1. **q1h**

   My answer: true

   New answer: false

   Explaination: A literal value is evaluated as it's declaration. A String literal 'time' is evaluated as 'time' and an int literal 1 is evaluated as 1. Thus the program has access to the values of literal types. They are allocated statically by the compiler and usually stored in read only memory. They do not need to be allocated.

2. **q1j**

   My answer: true

   New answer: false

   Arguments are not passed by pointer in C. They are passed by value. This is relatively standard for programing langauges.

3. **q2**

   My original answer: I thought that there was a problem with memory allocation, but I had a very wrong idea about where this memory allocation problem happened, I was debating if the issue was with a double free at the end, or a problem with using arrarr that wasn't the real issue. Either way, I was wrong. I thought that this line:

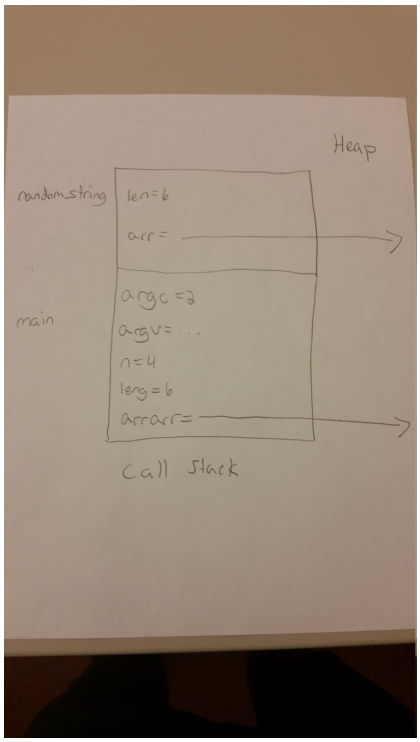
   ```
   char **arrarr = malloc(sizeof(char*) *n)
   ```

   was enough to allocate an array of C-Strings. Now I understand that this line only allocates memory for the pointers to C-Strings.

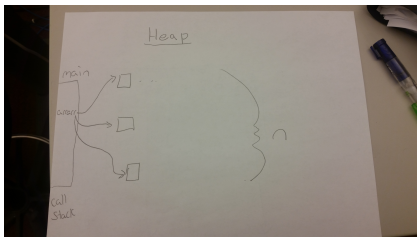
   New answer: The first attempt to write to arrarr is inside of the randomString function, by memset:

   ```
   memset(arr, '\0', len+1);
   ```

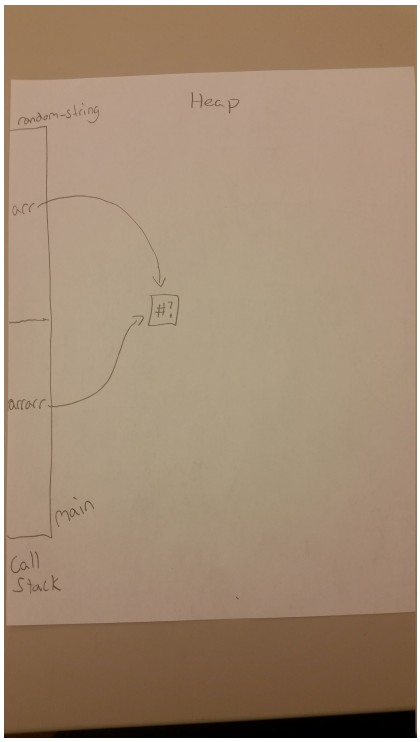
   Before this function is called, the call stack looks as follows:



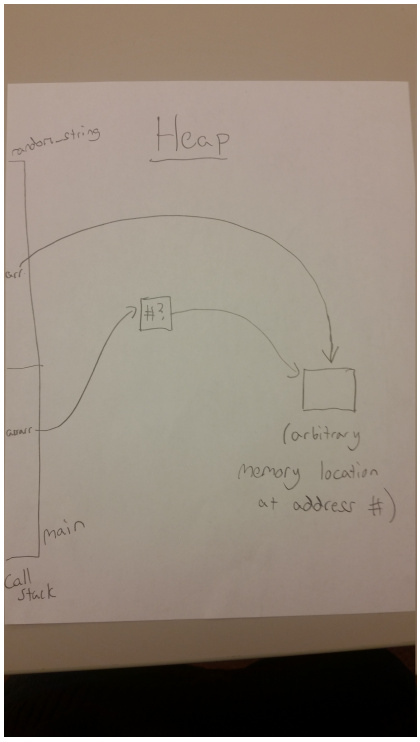
arrarr has been alocated and with it, the memory for C-string pointers on the heap. Visually, this can be displayed like this:



As we can see the pointers have been allocated, yet the memory for the values of the strings (the set of chars in the C-string) has not been allocated yet. This leads to the following behavior when randomstring is called:



Frist we see arr(passed to randomString as a pointer to arrarr) try to write to the memory pointed to by arrarr. But all it processes is a pointer. Memory for the values of arrarr have not been allocated. If a value is stored at the location pointed to, it is interpreted as another pointer, leading to the situation below:



randomString(and memset in particular) tries to write to an arbitrary memory location, at the address of the value pointed to by arrarr. If this is memory which is not allowed to be overwritten, a segmentation fault occurs and the program crashes.

Hence the patch is in the for loop which calls the randomstring function. It is placed directly before the function is called:

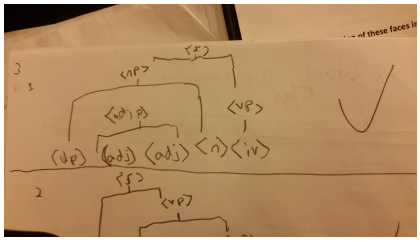
```
arrarr[i] =malloc(sizeof(char) *(len+1));
mcheck(arrarr[i]);

randomString(arrarr[i]);
```

The first line of the patch allocates memory for the char values of 1 C-string of length len (it is written as len+1 to allow space for the terminating null terminator in the C-string). These values will be written to by the randomString call. Then the for loop will iterate or terminate based on the condition (i¡n). The program will continue to allocate memory for C-strings, and write to C-strings with randomString, until this point. The final line of the patch is a basic check to see if the memory allocation succeeded.

4. **q3a**

   my answer: The sentance is valid. (photo attached)

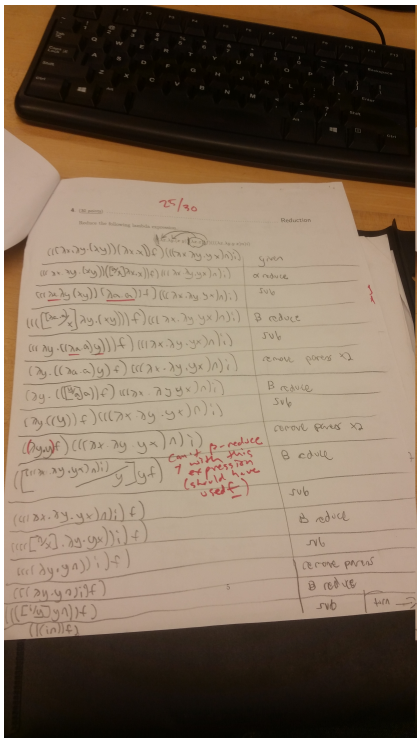


   new answer: The sentance is invalid.

   Explaination: The word ‘The‘ is not present in the grammar for the language. It should have been ‘the‘. Then the parse tree created would work out. I did not realize that the T was capitalized.

5. **q4**

   my answer: inf (photo attached)



I removed parenthesis that were essential in determining what could be evaluated, and thus evaluated the rest of the expression slightly wrong, arriving at the answer ‘inf‘.

new answer:

$$
\begin{array}{rcl}
(((\lambda x.\lambda y.(xy))(\lambda x.x))f)(((\lambda x.\lambda y.yx)n)i) & | & given \\
(((\lambda x.\lambda y.(xy))([a/x]\lambda x.x))f)(((\lambda x.\lambda y.yx)n)i) & | & \alpha\ reduce \\
(((\lambda x.\lambda y.(xy))(\lambda a.a))f)(((\lambda x.\lambda y.yx)n)i) & | & sub \\
((([(\lambda a.a)/x]\lambda y.(xy)))f)(((\lambda x.\lambda y.yx)n)i) & | & \beta\ reduce \\
(((\lambda y.((\lambda a.a)y)))f)(((\lambda x.\lambda y.yx)n)i) & | & sub \\
(((\lambda y.(([y/a]a))))f)(((\lambda x.\lambda y.yx)n)i) & | & \beta\ reduce \\
(((\lambda y.((y))))f)(((\lambda x.\lambda y.yx)n)i) & | & sub \\
((\lambda y.y)f)(((\lambda x.\lambda y.yx)n)i) & | & remove\ parens(*3) \\
(([f/y]y))(((\lambda x.\lambda y.yx)n)i) & | & \beta\ reduce \\
((f))(((\lambda x.\lambda y.yx)n)i) & | & sub \\
f(((\lambda x.\lambda y.yx)n)i) & | & remove\ parens(*2) \\
f((([n/x].\lambda y.yx))i) & | & \beta\ reduce \\
f(((\lambda y.yn))i) & | & sub \\
f((([i/y].yn))) & | & \beta\ reduce \\
f(((in))) & | & sub \\
fin & | & remove\ parens(*3)
\end{array}
$$