	0	1	0
0	0	0	1	0	0
0	0	1	0	0	0
0	1	0	0	0	0
1	0	0	0	0	1

E.g., if all inputs are 0, then the decoder outputs a 1 on wire 0. Thus, the input to the adder is 1 0 0 0 0 0 0 0. The adder adds 4 and 0 and outputs 1 0 0 0. The multiplexer selects its left input line 0. Since this input line is 1, the multiplexer outputs 1.

Suppose s is switched to 1. Now the decoder outputs a 1 on wire 4. The input to the adder is 0 0 0 0 1 0 0 0. The adder adds 0 and 4 and outputs 1 0 0 0. The multiplexer outputs a 1 as before, since it has the exact same inputs.

Question 4

```
link remove0(link x)
{
  if (x == NULL) return x;
  x->next = remove0(x->next);
  if (x->key == 0) return x->next;
  else return x;
}
```

The answer is deceptively simple, but requires quite a bit of thought to get right. First lets see what our function prototype (input and output variables) should look like. When we call `remove0`, we must pass a pointer to the first node of the linked list; otherwise it would have no clue as to which linked list to examine. Thus the function should have one input argument - we use the `link x` to denote a pointer to the first node of the linked list. Now, what should the function return? Since we must allow the possibility of deleting the first node on our linked list (in the event that its key is 0), the function `remove0(link x)` should return a link to the new first node, i.e., be `link remove0(link x)` and not `void remove0(link x)`.

Now we describe the idea behind the function. The variable `link x` is a pointer to the first node of linked list of size, say n. Clearly if `x` is `NULL` we should return `NULL`. This is the base case.

Now, suppose we knew how to remove all of the 0's from the $n - 1$ node linked list starting at `x->next`. If `x->key` is 0, then we should return just the 0-free linked list starting at `x->next`. Otherwise if `x->key` is nonzero, we should return the same thing, but now starting at `x` instead of `x->next`, since we don't want to delete `x`.

So we have reduced our problem to removing all of the 0's from the (smaller) linked list starting at `x->next`. But this is the magic of recursion (induction). In summary - we have shown (i) that our code

works for an empty linked list and (ii) if our code works for an $n-1$ node linked list then it will also work for an n node linked list. So, by induction, our code works for any linked list and we are done! (In practice, it would be a good idea to free up the memory reserved for the node that `x` points to.)

The following *faulty* solution (and equivalent variants) were given by many students. So it is worth considering in a little more detail, even if you didn't follow this approach.

```
/* This function has a serious bug. */
void remove0(link x)
{
  if (x == NULL || x->next == NULL) return;
  if (x->next->key == 0) x->next = x->next->next;
  remove0(x->next);
  return;
}
```

The obvious flaw is that if the initial key is 0, it does not delete it. In fact, it never even refers to `x->key`, only `x->next->key`. So at first glance, you might think this function deletes all of the 0's in the list, except for possibly the first node if its key is 0. E.g., if the initial list is 01230000045600007809, it would return 0123456789

However, this is a recursive function, so the single flaw is propagated in each recursive call! Every time that the function is called with a linked list whose first node has a key of 0, it overlooks deleting that node. In fact the function applied to the list above would return 01230045600789. If there are consecutive 0's, it only deletes (roughly) half of them. If you aren't convinced, try it out for yourself. Sometimes recursive functions are hard to analyze (just ask the grader for this question!).

Question 5

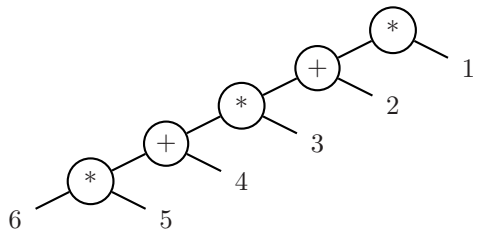
104.

There are several ways to do this problem.

- if you see a + or * followed by two numbers, do the arithmetic, replace the result, and recurse

```
* + * + * 6 5 4 3 2 1
* + * + (* 6 5) 4 3 2 1
* + * + 30 4 3 2 1
* + * (+ 30 4) 3 2 1
* + * 34 3 2 1
* + (* 34 3) 2 1
* + 102 2 1
* (+ 102 2) 1
* 104 1
104
```

- In this solution we will construct the parse tree associated with the prefix expression. The arithmetic symbols + and * will have exactly two children; the numbers are not allowed to have children (much like `NULL` in ordinary binary trees). Now the idea is to fill up the tree from top to bottom, then left to right (just like a preorder traversal). Once the parse tree is constructed, it is easy to evaluate the expression. Start at the bottom and work your way up the tree, just as the recursive prefix program would work its way up through the recursive calls. So 6 * 5 = 30. Then 30 + 4 = 34. 34 * 3 = 102. 102 + 2 = 104. 104 * 1 = 104.

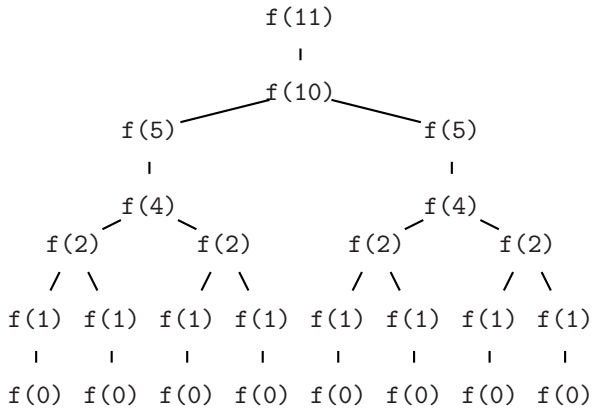


The prefix expression is actually the preorder traversal of this parse tree! This is the correlation between prefix and preorder. Now, guess what the postorder traversal corresponds to? Yes, the postfix expression! In this case it is `1 2 3 4 5 6 * + * + * +`, which also evaluates to 104. The postfix expression is nothing more than the original prefix expression read backwards. So what does the inorder traversal correspond to? It would give `6 * 5 + 4 * 3 + 2 * 1`. Using the conventional arithmetic grouping rules, this evaluates to (6*5) + (4*3) + (2*1) = 44, which is probably not what you expected. To form an equivalent infix expression, you must specify the parentheses appropriately: `((((6 * 5) + 4) * 3) + 2) * 1` = 104.

Question 6

It returns 11 after 1 + 2 + 2 + 4 + 8 + 8 = 25 recursive calls. In general, `f(n) = n`, and it turns out the number of recursive calls is $2^{\lfloor \lg n+1 \rfloor} + n - 2$.

Recall, a recursive function is a function that calls itself. First we try to evaluate `f(11)` from `main`. This calls the function `f`, and hence is not a recursive call. When we try to evaluate `f(11)`, we see that 11 is not divisible by 2, so we return `1 + f(10)`. So we need to (recursively) compute `f(10)`.Now, since 10 is divisible by 2, the call to `f(10)` returns `f(5) + f(5)`. To evaluate this, we need to (recursively) call `f` twice, each time with the parameter 5. (Our program is not smart enough to realize that it is computing the same thing twice. A more clever way would be to store all computed values in a table. Then before computing `f(n)`, first check to see if you've already computed it. This is called *dynamic programming*.) The full tree of recursive calls is given below. Note that to compute `f(1)` we compute `f(0)` recursively, so these calls are counted in the total.



Question 7

`caravan`
`caraway`

The command `egrep '^(.a)*.$' test` will print out all lines in `test` such that all of the even numbered characters are `a`, i.e., `caravan` and `caraway`.

Let's see why. In `grep` and `egrep` the period matches any single character. Thus `.a` matches any two consecutive characters in which the second character is an `a`. By using the `*` (replication) operation `(.a)*` matches things like `gagadada` and `salamander`, but also `gggggg` and the empty string. Why does it match these? Remember, zero replications are allowed! Finally, the `^` matches the beginning of the line and the `$` matches the end of the line. E.g., `^ba` matches `banana` but not `rutabaga`. Putting all of this together gives the answer above.

Question 8

`caraway`

Let's see why. The program `awk` scans each line in the input file sequentially. For each input line, it checks to see if any of the search criteria are matched. If so, it performs the statement(s) between the pair of curly braces. There are two search criteria in the command `awk '/^.a.a.a.$/ {s = $1} END {print s}' test` The first is `/^.a.a.a.$/`; the second is `END`. As indicated on the exam, the slashes delimit patterns in `awk`. The pattern `^.a.a.a.$` matches all 7 letter lines whose even letters are `a`, i.e., `caravan` and `caraway` (this is very similar to the search pattern in question 7). The criteria `END` only matches the end of file. No criteria is matched on the first 3 lines of input, so nothing is done. The fourth line of input `caravan` matches the `/^.a.a.a.$/` criteria; thus the statement `s = $1` is performed. This assigns the variable `s` the contents of column 1 on this line, namely `caravan`. (Nothing is printed.) The 5th line of input also matches the same search pattern so now the variable `s` is assigned the value `$1`, which is `caraway`. No criteria are matched on lines 6-12. Finally `EOF` matches the end of file. The statement `print s` prints out the contents of variable `s`, which is now `caraway`.

Question 9

State 6

It's not necessary to write out the FSA graphically. You start in state 0. The first input bit is 0. The table indicates that if you are in state 0 and read a 0, you go to state 0 (if you are in state 0 and read a 1 you would go to state 3). The next input bit is 1. Since you are in state 0, you would go to state 3. Repeating this process, we see that the FSA will end up in state 6, as indicated below.

$$0 \xrightarrow{0} 0 \xrightarrow{1} 3 \xrightarrow{1} 2 \xrightarrow{1} 5 \xrightarrow{0} 0 \xrightarrow{1} 3 \xrightarrow{0} 7 \xrightarrow{1} 7 \xrightarrow{0} 6 \xrightarrow{0} 7 \xrightarrow{0} 6$$

Question 10

C

The easiest way to answer this question is to realize that A, B, D, E, and F all match 11. But C does not - hence C must be the right answer. Note that we didn't even need to verify that A, B, D, E, and F really do all describe the same language! However, it is not hard to see that A, B, D, E, and F actually do describe the same language. Each of them matches all bit strings that start with a 1. In contrast, C matches all *odd length* bit strings that start with a 1.

How does the RE `1 + 1(0+1)*0 + 1(0+1)*1` in part F work? The first term matches the bit string 1. The second matches any bit string starting with a 1 and ending with a zero. The third matches any bit string starting with a 1 and ending with a 1. The RE accepts either of the three possibilities - this consumes all possible bit strings that start with a 1.

Question 11

A, B, C, D, E, F

One way to answer the remaining parts is to stare at a diagram of the FSA until you find a path of the desired form. We'll try to give a faster method here. You probably did something quite similar.

A nondeterministic FSA accepts all input strings for which there is some possible path from the starting state that ends exactly in the accept state. (It must also use up all of the characters in the input string). So the bit string 1 is accepted because we could do $0 \xrightarrow{1} 3$. 111 is also accepted since $0 \xrightarrow{1} 3 \xrightarrow{1} 0 \xrightarrow{1} 3$ is a possible path. This cycle implies that any odd length bit string of all 1's is accepted, i.e., corresponding to the RE 1(11)*. The bit string 1111111111111 (13 1's) in part F is thus accepted.

Consider the following subset of the FSA: (i) $0 \xrightarrow{0+1} 2$ - this notation means if we start in state 0 and the next input bit is 0 or 1 (0 + 1) then we can go to state 2 (ii) $2 \xrightarrow{0+1} 2$, (iii) $2 \xrightarrow{0+1} 1$, and (iv) $1 \xrightarrow{0} 3$. Combining (i), (iii), and (iv), we see that any 3-bit input string ending in 0 is accepted, since the path $0 \xrightarrow{0+1} 2 \xrightarrow{0+1} 1 \xrightarrow{0} 3$ would provide such a proof. Now using (iii), we see that any 4-bit input string ending in 0 is accepted, since the path $0 \xrightarrow{0+1} 2 \xrightarrow{0+1} 2 \xrightarrow{0+1} 1 \xrightarrow{0} 3$ would provide such a proof. Similarly, any bit string ending in 0 with 3 or more bits is accepted - to see this consider the following path.

$$0 \xrightarrow{0+1} 2 \xrightarrow{0+1} 2 \xrightarrow{0+1} \ldots 2 \xrightarrow{0+1} 2 \xrightarrow{0+1} 1 \xrightarrow{0} 3$$

Many many strings (that end in zero) are of this form and hence are accepted by the FSA. This form (written as a RE) is `(0+1)(0+1)*(0+1)0`. The bit strings in A, B, C, D, and E are all of this form, and the problem is solved.

Above, we demonstrated that all strings of the form `1(11)* + (0+1)(0+1)*(0+1)0` are in the language. (We do not claim this is actually an exhaustive list.) In general, it is harder to prove that a bit string is not in the language. For this automaton, it is actually not hard to see that any even length bit string of all 1's is not accepted. To see this, note that we cannot get from either state 1 or state 2 to state 3 using any path consisting entirely of 1's. Thus, to have any hope of accepting a bit string of all 1's, we must bounce back and forth between states 0 and 3. If there are an odd number of all 1's we end up in state 3 and accept (as discussed above); it is not possible to end up in state 3 with an even number of 1's so the FSA rejects bit strings of the form (11)*.

Question 12

B - *regular expressions are not capable of counting.* What this means is that RE's are not capable of matching bit strings with equal number of 0's and 1's. It does not mean that people cannot count. So RE's are perfectly capable of matching all bit strings with at least 123 0's (see below).

Regular languages are languages that can be expressed as regular expressions. For example, consider the language consisting of all bit strings ending in 0, i.e., $\{0, 00, 10, 000, 010, \ldots\}$. It is regular since it can be expressed by the RE (0+1)*0. Remember, FSA's and RE's are equally powerful, so regular languages are also precisely those languages accepted by FSA's.

We actually indicate how to construct the corresponding RE's for parts A, C, D, and E (even though this isn't actually necessary to answer the question, it is still good practice). Since 123 is a big number, we demonstrate the corresponding RE's if we replace 123 by 3. The same logic would apply for 123, but the expression would be much too large to be comprehensible. Parts C, D, and E are a bit easier to digest, so you may wish to start there.

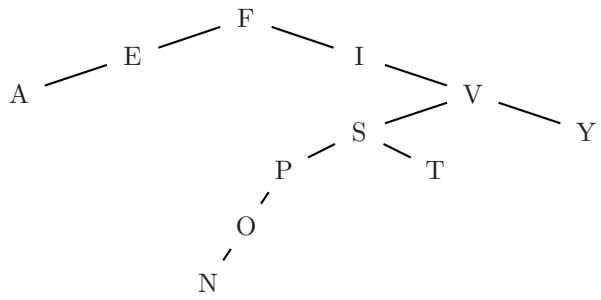
- A - A FSA that test divisibility by 3 is given in Lecture 15. Using the exact same logic, we could build a FSA that tests divisibility by 123. And remember, FSA's and RE's are equally powerful, so if an FSA can do it, then so can a RE!
- B - FSA's cannot decide whether or not its input has the same number of 0's and 1's. This was proved mathematically in Lecture 15. Using the same argument (or by taking a small leap of faith), FSA's

cannot determine whether or not a bit string has 3 more 0's than 1's. Remember, FSA's and RE's are equally powerful, so if you can't do it with a FSA, you can't do it with a RE.

- C - the following RE accepts all bit strings with more than 3 zeros: (0+1)*0(0+1)*0(0+1)*0(0+1)*. The RE (0+1)* matches any number of 0's and 1's in any order. It is an important component of many more complicated RE's.
- D - the following RE accepts all bit strings with at least 3 consecutive 0's: (0+1)*(000)(0+1)*
- E - the following RE accepts all bit strings whose number of 0's is a multiple of 3: (1*01*01*01)*1*. The 1* at the end is used to accept strings with no zeros.

Question 13

Recall the BST property: for any node `x`, all of the keys in its left subtree are smaller than the key of node `x`, and all of the keys in the right subtree are bigger (assuming no duplicate keys). F is the root node. The next node to insert is I. I occurs after F in the alphabet so must be the right child of F. The next key is V. V occurs after F in the alphabet so must be in the right subtree of F. V is also after I, so it becomes the right child of I The next input is E. It occurs before F in the alphabet, so it becomes the left child of F. Continuing along in this fashion (and discarding duplicate keys) we obtain the following BST.



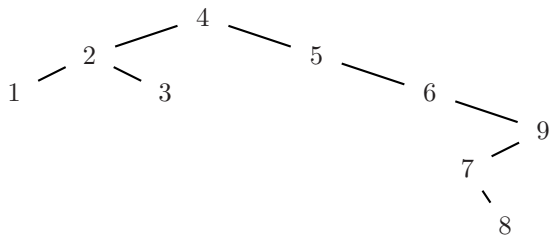
You should become aware of one important flaw of standard BST's. Suppose the list of keys to insert are already sorted. What will the BST look like?

Question 14

Recall the preorder traversal visits (displays) a node before it displays any node in its left or right subtrees. Thus, we know that 9 cannot be an ancestor (parent, grandparent, etc.) of node 3 (or any node that appears earlier on the list). Similarly, no node can be an ancestor of node 4 since 4 appears first on the list, i.e., 4 must be the root. Knowing the preorder traversal gives us lots of information about the structure of the original tree. However it does not uniquely determine the tree (if we are not given the places where the external nodes appear in the traversal).

If we add the extra condition that the original tree was a BST, then it turns out the tree will be uniquely specified. Recall the BST property: for any node `x`, all of the keys in its left subtree are smaller than the key of node `x`, and all of the keys in the right subtree are bigger than the key of `x` (assuming no duplicate keys).

By the preorder property, node 2 can only have node 4 as an ancestor. By the BST property, if node 2 is a descendant of node 4, it must be in its left subtree. Combining these two facts, node 2 must be the left child of root node 4. Using similar logic, we can reconstruct the whole BST given just the preorder information.



Note that is really the same as Question 13 (which you probably thought was much easier) - reconstructing the BST turns out to be the same as inserting the keys into an empty BST in the preorder order.