Princeton University
COS 217: Introduction to Programming Systems
Spring 2008 Midterm Exam Answers

The exam was a 50-minute, open-book, open-notes exam.

Question 1 Part A

The decimal number 38 = 32 + 4 + 2, which is represented in binary as **00100110**.
The one's complement flips all of the bits, resulting in **11011001**.
The two's complement adds 1 to the one's complement, resulting in **11011010**.

Question 1 Part B

The output is: **18,2,1,0,2**

74/4 is 18 remainder 2, which rounds down to 18.
74%4 is the remainder of 2.
74&&4 is "true and true", which is true, i.e., 1.
74&4 is, in binary, 01001010 & 00000100, which is 00000000, which is 0.
74&3 is, in binary, 01001010 & 00000011, which is 00000010, which is 2.

Question 1 Part C

The "k >> 2" shifts out the last two bits, and the "<< 2" shifts back in two 0 bits in their place. As such, "((k >> 2) << 2)" replaces the last two bits of k's binary representation with 00. Subtracting that number from k produces the last two bits of k's binary representation, which is more concisely expressed as "**k & 3**". This is equivalent to "**k % 4**", though the "**k & 3**" is a more efficient way to achieve the same result.

Question 1 Part D

The code produces a string of the unsigned integer n's binary representation.

The variable numbits is the number of bits needed to represent an unsigned int (i.e., the number of bytes multiplied by 8 bits/byte). The malloc() allocates enough space to store one character for each bit, plus room for the '\0' to terminate the string. The for loop starts at the position for the last character and proceeds up to and including the 1st character, assigning the ith character to the ith bit in the unsigned int. The assignment to ret[i] extracts the last bit of the current value of n (which is shifting out the rightmost bit on each iteration of the for loop), and assigns a '0' if the value is 0, and a '1' if the value is 1. Then, the string is terminated with '\0' and the function returns the pointer to the string.

A common mistake was to assume the code reversed the order of the bits, because the loop counts backwards. The "counting backwards" is necessary because the "n & 1" extracts the last bit, rather than the first, and this bit should appear at the end of the string.

Question 2 Part A

The variable retbuf is a local variable, which is stored (temporarily) on the stack. As such, the "return" statement returns a pointer to memory that is no longer allocated after the function ends. In addition, the definition of retbuf[5] does not necessarily add enough space to store the string, depending on the size of the integer.

Question 2 Part B

```
char *itoa(int n) {
   int size = 0;
   int temp = n;
   char *retbuf;

   /* Count number of decimal digits in n */
```

```
   while (temp /= 10)
      size++;
   size++;

   /* If n is negative, add room for the "-" sign */
   if (n < 0)
      size++;

   retbuf = (char *) malloc(size + 1);
   assert(retbuf != NULL);
   sprintf(retbuf, "%d", n);
   return retbuf;
}
```

A common mistake was to mishandle the case where n is 0, which requires 1 character (rather than 0 characters).

Another common mistake was to assume that n is an unsigned int, and not allocated space for the minus sign.

Another common mistake was to omit the "assert(retbuf)" after the call to malloc, though no points were taken off for this.

Another mistake a few students made was to use "sizeof(n)" to compute the length; this is incorrect because it returns the number of bytes in a "int", not the number of digits in the decimal representation of n.

Some students extracted each of the decimal digits of n, using code similar to question 1b (though modified to manipulate n in base 10 rather than base 2). This is perfectly valid (and, as such, no points were taken off), though using "sprintf" is simpler.

An interesting, and clever, answer was to create a large array of characters as a local variable, use sprintf to place the string representation of n in the array, use strlen() to compute the length of the string, use malloc() to allocate the appropriate amount of space to retbuff, and then copy the string from the local variable to retbuf.

Question 3 Part A

(1) Prevent client code from accessing or modifying the data structure.
(2) Allow implementation to change without requiring client code to be recompiled.

Done in C by placing the structure type definition in the ADT's implementation (.c file) rather than in the ADT's interface (.h file); the interface contains only an opaque pointer.

Question 3 Part B

argv[0][2] is the 2nd character of the 0th argument ("a.out"), i.e., **'o'**.
argv[2][0] is the 0th character of the 2nd argument ("rules"), i.e., **'r'**

A common mistake was forgetting that argv[0] is the name of the program ("a.out" in this case), not the first command-line argument ("cs217").

Question 3 Part C

Because the size of the corresponding data is unknown.

Question 3 Part C

The **five** bytes of "jrex" (four letters, plus the '\0') are stored in the **rodata section** (in our system), and **the text section** in some other systems. (Either answer was acceptable.)

Question 3 Part D

The variable myname (a pointer) is stored in the **data section**, for initialized global variables. Initially the variable is equal to the address where the constant "jrex" is stored in the text/rodata section. Later, the "myname = yourname" changes the variable myname to the address where the constant "dondero" is stored. Since changing the value of (non-constant) variables is allowed, the "myname = yourname" line is valid.

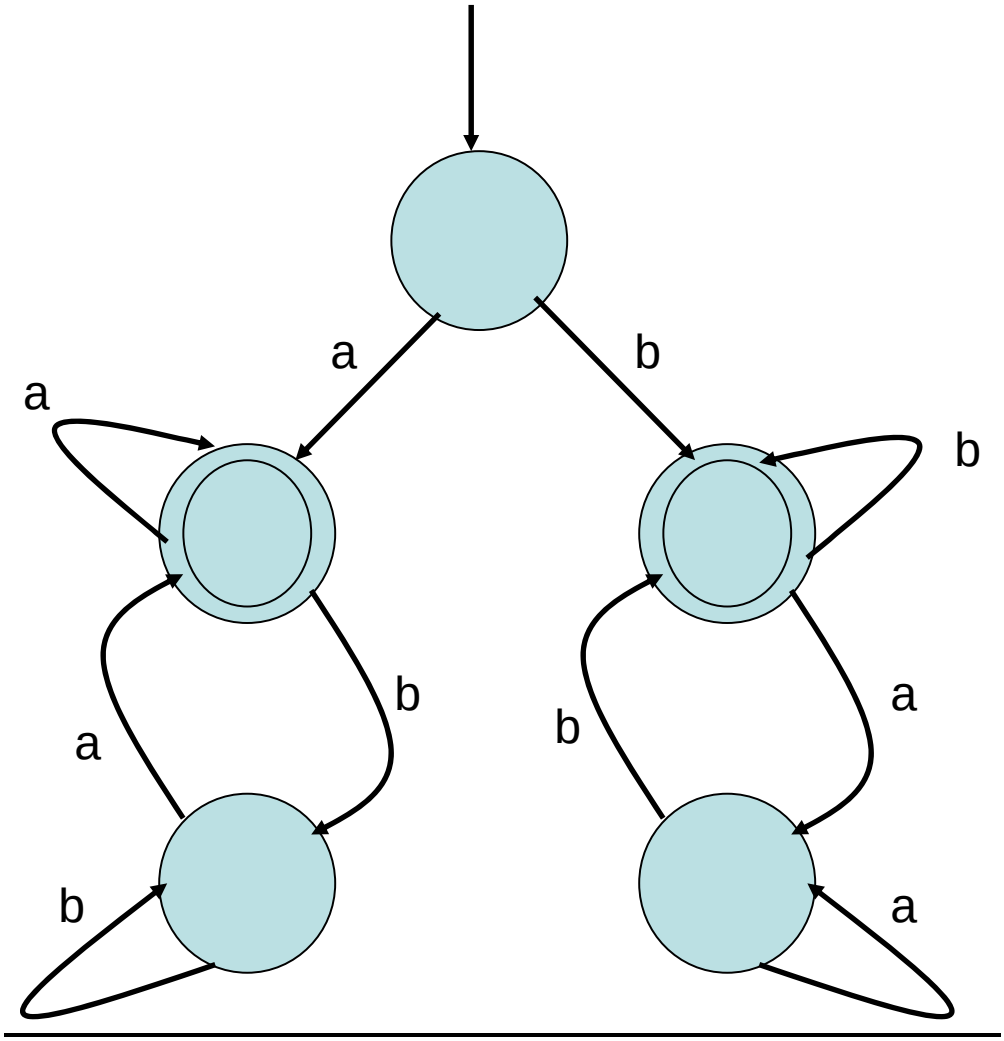
Question 3 Part E

Because the "getchar() != EOF" has precedence over the "c =", the variable c is assigned the boolean result of comparing the output of getchar() to EOF. As such, this code prints the character associated with ASCII code 1 until the EOF is reached.

A common error was not realizing that the "!=" operator has higher precedence than the "=" operator. Thus a common incorrect answer was "The code copies all characters from stdin to stdout."

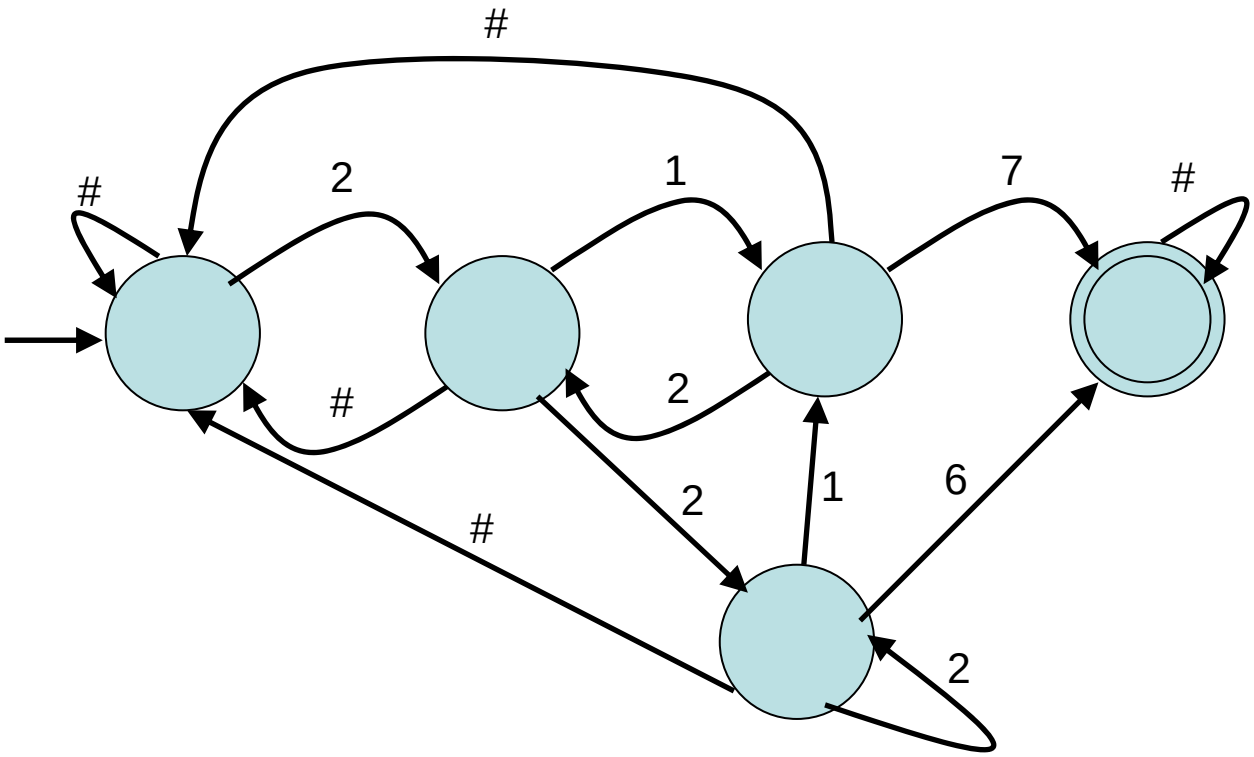
Question 4 Part A

For problem 4a, the most common error was not handling the strings 'a' and 'b' correctly. These strings begin and end with the same character, and thus should be accepted. Some students also forgot one or more of the self arcs. A couple students tried to unify the accepting states on the A and B paths. These approaches ended up accepting strings that should be rejected like 'ababb'.



Question 4 Part B

Overall people did very well on this problem. The common errors were forgetting the arc from C to B or B to A or the self arc on C.



Question 5 Part A

No, the assert() statements are still necessary, since the client code could have a memory leak that exhausts the available memory, or a bug that corrupts the heap, causing malloc() to fail to allocate space in the functions implementing the ADT.

Question 5 Part B

The malloc() call is allocating space for the list structure, which has the size of "the thing pointed to by the pointer newnode" (i.e., "struct list"). A malloc() of "sizeof(newnode)" would only allocate space of the size of a pointer.

Question 5 Part C

```
if (queue->head == NULL)
   queue->head = newnode;
else {
   struct list *walk;
   for (walk = queue->head; walk->next; walk=walk->next)
      ;
   walk->next = newnode;
}
```

If the queue is empty, add the new node at the head. Otherwise, walk through the list until reaching the last node (which that has a "next" pointer of NULL), and add the node to the end.

A common mistake was to add the new node at the **head** of the list even when the list is non-empty. Doing so would be reasonable if and only if the question also indicated that Queue_remove() will be redesigned to remove a node from the **tail** of the list.

Question 5 Part D

```
void Queue_add(Queue_T queue, void *item, int (*compare)(void *, void *)) {
   struct list *newnode ;

   assert(queue != NULL);
   newnode = (struct list*) malloc(sizeof(*newnode));
   assert(newnode != NULL);
   newnode->item = item;
   newnode->next = NULL;

   if (queue->head == NULL)
      queue->head = newnode;
   else if ((*compare)(queue->head->item, item) > 0) {
      /* Smallest element is added to beginning of list */
      newnode->next = queue->head;
      queue->head = newnode;
   }
   else {
      /* Add element after "walk" */
      struct list *walk;
      for (walk = queue->head; walk->next; walk=walk->next) {
         if ((*compare)(walk->next->item, item) > 0) {
             newnode->next = walk->next;
             break;
         }
      }
      walk->next = newnode;
   }
}
```

There were a few common mistakes:

- Failure to realize that a "compare" callback function is required.
- Failure to handle the empty list.
- Failure to handle the situation where the new node should appear at the beginning of the list.
- Splicing the new node into the list one spot beyond the proper place.

Question 6

The program produces its own source code as output. The printf() in the main function prints the string "char*s=%c%s%c;main(){printf(s,34,s,34);}", filling in (i) the double-quote character (ASCII code 34) for the first %c, (ii) the string "char*s=%c%s%c;main(){printf(s,34,s,34);}" for the %s, and (iii) another double-quote character (ASCII code 34) for the last %c.