Max Stein
CSCI 334

Midterm Exam Corrections:

1. True or false

(d) The expression (λ*x.y*)((λ*x.xx*)(λ*x.xx*)) is not confluent

I answered True for this question because I did not remember the definition of confluent, so I thought confluent meant that the expression would fully reduce to a form that cannot be reduced further. I was confused by the fact that an beta reduction on the second part of this expression reduces to the same expression.

The correct answer is False because all expressions in the lambda calculus are confluent. Confluent really just means that the order of evaluation of an expression does not affect the final reduced form. Since the order of valid beta and alpha reductions does not affect the final reduced form of a lambda expression, all lambda calculus expressions are confluent, including this one.

(j) Because C has pointers, function arguments are passed by reference.

I answered True because I was confused by the phrase “passed by reference.” My thought process was that passed by reference sounded like the opposite of dealing with the variable that was passed directly, so since I knew that function arguments were not dealing with the variable directly, this statement sounded pretty correct. In fact, my thought process turned out to be the opposite of the truth. If function arguments are passed by reference, it means that the function has a pointer to the original variable that was passed, so it can change it directly. The opposite of dealing with the original variable directly is making a copy of it, which is really what functions do (pass by value = copy, pass by reference = original argument).

The correct answer is False, then, because function arguments are not passed by reference, they are passed by making a copy of the variable that is passed and then doing things with the copy variable in place of the original one.

2. Troubleshooting?

My answer to this question involved allocating memory within the `random_string()` function. I wanted to take the function:

```
void random_string(char *arr, unsigned len) {
  const char *letters = “abcdefghijklmnopqrstuvwxyz”;
  memset(arr, ‘\0’, len + 1);
  for(int i = 0; i < len; i++) {
    Arr[i] = letters[rand() % 52];
  }
}
```

And change it to the following:

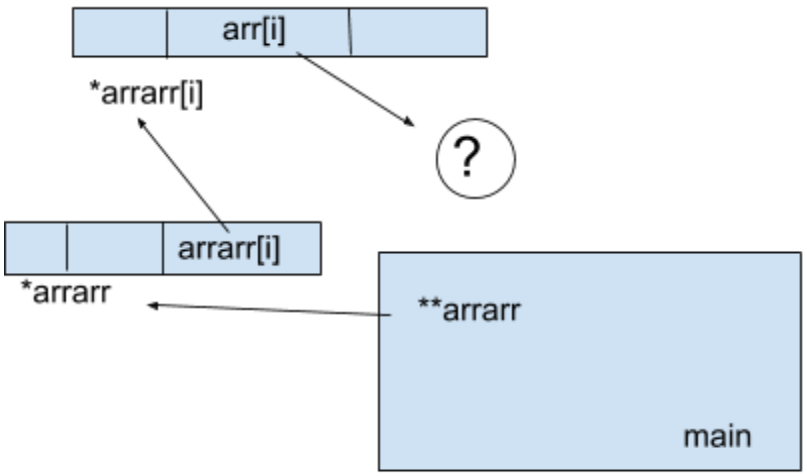
```
char* random_string(unsigned len) {
    const char *letters = "abcdefghijklmnopqrstuvwxyz...";
    char *str = malloc(sizeof(char *));
    for(int i = 0; i < len; i++) {
        strncpy(str[i], letters[rand() % 52], sizeof(char));
    }
    return str;
}
```

And then I thought I could put the `char*` returned by this method into the `char** ararr` so that each `char* arr` is allocated by the `random_string` method.

I knew that there was some sort of problem with the allocation of "`arr`" so I tried to allocate it in the `random_string` method, copy letters into it, and then return a pointer to the allocated memory. When I run this code (with a few syntax fixes, mainly saying `&str[i]` in strncpy since it needs to take in pointers), it puts out a nonsense string about half the time but the other half of the time it says "Aborted (Core Dumped)." This was actually caused by the fact that I said `malloc(sizeof(char *))` instead of `malloc(sizeof(char) * n)` which causes my allocation to sometimes be smaller than the memory that needs to be used.

Regardless, allocating within the `random_string` function is not the right solution because the `ararr` and `arr` arrays both are created and used in `main`, so they both should be allocated in main before they are referenced.

Even though it seems like I was at least attempting to solve the right problem, my call stack diagram shows that my real understanding of the problem was even further behind. The diagram I drew of the problem looked something like this:

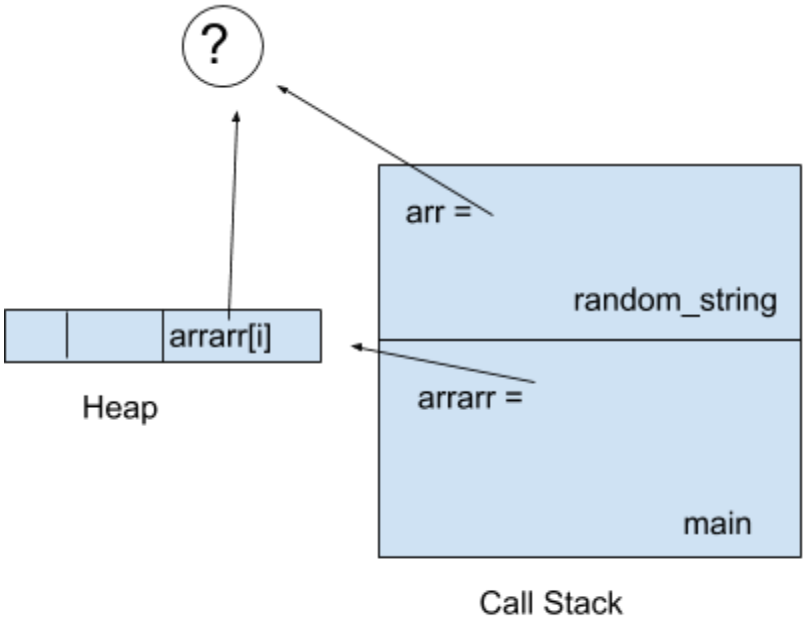


And it said “Once random_string is done, *letters is gone and arrarr[i] is full of pointers that point to nothing.” This shows that I really thought the problem was not with allocating `arr` itself, but rather with the letters within `arr`. Letters is a `const char *`, so there’s no reason to think the allocation error would be with this. I somehow thought that `arrarr[i]` was pointing to something that was allocated, but in reality the string at `arr[i]` never was allocated. I also thought the problem arose after `random_string`, but really it happens `arr` is attempted to be dereferenced to pass to that function to run.

Therefore the real fix has to come from allocating `arrarr[i]` within the main method.

Correct Answer:

If I were to draw the correct diagram, showing that `arrarr[i]` was never allocated so the `arr` in `random_string` would minimally look something like:



It is important to show that `arrarr[i]` was never allocated so points to a random place in the heap and that `random_string` tries to dereference.

The fix, allocating `arrarr[i],` looks like this:

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

In the `main` function, right before the line that calls `random_string,` because we know the memory needs to be allocated before `random_string` when the function tries to dereferences `arrarr[i]`.

Max Stein
CSCI 334

However, the comment on my exam said “No amount of allocating in this function [random_string] will help,” but I would like to challenge that. I was actually able to get the program working by just fixing a few small errors in my exam’s “fix” (length of memory allocated was too short and I needed to pass addresses to the strncpy function). On the next page is a functional version (likely still with mistakes) of my solution (with malloc running in the random_string function). The problem was not that this approach was impossible, but that it was just really *bad.*

This is a way sloppier fix than the solution we were given, but it made me realize that my first attempt was actually closer to being functional than I expected at first glance. In terms of my mindset during the test itself, it seems like I thought of the first solution that came to mind and put in a lot of effort to make my vague vision come to life, instead of spending time to think of the simplest fix. Instead of thinking “what is the best way to allocate memory here,” I thought “how can I make my idea for allocating memory work.” In all of programming, this is a mistake. This was partly a mistake caused by lack of understanding of memory management, but I think was more of a mistake caused by a lack of preemptive thoughtfulness when attacking a bug fix situation.

Max Stein
CSCI 334

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#define MAXLEN 10

void usage() {
    printf("Usage: n_random_words <n>\n");
    printf("\tGenerates a sentence from n random strings, where n > 0.\n");
    exit(1);
}

void mcheck(void *mem) {
    if (!mem) {
        printf("out of memory\n");
        exit(1);
    }
}

char* random_string(unsigned len) {
    const char *letters = "abcdefghijklmnopqrstuvwxyz";
    char *str = malloc(sizeof(char) * len);
    memset(str, '\0', len + 1);
    for(int i = 0; i < len; i++) {
        strncpy(str + i*sizeof(char), letters + (rand() % 26)*sizeof(char), sizeof(char));
    }
    return str;
}

void cleanup(char **arr, int n) {
    for(int i = 0; i < n; i++) free(arr[i]);
    free(arr);
}


int main()
{
    if(argc != 2) usage();
    int n = atoi(argv[1]);
    if (n <= 0) usage();
    srand(time(NULL));
    char **arrarr = malloc(sizeof(char *) * n);
    mcheck(arrarr);

    for(int i = 0; i < n; i++) {
        unsigned len = 1 + rand() % (MAXLEN - 1);
        arrarr[i] = random_string(len);
    }

    for(int i = 0; i < n-1; i++) {
        printf("%s ", arrarr[i]);
    }
    printf("%s\n", arrarr[n-1]);

    cleanup(arrarr, n);
    return 0;
}
```