Lab 4: Tracery Recursion in C with Linked Lists

For this lab we will be building on our previous lab — at the end of the previous lab you should have had:

```
#include <stdio.h>
#include <stdlib.h>

char* make_string_from(char* from, int count){
  char* to = calloc(count+1,sizeof(char));
  for (int ii = 0; ii < count; ii++){
    to[ii] = from[ii];
  }
  return to;
}

int main(int argc, char** argv){
  for (int ii = 0; ii < argc; ii++){
    printf("%s\n",argv[ii]);
  }
  char current;
  current = getchar();
  char* rule = NULL;
  char* expansion = NULL;
  char buffer[1000];
  int buffer_index = 0;
  while (current != EOF){

    if (current == ':'){
            if (rule != NULL){
                  free(rule);
                  rule = NULL;
            }
            rule = make_string_from(buffer,buffer_index);
            printf("The rule is '%s'\n",rule);

            buffer_index = 0;
    }
```

```
    else if (current == ',' || current == '\n' || current=='\r'){
            if (expansion != NULL){
                  free(expansion);
                  expansion = NULL;
            }
            expansion  = make_string_from(buffer,buffer_index);
            printf("An expansion for rule '%s' is '%s'\n",rule,expansion);
            buffer_index = 0;

    }
    else {
      buffer[buffer_index] = current;
      buffer_index++;
      }

    current = getchar();
  }

  if (rule != NULL){
    free(rule);
    rule = NULL;
  }
  if (expansion != NULL){
    free(expansion);
    expansion = NULL;
  }
}
```

This code read in the grammar file and printed out the rules. We will now be re-implementing Lab 1, except we will be doing it using Linked Lists, in C, from scratch.

To do this, we will be using the part of C that is closest to the classes that we are used to in Java, the `struct`. A `struct` in C is a bundling of variables into one cohesive unit, similar to a class in Java, except unlike a class, the structure doesn't have any methods. You might have noticed that the basic data types in C don't come with a lot of frills (there is no built in way to figure out the length of a string or array, for example), so structures can be utilized to add these on, to get closer to what we became accustomed to in Java.

For instance:

```
struct string {
  char* characters;
  int length;
};
```

Will declare a new structure that can then be utilized as follows:

```
struct string myString;
myString.characters = "abc";
myString.length = 3;
```

If we wanted to declare it dynamically we would do:

```
struct string* myStringPtr  = calloc(1,sizeof(struct string));
myStringPtr->characters = "abc";
myStringPtr->length = 3;
```

Note the difference in how we access the member variables when we are using a pointer to the `struct` (`->`) vs. (`.`).

While we can't declare member functions of the class like we can in Java, we can define functions that will operate in much the same way. So instead of:

```
class MyLinkedList{
  ...
  public Object get(int index);
  ...
}
```

we have

```
struct LinkedList {
      ...
};
...
get(struct LinkedList* list, int index);
...
```

If we don't want to write `struct` every time we want to use the `struct`, we can utilize `typedef`,

which allows us to specify a way to refer to the struct without the use of the `struct` keyword.

```
typedef struct list_struct {
      ...
} LinkedList;
...
get(LinkedList* list, int index);
...
```

In Java, it is enforced that all classes be in their own files, but C has no enforcement like this, and theoretically everything could exist in the same file (but this is obviously very bad practice for all but the simplest projects). However, if implement our linked list in the main file in C, there would be no way for any other file to access it. In C, if we want other files to be able to access the code, we have to put them in a header file. You've encountered standard library header files already:

```
#include <stdlib.h>
#include <stdio.h>
```

But now we are going to be defining our own. You have been supplied with a header file for this assignment, list.h, which defines a subset of the aspects that we would expect in a linked list:

```
Node* make_node(void* data, void* next); //constructor for Node
List* make_list(); //constructor for List

void free_node(Node* node); //destructor for Node
void free_list(List* list); //destructor for List

void add(List* list, int index, void* data); //add method for List
void* get(List* list, int index);  //get method for List
```

The first thing we are going to want to do is to implement the structs for Node and List.

TODO 1

- In list.h, you will find the stubs for the Node and List structs, we are going to put the correct member variables in them.
- For Node we need:
    - A `data` member like the `Object` we used in our Java implementation. In C, this is a `void*`. A pointer of type `void *` means that it can point to *anything*. C doesn't do any type checking for a `void *` other than to make sure you're assigning it some kind of pointer value.
    - A `next` member that points to the next element in the list. In C, you can't have a struct be self-referential with the typedef'd name (i.e. we can't say `Node* next`), so you have to use `struct node_type*`.
- For List we need:
    - The `size` of the list
    - The `head` of the list

With our header defined, it is time to implement the functions defined in the header. While it is possible to implement them in the header, this is generally a bad practice in C. So we will be implementing them in their own file — generally this should be parallel to the filename of the corresponding header, so in this case in `list.c` .

TODO 2

- In list.c implement the functions defined in list.h
- Instead of throwing exceptions when trying to do something out of the bounds of the list, we will be using `asserts`. Where in Java we would write something like:

```
if (index > size || index < 0){
  throw new ListIndexOutOfBoundsException("Index was out of bounds");
}
```

In C we would write:

```
assert( (index > size || index < 0) && "Index was out of bounds");
```

(The `&& "Index was out of bounds"` code is a bit of a cheat in C — in C, 0 is false and everything else is true, so a string is always true. This means that we can add on a message to our assert by &&ing with the actual error we wish to check for). To use asserts, we will need to include `<assert.h>`

- Implement the `make_node` function that takes in a `void* data` and `Node* next`, allocates the space for a `Node`, sets the members of the new `Node*`, and returns a pointer to the new `Node`.
- Implement the `make_list` function that allocates the space for a `List`, sets the members of the new `List` (`size` should be 0, and `head` should be `NULL`), and returns the pointer to the new `List`.
- Implement the destructor method for a `Node` — `free_node` which frees the `data` (if it isn't NULL) contained by the `Node` and then frees the `Node` itself.
- Implement the destructor method for the `List` — `free_list` which iterates through each `Node*` in the list and frees it, and finally free the `List` itself.
- Implement `add` and `get` which have the assertions we discussed above
- Implement a new method, `set`, which iterates to the correct `Node` and then replaces the data of that `Node` with the specified data.

With our list implemented — we can try it out in tracery_recursion.c — uncomment the `List` test code and compile tracery_recursion.c and list.c:

```
$ gcc --std=c99 tracery_recursion.c list.c helpers.c -o tracery_recursion
```

and run the test

```
$ ./tracery_recursion abc def xyz
```

Which should add the arguments supplied to the program, print out the size of the list, print out the elements in reverse order, print out the elements in the correct order, and then free the list correctly.

```
The list size is 4
./a.out
abc
def
xyz
xyz
def
abc
./a.out
```

Once we have a working list, we can move on to implementing Tracery.

TODO 3

- In rule.h — you will see the stub for the `Rule` struct — fill it out
    - A Rule will have a `key`, a string, (e.g. origin, color, name, animal, emotion, etc. in the grammar-story.txt)
    - A Rule will have a `List*` of expansions
- In rule.c —
    - Implement the constructor for a `Rule`, `make_rule` which will take in a `key`, allocate the space for a `Rule`, put the `key` in it, initialize the `expansions` to be an empty list, and return the pointer to the `Rule`.
    - Implement the destructor for a `Rule*`, `free_rule`, which will free the key and the expansions

We now have the building blocks to fill up the grammar — we will be using code that is very similar to what we did previously, except instead of just building up strings, we will be constructing a grammar list as we go along. The file I/O portion is handled for you, but you will need to construct the grammar and rules yourself.

TODO 4

- In rule.c implement `read_grammar`—
    - We first need to construct a `List` for the grammar that we will fill up — TODO 4A
    - When we have read in a `Rule` key, we will construct a `Rule` with that key, and add it to the end of our grammar list. — TODO 4B
    - When we read in an expansion, we will add it to the end of the most recent rule (the last `Rule` in the grammar).— TODO 4C
    - We will return the grammar that we just filled up — TODO 4D

To test that your grammar is correctly loaded, you can call `print_grammar(List* grammar)` from the included helper code to print the grammar (see the commented code in `tracery_recursion.c`). When this has been implemented, compile with the command:

```
$ gcc --std=c99 tracery_recursion.c list.c helpers.c rule.c -o
tracery_recursion
```

And your output should look like:

```
A potential expansion of rule 'animal' is 'cat'
A potential expansion of rule 'animal' is 'emu'
A potential expansion of rule 'animal' is 'okapi'
```

```
A potential expansion of rule 'animal' is 'happy'
A potential expansion of rule 'animal' is 'sad'
A potential expansion of rule 'animal' is 'elated'
A potential expansion of rule 'animal' is 'curious'
A potential expansion of rule 'animal' is 'sleepy'
A potential expansion of rule 'animal' is 'red'
A potential expansion of rule 'animal' is 'green'
A potential expansion of rule 'animal' is 'blue'
A potential expansion of rule 'animal' is 'emily'
A potential expansion of rule 'animal' is 'luis'
A potential expansion of rule 'animal' is 'otavio'
A potential expansion of rule 'animal' is 'anna'
A potential expansion of rule 'animal' is 'charlie'
A potential expansion of rule 'animal' is '#name# the #adjective# #animal'
A potential expansion of rule 'animal' is 'school'
A potential expansion of rule 'animal' is 'the beach'
A potential expansion of rule 'animal' is 'the zoo'
A potential expansion of rule 'animal' is 'Burning Man'
A potential expansion of rule 'animal' is '#color#'
A potential expansion of rule 'animal' is '#emotion#'
A potential expansion of rule 'animal' is 'once #character# and #character#
went to #place#'
```

Now that we have read in the grammar, we can do the recursion to expand the grammar.

(EXTRA CREDIT) TODO

- In tracery_recursion.c —
    - Uncomment the code in TODO 5 to read in the grammar and call expand
- In rule.c —
    - Implement the `expand` method
        - First, `split` the text passed in to expand using the `split` function supplied for you in helpers.h and helpers.c
            - `split` takes in `char*` string and `char*` set of delimiters and returns a `List*` where each element is a string found in between those delimiters
        - Next construct a new List that will contain the expanded text pieces
        - Next iterate over the elements of the split text
            - If the index of an element is even, then it is a terminal symbol, and it can be added to the expanded List

        - NOTE: Lists take complete ownership of the data added to them, so you need to make sure that anything added to the List can be freed by the List — to do this use the supplied `copy_string(char* from)` method to copy the text
    - If the index of an element is odd, then it is a non-terminal symbol
        - Iterate through the grammar until you find the `Rule*` whose key is equal to the element — use `strcmp(char*, char*)` to test for equality (`strcmp(A,B) == 0`)
        - Once you have the rule, use the supplied `get_random(List*)` which takes a List and returns a random element to pick an expansion at random
        - Do the recursive call with the expansion and add the returned value to the expanded List
- Once all elements have been iterated over —
    - Use the supplied `join(List*)` which joins a `List*` of `char*`s to join them into a new string
    - Free the List* made by calling split
    - Free the List* that contained the expanded text
    - Return the joined string

Expected Output

You should compile with the command

```
$ gcc tracery_recursion.c  list.c rule.c helpers.c --std=c99 -o
tracery_recursion
```

And when run with the command

```
$./tracery_recursion
```

Multiple times in a row, you should receive different output

```
$./tracery_recursion
once luis the blue emu and charlie the sad okapi went to the zoo
$./tracery_recursion
once charlie the happy okapi and otavio the happy cat went to the zoo
$./tracery_recursion
once anna the happy cat and otavio the sleepy emu went to school
```

And if you run with the command

`$./tracery_recursion <int>` then you should receive the same output

```
$./tracery_recursion 1
once otavio the red emu and otavio the green cat went to school
$./tracery_recursion 1
once otavio the red emu and otavio the green cat went to school
$./tracery_recursion 1
once otavio the red emu and otavio the green cat went to school
```

Grading

TODO 1: 15 points
TODO 2: 40 points
TODO 3: 30 points
TODO 4: (EXTRA CREDIT) 30 points
Style: 15 points

As a note, it is far more important to turn in code that compiles and is missing a TODO or only partially does a TODO, than it is to attempt a TODO and make your code uncompilable. If you code doesn't compile, you will receive 40 points off, so make sure your code compiles before you submit it (even if you are missing a TODO).

Turning the code in

- Create a directory with the following name: `<student ID>_lab4` where you replace `<student ID>` with your actual student ID. For example, if your student ID is 1234567, then the directory name is `1234567_lab4`.
- Put a copy of your edited `tracery_recursion.c, list.h, list.c, rule.h, and rule.c` files in the directory.
- Compress the folder using zip. Zip is a compression utility available on mac, linux and windows that can compress a directory into a single file. This should result in a file named `<student ID>_lab4.zip` (with `<student ID>` replaced with your real ID of course).
- Upload the zip file through the page for Lab 4 in canvas.