Preface

This lab uses a modified version of the language *Cava*. As the name implies, it is a combination of C and Java, in theory offering the low-level memory management of C, as well as high level concepts such as objects and interfaces. *Cava* compiles down into C, generating a series of header and source files to be further modified or compiled down to executable code.

Interfaces

Interfaces compile down to header files, generating a typedef'd struct that contains a void pointer called "vars", along with function pointers matching the interface. Additionally, a "del" method is also added to the struct.

For example, consider the interface Set:

```
public interface Set<T>{
    void add(T x);
    T remove(T x);
    T removeAny();
    int contains(T x);
    int size();
}
```

Run through the *Cava* compiler it will generate the following header:

```
typedef struct Set{
    void*  vars;
    void   (*add)        (Set*, void*);
    void*  (*remove)     (Set*, void*);
    void*  (*removeAny)  (Set*);
    int    (*contains)   (Set*, void*);
    int    (*size)       (Set*);
    void   (*del)        (Set*);

}Set;
```

There are several interesting details here. Firstly, is the void pointer “Set”. This will be used by implementing classes to store their instance variables. Next is the first formal parameter is always a pointer to a struct of the same type. This is effectively passing in “this” to make the methods instance methods. Finally, in the case of generics, they are converted to void pointers, which the compiler later casts to the corresponding type.

Classes

Classes compile down to a header file and a C file. The header contains typedef’d struct of all the instance variables, the generated struct(s) of any private nested classes, function declarations for methods inherited from the interface, and finally the constructor and destructor for the class. The C file will then contain the implementation of all its functions.

For example, considering the following class Set1:

```
public class Set1<T> implements Set<T>{
    class LLNode{
        T data;
        LLNode next;
    }

    private LLNode first;
    private int size;

    public Set1(){
        // Constructor code
    }

    public ~Set1(){
        // Destructor code
    }

    private static void myPrivateStaticMethod(){
        // Static code stuffs
    }

    private void myPrivateInstanceMethod(){
        // Instance code stuffs
    }

    public void add(T x){
        // Add code
    }

    public T remove(T x){
        // Remove code
    }

    /* RemoveAny, Contains, and Size removed for example */

}
```

The two files it will generate are the header, Set1.h:

```c
#include "Set.h"

typedef struct LLNode{
    void* data;
    LLNode next;
}LLNode;

typedef struct Set1{
    LLNode first;
    int size;
}Set1;

Set*  newSet1           ();
void  addSet1           (Set*, void*);
void* removeSet1        (Set*, void*);
void* removeAnySet1     (Set*);
int   containsSet1      (Set*, void*);
int   sizeSet1          (Set*);
void  delSet1           (Set*);
```

And Set1.c:

```c
#include "Set1.h"

Set* newSet1(){
    // Constructor Code
}

void delSet1(Set*){
    // Destructor code
}

void myPrivateStaticMethod(){
    // Static code stuffs
}

void myPrivateInstanceMethod(Set*){
    // Instance code stuffs
}

void addSet1(Set*, void* x){
    // Add code
}

void* removeSet1(Set*, void* x){
    // remove Code
}

/* Remaining methods hidden for example */
```

All Method code is left as is, unless involving a *Cava* object construction, destruction, or method call. The code transformation is described in the following section.

Method Calls

During *Cava* compilation, any method calls are transformed into equivalent C code simply by using the function pointer contained in the interface struct and copying the ordering of formal parameters.

For example, the following *Cava* code:

```
import Set;
import Set1;

#include <stdlib.h>

typedef struct myObj{
    int data;
}myObj;

int main(int argc, char *argv[]){
    Set mySet = new Set1<myObj>();

    myObj* obj = malloc(sizeof(myObj));
    obj->data = 4;
    mySet.add((void*)obj);

    obj = mySet.removeAny();

    free(obj);
    free(mySet);

    return 0;
}
```

Is compiled into the following C code:

```c
#include "Set.h"
#include "Set1.h"

#include <stdlib.h>

typedef struct myObj{
    int data;
}myObj;

int main(int argc, char *argv[]){
    Set* mySet = newSet1();

    myObj* obj = malloc(sizeof(myObj));
    obj->data = 4;
    mySet->add(mySet, (void*)obj);

    obj = (myObj*)mySet->removeAny(mySet);

    free(obj);
    mySet->del(mySet);

    return 0;
}
```

Simplifications

You probably don't want to make a compiler in a single week, so we'll save that for CSE 3341/5343. The headers have been precompiled and combined into "Lab4.h" for you that way there's less looking around. We've also remove support for generics, so you don't have to cast, and can just use all instances of T as simply ints.

As a hint, you don't need (and shouldn't, remember Kernel Purity Rule?) to call other instance methods while inside one. Keep in mind that you can declare private methods to help yourself out inside the "generated" C files. Reference things above when you're stuck and/or ask in the discord for clarification. Good luck!