COS 126	General Computer Science	Spring 1999

Midterm 2 - Solutions[1]

1. Write a C function `int count(char s[ ])` that takes as input a '\0' terminated string and outputs the number of characters in the string (not including the '\0'). Do **not** use any library functions or pointer arithmetic.

```
int count (char s[])
  {
    int i = 0;
    while (s[i] != '\0')
      i++;
    return i;
  }
```

See also String Exercise 1 in the course packet.

2. Consider the following recursive C program.

```
void mystery(int N)
  {
    if (N < 1) return;
    printf("%d ", N);
    mystery(N-2);
    mystery(N-3);
    printf("%d ", N);
  }
```

Give the results of `mystery(6)`. Circle the correct answer.

(a) 2 4 1 6 1 3
(b) 2 1 4 1 3 6
(c) 6 4 2 1 3 1
(d) 6 4 2 2 4 1 1 6 3 1 1 3
(e) 6 4 2 2 1 1 4 3 1 1 3 6
(f) 6 3 1 1 6 4 2 2 4 1 1 6
(g) 6 3 1 1 3 4 1 1 2 2 4 6
(h) segmentation fault

e

The very first and very last thing that `mystery(N)` does is print the integer `N`. So the answer must start and end with 6. This eliminates everything but (e), (f), and (g). Just after printing the first 6, `mystery(6)` calls `mystery(4)`. The very first thing that `mystery(4)` does is print 4, so the second number printed is 4. This leaves only (e).

[1]Copyright 1999, COS 126.

3. What does the following TOY program print out? Assume that the following numbers are loaded into memory and that the machine is started with the PC set to `10`.

```
10: B004       R0 <- 4
11: B101       R1 <- 1
12:   3101     R1 <- R0 * R1
13:   4100     print R1
14:   7012     R0--;  if (R0 > 0) goto 12
15: 0000
```

`0004 000C 0018 0018`

The program is a single loop which decrements `R0` in each iteration. In each iteration `R1` is multiplied by `R0` and `R1` is printed. Thus the function prints $n,\ n\times(n-1),\ \ldots,\ n\times(n-1)\ldots\times 2,\ n\times(n-1)\ldots\times 2\times 1$, where n is the initial value assigned to `R0`. Note the last two terms are both $n!$. Also, be careful to do all arithmetic in hex.

4. What does the following TOY program print out? Assume that the following numbers are loaded into memory and that the machine is started with the PC set to `10`.

```
10: B000  R0 <- 0                  40: 0001
11: B101  R1 <- 1                  41: 0046
12: B240  R2 <- 40                 42: 0002
13: 9B02    R3 <- mem[R2 + 0]      43: 0048
14: 4300    print R3               44: 0003
15: 9A12    R2 <- mem[R2 + 1]      45: 0000
16: 6213    if (R2 > 0) goto 13    46: 0004
17: 0000  halt                     47: 0042
                                   48: 0005
                                   49: 0044
```

`0001 0004 0002 0005 0003`

The `PC` will never be set to `40-49` so you can think of these memory locations as storing data. The guts of the program is in lines `10-17`.

Throughout the computation `R0 = 0` and `R1 = 1`. Indexed addressing is used in instructions `9B02` and `9A12` since the second hex digit is greater than or equal to 8. Line 13 reads in a value from memory location `R2` and line 14 prints it out. Think of `R2` as a pointer - it is the address in memory where some data resides. Line 15 updates `R2` to be the contents of memory address `R2 + 1`.

The memory address pair `40, 41` contains two pieces of information: the first is the data and is printed, the second is the memory address of the next piece of data. This process is repeated until the memory address is 0000 (i.e., NULL). The data in lines `40-49` is really a linked list!

5. Give the contents of the stack after the given PostScript program is executed.

```
1 2 3 4 5 dup add mul add dup add mul add
```

173

Here are the stack contents along the way, where the top of the stack is at the left.

```
1
2 1
3 2 1
4 3 2 1
5 4 3 2 1
5 5 4 3 2 1
10 4 3 2 1
40 3 2 1
43 2 1
43 43 2 1
86 2 1
172 1
173
```

6. Which strings are generated by the regular expression? $(111\ +\ 010)*011*$

Circle one or more of the following.

(a) 111010011
(b) 010000000
(c) 111111111111
(d) 11111101001001011100000111010111111
(e) 1011110101011101101100111010111100

(a)

First observe that the last bit must be 1: this eliminates (b) and (e). Second observe that there must be at least one 0: this eliminates (c). Third, observe that there can't be more than two consecutive 0's: this eliminates (b) and (d).

7. Consider the language generated by the following grammar.

terminals	$0, 1$
nonterminals	A, B
start	A

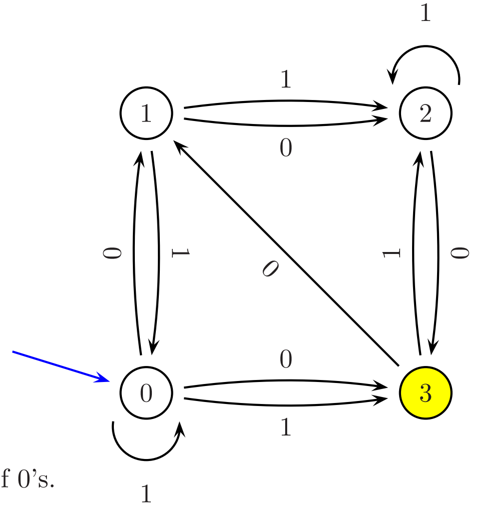
Rules
$A \to B0$
$B \to A1$
$A \to 0$

Is the language regular? If yes, give a regular expression that generates *exactly* the same language. If no, explain why not.

yes, `0(10)*` or `(01)*0`

The language is Type III (regular). This implies that there must exist a corresponding regular expression. The production rule $A \to B0$ must be immediately followed by $B \to A1$, so you could think of having the compound rule $A \to A10$. Thus, expressions of the following form could be generated: $A, A10, A1010, A101010, A10101010$. Ultimately, we must end up with strings. This is accomplished as soon as we apply the rule $A \to 0$.

8. Consider the following FSA.



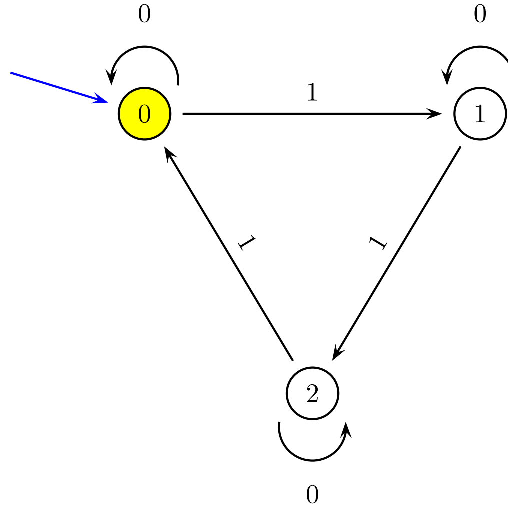
Circle all of the statements below that are true.

(a) The FSA accepts 011111101.
(b) The FSA accepts 11101000.
(c) The FSA rejects 0000.
(d) The FSA accepts all bit strings with an odd number of 0's.
(e) The FSA is nondeterministic.
(f) There exists a *deterministic* Turing machine that recognizes the same language as the FSA.
(g) There exists a *nondeterministic* Turing machine that recognizes the same language as the FSA.

b, e, f, g

The FSA is nondeterministic for many reasons: if you are in state 0 and the next bit is 0, you can transition to state 1 or 3. The n-FSA accepts 0000 since it could choose the following transitions: $0 \xrightarrow{0} 3 \xrightarrow{0} 1 \xrightarrow{0} 2 \xrightarrow{0} 3$. It accepts 11101000 because of $0 \xrightarrow{1} 0 \xrightarrow{1} 0 \xrightarrow{1} 0 \xrightarrow{0} 1 \xrightarrow{1} 0 \xrightarrow{0} 3 \xrightarrow{0} 1 \xrightarrow{0} 2$. The n-FSA will not accept `01` or `00000` - either of these eliminates (d). Also, the n-FSA does not accept any strings ending in `01` (including `011111101`) since the only way to get to state 3 after reading a 1 is to come from state 0, but there is no way to get to state 0 after reading a 0. Statement (f) is true since deterministic Turing machines are more powerful than FSA's or n-FSA's. Deterministic Turing machines can recognize exactly the same languages as nondeterministic ones, so (g) is also true.

9. Construct a deterministic finite state automaton (FSA) that recognizes all bit strings with a multiple of three 1's. (For example, the following strings are in the language: 111, 111111, 1110, 0111, 10011, but not 1, 11, 1111, 0110.) Use as few states as possible.



a / a / R
x / x / R
search b
b / x / L
a / x / R
/ # / R
a / a / L
b / b / L
x / x / L
x / x / R
left
/ # / R
skip x
/ # / R
yes
no
a / x / L
b / x / R
/ # / R
search a
x / x / R
b / b / R

10. Consider the Turing machine above. The alphabet consists of the four characters: `a`, `b`, `x`, and `#`. Suppose the Turing machine is started with the input tape and initial cursor as shown below. Circle all of the inputs below that will be accepted.

(a)

									↓							
							#	a	b	#						

(b)

										↓									
							#	a	b	b	a	#						

(c)

							↓									
			#	a	b	b	b	b	b	b	a	#				

(d) Characterize in plain English which inputs (i.e., characters between the two delimiting #'s) are accepted by the Turing machine.

(a), (b), all strings with an equal number of a's and b's

The TM will accept all strings with an equal number of a's and b's. To do this it first moves to the left end of the tape. If the first character (other than #) is an a, it will "delete" it by overwriting it with an x. Then the TM will search for the leftmost b. If it finds a b, it "deletes" it by overwriting it with an x. Otherwise, the TM concludes that there are more a's than b's and proceeds to state no. (The process is analogous if the first character is a b. Then it deletes the b and searches for an a to delete.) This whole procedure is repeated, until the string contains no more a's or b's. In this case, the TM will go to state yes. It works because every time it "deletes" an a, it also "deletes" a corresponding b, thereby reducing the size of the problem. Clearly the new smaller input has an equal number of a's and b's if and only if the original input did.

Here's a description of each state.

- `yes`: the accept state
- `no`: the reject state
- `left`: This state repeatedly moves the cursor one position to the left, unless the cursor already points to #. That is, `left` moves the cursor to the left #. If instead, the cursor is already pointing to #, then it transitions to `skip x` and moves the cursor one position to the right, i.e., to the beginning of the interesting portion of the tape.
- `skip x`: This state repeatedly moves the cursor one position to the right, until it reaches the first character that is not x. If the first such character is #, then it accepts the string; if it is a, the TM goes to state `search b` to find a b to "delete"; if it is b, the TM goes to state `search a` to find an a to delete.
- `search b`: This state skips over all a's and x's. If it finds a b, it "deletes" it by overwriting it with an x and goes back to state `left`. If it doesn't find a b (it reaches the right #) then in concludes there were more a's than b's, and rejects the input.
- `search a`: analogous to `search b`

11. For each of the 10 description on the left, choose the best matching machine on the right.

d	corresponds with intuitive notion of "algorithm"	(a) deterministic FSA's
c	pushes and pops characters using a stack	(b) nondeterministic FSA's
d	reads and writes characters to and from an array	(c) nondeterministic PDA's
e	can recognize any language	(d) deterministic Turing machines
c	least powerful machines that can recognize language of all valid C programs	(e) none of the above
e	least powerful machines that can recognize language of all C programs that don't go into an infinite loop	
c	these nondeterministic machines recognize more languages than their deterministic counterparts	
e	recognizes fewer languages than nondeterministic FSA's	
d	named after Alan Turing	

The machines in the right column are listed in nondecreasing order of power, as in the Chomsky hierarchy. Note that deterministic and nondeterministic FSA's recognize exactly the same (regular) languages. Nondeterminism makes the PDA more powerful. (See Lecture Note 15.12). PDA's are FSA's with a stack; Turing machines are FSA's with an infinite tape or array.

No machine in the Chomsky hierarchy can recognize every language. The most famous of these examples is the Halting problem, i.e, recognize the language of all C programs that don't go into an infinite loop.

The language of all legal C programs (programs that compile without syntax errors) is described by a context-free grammar (as in one of the exercises and K+R Appendix A13). The Chomsky hierarchy says that n-PDA's are equivalent to context-free grammars, so PDA's are used by compilers to find syntax errors.

12. The following sequential circuit describes a master-slave flip flop. Fill in the timing diagram for the output ('OUTPUT'), given the input ('D') and clock signal ('CLOCK') shown below.

    Recall that a D-flip flop stores a bit. The bit can only change when its 'Cl' input is 1. In this case, it sets the bit to 1 if its 'D' input is 1, and to 0 if its 'D' input is 0.

    See Lecture Note 12.6. The important feature of a master-slave flip flop is that the output can only change just when the clock goes from on to off. The 'OUTPUT' is the value of 'D' at this time. It remains the same until just after the next clock tick.