CS24 Assignment 4

The files for this assignment are provided in the archive **`cs24hw4.tgz`** on the course website. The archive contains a top-level directory **`cs24hw4`**, in which all other files are contained.

For the questions you need to answer, and the programs you need to write, put them into this **`cs24hw4`** directory. Then, when you have finished the assignment, you can re-archive this directory into a tarball and submit the resulting file:

> *(From the directory containing **`cs24hw4`**; replace **`username`** with your username.)*
> **`tar -czvf cs24hw4-username.tgz cs24hw4`**

Then, submit the resulting file through the course website.

Problem 1: Implementing OOP in C *(25 points)*

(The files for this problem are in the **`cs24hw4/classes`** directory.)

As discussed in class, it is straightforward (if tedious) to map many object-oriented programming concepts straight into the C programming language. This problem will give you an opportunity to experiment with this yourself.

In this problem we will focus on a simple class hierarchy for representing shapes, implemented in C++. C++ is interesting because it allows you to explicitly specify which methods are virtual (dynamic dispatch via virtual function pointer tables) and which are nonvirtual (static dispatch).

You can also declare a method to be *pure virtual*, which means that the class is not providing an implementation; subclasses are expected to provide an implementation of the method themselves. This is simply C++'s mechanism for declaring a method to be abstract. A class containing pure-virtual methods cannot be instantiated; it is missing some of its functionality so it would make no sense to instantiate it.

Back when C++ was still called "C with Classes," programs were actually translated straight into C code and then compiled with the C compiler. The mechanisms for implementing objects are exactly the ones we have discussed in class.

In the **`shapes.h`** and **`shapes.c`** files you will find the C code that implements our shape class hierarchy. You can see how many declarations and functions are needed even for simple classes (and I haven't included destructors), so you can see how much the C++ syntax helps you. This is the kind of code that the "C with Classes" translator would generate, although **`shapes.h`** is much more well-commented so you can understand what is going on.

Your Tasks

1. You won't need to change anything in **`shapes.h`**, but in **`shapes.c`** you will need to fill in the functions that are marked "TODO". Virtually all of these functions are very short to implement.

   See the Tips and Guidelines section below for additional information.

There is also a test program provided, called **`shapeinfo.c`**. It creates a cone, a sphere, and a box, and then prints out some details of each shape using a generic function written against the Shape base-class. You also need to fill in the part marked "TODO" in this file.

Once you have completed these functions, you should be able to compile **`shapeinfo`** using the provided **`Makefile`**, and then you can give it a test.

2. In a file **`cs24hw4/classes/invoke.txt`**, answer the following question. *(3 points)*

   Assume that **`8(%ebp)`** contains the address of an object that is a properly initialized **`Shape`** subclass (e.g. a sphere), and we want to invoke the virtual function **`getVolume()`** on this object. Write the series of IA32 assembly instructions necessary to do this. Make sure to explain the sequence of operations.

   Ignore concerns about callee/caller-save registers, and also ignore how floats are returned since we haven't talked about this in class. Just focus on how the method is invoked.

   If you ever need to debug C++ code at the assembly level, you will frequently see this sequence of instructions.

Additional Tips and Guidelines

In C++, a subclass constructor is not allowed to directly initialize the superclass data-members. Therefore, the subclass constructor always calls the superclass constructor as the very first step, so that the superclass can initialize its data-members before the subclass constructor does anything else. When you implement the subclass constructors, you should do the same thing. (It will keep your constructors nice and short, too!)

When subclasses pass pointers to their information to the superclass, you might notice that the pointer-types don't match up, and in fact C will complain about this. **You need to perform an explicit cast operation to get the compiler to accept this.** See the **`main()`** function in **`shapeinfo.c`** for an example of this. As mentioned in class, we can cast from the "subclass" type to the "superclass" type because the structs have the same preamble.

Make sure to pay attention to which methods are nonvirtual and which methods are virtual. The calling mechanism is different for each of these kinds of methods. You should not have to change **`shapes.h`** at all.

You should avoid repetition as much as possible! For example, all of the subclass constructors take initial values for their members, and they also have a "**`setXXXX()`**" mutator for setting the values of their members. The constructors should use these mutators. Again, this will keep your code nice and short.

Don't forget to use assertions in your code, where the comments state that inputs are checked or constrained in some way. (Another reason to have constructors reuse mutators; that way your assertions are all in one place and your code stays short.)

Definitely check the answers your program generates! If you get wrong answers, you may have either a math bug or an OOP bug in your code. In C, you might want to write your numbers as decimal numbers instead of whole numbers, e.g. 1.0 / 3.0 instead of 1 / 3.

Problem 2: Exceptional C Code *(32 points)*

(The files for this problem are in the **`cs24hw4/exceptions`** directory.)

For this problem you will create your own IA32 implementations of **`setjmp()`** and **`longjmp()`**, called **`my_setjmp()`** and **`my_longjmp()`**, as well as a comprehensive test suite for verifying the correctness of these functions.

Background

The C programming language does not include exception handling, but it is possible to implement a primitive form of exception handling using the standard functions **`setjmp()`** and **`longjmp()`**, declared in the standard C header **`setjmp.h`**. These are very interesting creatures, because they allow you to perform *non-local jumps* (i.e. not within the same procedure) in your C programs. These functions use a data-type named **`jmp_buf`**, for "jump buffer," which is used to save and restore the execution state of your program. This **`jmp_buf`** type is simply an array of integers, the number of elements depending on the platform and OS. These functions work as follows:

`int setjmp(jmp_buf buf)`

> When **`setjmp()`** is called, it stores the current execution state into **`buf`**, then returns 0.
>
> You can use **`setjmp()`** multiple times to create **`jmp_buf`**s corresponding to different points in a program's execution, since **`longjmp()`** takes a **`jmp_buf`** argument.

`void longjmp(jmp_buf buf, int ret)`

> When **`longjmp()`** is called, it restores the execution state saved in **`buf`**, causing the program to *appear* that it has returned from **`setjmp()`** a second time. However, this time **`setjmp()`** returns a nonzero value: if **`ret`** is nonzero then that is what is returned; if **`ret`** is zero then the call returns 1.
>
> You can **`longjmp()`** using a specific **`jmp_buf`** multiple times; there is no limit. The only requirement is that you can only **`longjmp()`** back into a currently executing function! Or, as the documentation puts it: "The **`longjmp()`** routine may not be called after the routine which called the **`setjmp()`** routine returns."

As a simple example, here is how these functions could be used to handle an error:

```
#include <setjmp.h>

/* Used to jump back to main() if we get a bad input. */
jmp_buf env;

/* Computes sqrt(x - 3), as long as x >= 3.  Otherwise,
 * performs a longjmp() back to an error handler.
 */
double compute(double x) {
    if (x < 3)
        longjmp(env, 1);   /* Jump to error handler! */

    return sqrt(x - 3);
}
```

```
int main() {
    double x, result;

    x = ... ;  /* Get a value for x from somewhere. */

    if (setjmp(env) == 0) {
        // setjmp() returned because setjmp() was called, not because
        // longjmp() was called.  Everything is good so far!
        result = compute(x);
    }
    else {
        // setjmp() returned because longjmp() was called.
        // Report an error.
        printf("Couldn't compute the result!\n");
    }
    return 0;
}
```

At first blush, this seems to be some crazy magic going on, but you actually already know everything you need to know to understand exactly how they work:

> The **`setjmp()`** function saves execution state associated with the registers and stack into the **`jmp_buf`** array, which obviously must be large enough to hold all of this state. Since **`setjmp()`** also follows the cdecl calling convention, it really only needs to record the callee-save registers, and the values necessary to resume execution at the point that **`setjmp()`** was called. This would include the current stack pointer, the caller's **`%ebp`**, and the caller's return address. (You will notice that the last two can be grabbed off the stack.)
>
> The **`longjmp()`** function basically does the reverse of what **`setjmp()`** does: it restores all of this execution state from the **`jmp_buf`**, so that when **`longjmp()`** returns, it *appears* that **`setjmp()`** is returning a second time. We are *not* actually returning from the **`setjmp()`** code, but the caller won't know the difference since the execution state is restored as when **`setjmp()`** was called the first time. The only difference is that the return value (in **`%eax`**) is different, which is how the caller can tell why **`setjmp()`** returned.
>
> This is also how we are able to perform a *non-local* goto with these functions: since **`longjmp()`** replaces the stack-pointer with the old value saved by **`setjmp()`**, all stack frames from intermediate function-calls are simply chopped off of the stack. In addition, since **`longjmp()`** restores the caller's return address onto the stack, when **`longjmp()`** returns then execution will end up exactly where the caller originally called **`setjmp()`**.
>
> Of course, the one final detail is that **`longjmp()`** must also restore the caller's **`%ebp`** before returning, since surely the caller will still want to access their local variables and function-arguments…

Some important things are <u>not</u> saved or restored. Only the values specified above are saved and restored, but in general the data on the stack is <u>not</u> saved or restored. This means that if the caller's local variables have been modified since the time **`setjmp()`** was called, the changes to those local variables remain.

This is also why you can only call **`longjmp()`** to return to a function that is still executing. If the function where **`setjmp()`** was called has already returned when **`longjmp()`** is called, then that

function's frame is *long gone* from the stack. When **`longjmp()`** restores the execution state from the **`jmp_buf`**, the stack will now have garbage for that frame, and you are going to get some very strange behavior…

Exceptions in C

In the **`cs24hw4/exceptions`** directory you will find a simple implementation of exception handling in C. The main files to look at are **`c_except.h`** and **`c_except.c`**. The implementation uses the **`setjmp/longjmp`** mechanism described in class, although it has been extended to support nested exceptions. Different kinds of exceptions are represented by different integer values, defined in the **`ExceptionType`** enumeration in **`c_except.h`**.

The implementation uses C macros to provide a more exception-like syntax, which makes it easier to use but more complicated to understand. C macros allow you to perform transformations on source code before it is compiled, but it is a very dangerous feature because there is little control of when and how macros are applied, or what they are allowed to do. Nonetheless, given this C-style exception-handling code:

```
TRY (
  printf("Enter your first number, the dividend:  ");
  n1 = read_double();

  printf("Now enter your second number, the divisor:  ");
  n2 = read_double();

  printf("The quotient of %lg / %lg is:  %lg\n", n1, n2, divide(n1, n2));
)
CATCH (NUMBER_PARSE_ERROR,
  printf("Ack!!  I couldn't parse what you entered!\n");
)
CATCH (DIVIDE_BY_ZERO,
  printf("Ack!!  You entered 0 for the divisor!\n");
  RETHROW;
)
END_TRY;
```

The **`TRY`**, **`CATCH`**, **`END_TRY`** and **`RETHROW`** macros in **`c_except.h`** translate the code into:

```
{
  jmp_buf env;
  int exception = setjmp(env);
  if (exception == NO_EXCEPTION) {
    start_try(&env);    /* Pushes the jmp_buf, for nested try/catch. */
    {
      printf("Enter your first number, the dividend:  ");
      n1 = read_double();

      printf("Now enter your second number, the divisor:  ");
      n2 = read_double();

      printf("The quotient of %lg / %lg is:  %lg\n", n1, n2, divide(n1, n2));
    }
    finish_try();       /* Pops the jmp_buf, for nested try/catch. */
  }
  else if (exception == NUMBER_PARSE_ERROR) {
```

```
    printf("Ack!!  I couldn't parse what you entered!\n");
  }
  else if (exception == DIVIDE_BY_ZERO) {
    printf("Ack!!  You entered 0 for the divisor!\n");

    throw_exception(exception);    /* Generated by RETHROW macro. */
  }
  else {
    /* Exception wasn't handled by any catch-block.
     * Propagate to the next enclosing try/catch block.
     */
    throw_exception(exception);
  }
};
```

Notice that when we enter the **try**-block, we call **setjmp()** and use the result to tell if an exception has occurred, and if so, which **catch**-block to handle it with. If no exception has yet occurred, we execute the body of the **try**-block, calling **read_double()** and **divide()**, either of which may throw an exception using the **THROW(e)** macro. This macro simply calls the **throw_exception()** function.

If an exception is thrown, the **throw_exception()** function calls **longjmp()** to jump to the closest enclosing **try/catch** block. This will cause the **setjmp()** function to "return a second time" (you know the *real* story now), with a result that indicates the actual exception that occurred. This causes the appropriate handler to be invoked, or if there is no handler for that exception type, the final **else**-clause causes the exception to be propagated to the next enclosing **try/catch** block. (Or, if there is none, the program will be terminated.)

Your Task: Implement and Test `setjmp()` and `longjmp()`

For this problem you must implement your own version of **setjmp** and **longjmp**, as well as specifying the size of your own **jmp_buf** type. Open **my_setjmp.h** to see what you will need to do. In this file you will see that the implementation can either use the standard versions of **setjmp()** and **longjmp()**, or by defining the **ENABLE_MY_SETJMP** symbol you can make it use your own versions, named **my_setjmp()** and **my_longjmp()**.

As stated before, these functions can only be implemented in assembly language, since they manipulate the stack in ways not possible from C. Implement these two IA32 functions in the file **my_setjmp.s**, making sure to exactly mimic the behaviors of the standard **setjmp()** and **longjmp()** functions.[1] You will also need to set **MY_JB_LEN** to be the number of elements needed to store all of your execution state.

Once you have finished implementing **my_setjmp** and **my_longjmp**, you must also create a test suite that verifies the correctness of your implementations. You will be graded on the completeness of your suite, so think carefully about the different behaviors of **setjmp()** and **longjmp()** and how to test them. (Try to devise 4 or more kinds of tests.) Here are some additional guidelines:

- **Your test program should be called `test_setjmp.c`, and it should include `my_setjmp.h`.** We would recommend that you start out using the standard **setjmp** code, not

[1] The standard **setjmp()** and **longjmp()** functions also save and restore registers for floating-point, MMX/SIMD, etc. **You don't have to do any of that!** Just worry about the general-purpose registers, as described in the problem.

your **`my_setjmp`** code. This way you can verify your test code first, and then switch over to your own version of **`my_setjmp()`** when your tests are trustworthy.

- Add a build rule to the **`Makefile`** that builds **`test_setjmp`**. Add another rule called "**`check`**" that runs **`test_setjmp`** (building if necessary), so it's easy to test your code!
- **Make sure that every test exercises a different characteristic of your code.** Each test should verify different interactions or code-paths in your functions. Testing the same code-path five times doesn't buy you anything over testing something once.
- **Every kind of test should live in its own function.** That way your main function can just call each kind of test in sequence.
- **Make sure that every test clearly reports what it is testing, and whether it passes or fails!** Test suites are frequently automated, and you need to make it obvious what is going on. (e.g. "**`longjmp(buf, 0) returns 1: PASS`**")
- Speaking of which… don't forget that **`longjmp(buf, 0)`** should cause **`setjmp()`** to return 1, but **`longjmp(buf, n)`** for *n* != 0 should cause **`setjmp()`** to return *n*. You should verify this behavior.
- You should verify that **`longjmp()`** can correctly jump across multiple function invocations. Don't make all tests **`longjmp()`** back to the same function that **`setjmp()`**s. However, for some behaviors it's fine if you use **`setjmp()`** and **`longjmp()`** in the same function, such as verifying the **`longjmp()`** return-value.
- This is trickier, but you should definitely try to verify that your functions don't corrupt the stack in certain obvious ways. For example, you might try putting local variables with known values on both sides of your **`jmp_buf`** variable to ensure that your **`setjmp()`** implementation doesn't go beyond the extent of the **`jmp_buf`**. You might also do this to ensure that **`longjmp()`** properly restores **`%esp`** and **`%ebp`**, to ensure that local variables can be accessed afterward.

Problem 3: Hot Scheming Piles of Garbage *(43 points)*

Scheme is a syntactically simple higher-level programming language that supports many interesting programming concepts such as higher-order functions. You may already be familiar with Scheme (aka "Racket") from past iterations of CS4, but if not, here is a simple example of a factorial function:

```
(define (fact n)
    (if (= n 0) 1 (* n (fact (- n 1)))))
```

In other words, if $n = 0$, $fact(0)$ will be 1; otherwise, the result of $fact(n)$ will be $n \times fact(n - 1)$.

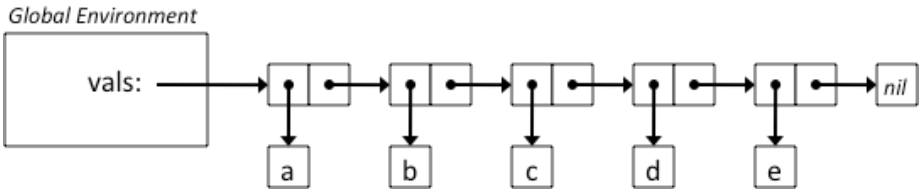
Scheme programs don't have to worry about memory management at all; they just dynamically construct values, lists, and many other such structures on the fly, and then the Scheme interpreter performs garbage collection to clean up the mess.

The simplest way to build a complex data structure in Scheme is by assembling together collections of *cons pairs*. For example, this expression constructs a sequence of cons pairs hooked into a list, and containing the specified values. The list is then bound to the name **`vals`**:

```
(define vals '(a b c d e))
```

```
;; This is identical to the above:
(define vals (cons 'a (cons 'b (cons 'c (cons 'd (cons 'e nil))))))
```

This simple expression would result in a memory layout as follows:



Since Scheme programs are themselves formulated from lists and values, it becomes natural to implement an interpreter that uses these data types internally to parse and evaluate a program.

The **`cs24hw4/scheme24`** directory contains a basic Scheme interpreter implemented in C. It contains many basic features of Scheme, but it is certainly far from complete. You can go to the above directory, type **`make`**, and then run the **`scheme24`** interpreter to try it out. For example:

```
[user@host:~/cs24hw4/scheme24]> ./scheme24
> 5
5

> '(a b c)
(a b c)

> (+ 3 5)
8

> (define (fact n)
    (if (= n 0) 1 (* n (fact (- n 1)))))
#lambda[args=(n) body=((if (= n 0) 1 (* n (fact (- n 1)))))]

> (fact 7)
5040

> ^C
[user@host:~/cs24hw4/scheme24]>
```

(You currently have to use Ctrl-C to break out of the interpreter when you are done. Additional snippets of example code are in the heavily-commented file **`examples.scm`**. Finally, this Scheme interpreter does not support tail-call optimization.)

Unfortunately, one thing the interpreter lacks is a garbage collector. Open up **`alloc.c`** (the interpreter's memory manager), and update the top of the file to say "**`#define GC_STATS`**". (No double-quotes, of course.) Then, **`make clean all`** to rebuild.

Then, do more experiments with the Scheme interpreter like the above. You will see that memory usage grows and grows and grows, without anything ever being freed.

Your Task

Update the Scheme interpreter to implement a simple mark-and-sweep garbage collector. You will complete the implementation of **`collect_garbage()`** in the file **`alloc.c`**. The

interpreter already contains many mechanisms to make this task easier. Read on for more details.

Scheme Values and the Allocator

All allocation and deallocation functions are implemented in **`alloc.c`**, and the "public" functions are declared in **`alloc.h`**. What you will notice is that only the "**`alloc_xxxx()`**" functions are made public; all **`free_xxxx()`** functions are only visible within the **`alloc.c`** module itself. The only function that is exposed by **`alloc.h`** is **`collect_garbage()`**, which (once you are finished with it) will traverse all objects reachable from the global environment and the execution stack, and reclaim any objects that are no longer reachable. This function is called at the end of every expression evaluation, so this happens *very* frequently.

Every time an object is allocated within the interpreter, a pointer to the object is recorded into a growable array of pointers. For example, when **`alloc_value()`** is called, a pointer to the newly heap-allocated **`Value`** object is stored into the **`allocated_values`** static variable. This makes it very easy to examine all objects that are currently allocated. *We don't have to guess what is or isn't a pointer; we exactly track every single object allocated in the interpreter.*

The Scheme interpreter contains three kinds of objects that need to be garbage collected. These are all declared in **`types.h`**, and there are various functions for manipulating them in **`values.h`** and **`values.c`**. They are:

- **`Value`** – a tagged data structure, containing a union that allows all supported data types to be stored in a single struct. Values of type **`T_ConsPair`** will refer to two other **`Value`**s. Values of type **`T_Lambda`** will refer to a **`Lambda`** object.
- **`Lambda`** – a data structure that represents a procedure in the Scheme interpreter. **`Lambda`**s are specified using lists of cons pairs; that is, lists of **`Value`** objects chained together. In addition, each **`Lambda`** refers to its parent **`Environment`**, so that procedures can maintain local state.
- **`Environment`** – a data structure that represents the arguments and local variables used by a procedure. Environments have a list of name/value bindings, where each value is a **`Value`** object. In addition, each environment has a parent environment that is used for variable resolution.

You will notice from the declarations of these types that they all include a **`marked`** field, which can be used for the mark-and-sweep garbage collection.

Pointer Vectors

In many places a "vector" (i.e. a growable array) of pointers is necessary to keep track of allocated objects. This data type is implemented in the **`ptr_vector.h`** and **`ptr_vector.c`** source files. (This collection is also used in the exception-handling code earlier.) The operation of this type is very simple. The struct keeps track of the current heap-allocated array, along with both its size (number of values currently stored in the array) and capacity (the total number of elements the array can hold). The functions implement various simple operations on these pointer-vectors, allowing you to manipulate them very easily.

Although you *could* access the members directly, it would be much safer to use the functions provided, since they also include assertions to help you catch bugs.

There is a second collection class used in various places, the "pointer stack." This is actually just a pointer vector, so any function that takes a pointer-vector will also take a pointer-stack.

The Marking Phase

There are two places to start when performing the marking phase of garbage collection. One is the global environment, and the other is the evaluation stack.

The Global Environment is a special environment that has no parent environment, and that never goes away. Starting with the global environment, you will need to recursively mark all values, lambdas, and other environments referenced from the environment.

The Evaluation Stack keeps track of what values are currently in use by an ongoing Scheme computation, since there are frequently temporary values being generated and then discarded. Without this stack, we could not perform garbage collection in the middle of a Scheme evaluation. When a nested expression needs to be evaluated, a new frame ("evaluation context") is pushed onto this stack, and it references the nested evaluation's intermediate values.

Here are some hints and suggestions for the mark phase:

- You should implement several functions, **`mark_environment(Environment *env)`**, **`mark_value(Value *v)`**, and **`mark_lambda(Lambda *f)`**. These functions can recursively call each other to handle various relationships between the data structures. For example, **`mark_lambda()`** can call **`mark_value()`** to mark the argument and body specifications for the lambda.

  (OPTIONAL) Recursively traversing memory structures like this can use a great deal of the stack, and can even cause stack overflows. You are not required to do this, but if you want to create a more efficient mark-and-sweep garbage collector, optimize it for some common cases: Instead of having **`mark_environment()`** recursively call itself for a non-**`NULL`** parent environment, have it iterate instead. Similarly, when **`mark_value()`** is passed a cons pair, you can recursively call **`mark_value()`** for the car, but then iterate when marking the cdr. These simple optimizations will use system resources somewhat more efficiently, since these kinds of object layouts will be very common cases.

- Environments keep variable bindings in an array named **`bindings`**. The **`num_bindings`** field specifies how many elements are valid in this array. Thus, you could have a loop like this:

```
for (i = 0; i < env->num_bindings; i++)
    mark_value(env->bindings[i].value);
```

- Some **`Lambda`**s are implemented as native C functions; these are the procedures provided by the Scheme interpreter itself. You should mark all lambdas, whether they are native or interpreted. However, only interpreted lambdas will have an **`arg_spec`** and a **`body`**, so you don't mark these fields when the lambda is native (i.e. when the **`native_impl`** flag is 1).

- As with the environments, you will probably want to implement a function called **`mark_eval_stack`** that specifically handles the evaluation stack.

- The evaluation stack is a bit subtler than the global environment. Any of the first three members can be **`NULL`** if an evaluation doesn't require those values. Also, the local_vals member actually stores **`Value**`** (pointer-to-pointer-to-**`Value`**) elements, so you must

dereference the **`Value**`** to get to a **`Value*`** for marking. Also, just as before, the **`Value*`** might actually be **`NULL`**, so you must check for this. You will end up with code like this:

```
Value **ppv = (Value **) pv_getelem(&ctx->local_vals, idx_local);
if (*ppv != NULL)
    mark_value(*ppv);
```

The Sweep Phase

Once you have successfully marked all reachable objects, you should be able to sweep through the set of all allocated objects, freeing ones that are no longer reachable. Again, **`alloc.c`** contains helpful details to simplify this process:

- There are three global variables in **`alloc.c`**: **`allocated_values`**, **`allocated_lambdas`**, and **`allocated_environments`**. These are vectors of pointers to all objects that are currently allocated. Thus, it is very easy to traverse these vectors and remove unreachable objects. You can see the **`ptr_vector.h`** file for operations that you can perform on pointer-vectors. (Note that the functions take a struct-pointer, not the struct itself, since that would be too expensive to pass. You can simply pass **`&allocated_values`**, etc., to these functions.)
- Since C doesn't have very good support for abstract data types, these vectors store untyped pointers (i.e. **`void *`**). Thus, you will need to cast them when you get them out of the vector. For example, to retrieve **`Lambda i`** from the vector of allocated lambdas, you might do this:

  ```
  Lambda *func;
  ...
  func = (Lambda *) pv_get_elem(&allocated_lambdas, i);
  ```

- You should create three helper functions, **`sweep_values()`**, **`sweep_lambdas()`**, and **`sweep_environments()`**, each of which traverses the corresponding pointer-vector mentioned above. Each of these helper functions will simply look at each allocated object of a particular type. If the object is marked, unmark it for the next garbage collection. If the object is unmarked, delete it (using the **`free_value()`**, **`free_lambda()`**, etc. functions mentioned earlier) and set the pointer to **`NULL`**.

  Note that all pointers in the pointer-vectors should either point to valid objects, or they should be set to **`NULL`**! So when you delete an object, make sure to set the pointer-vector entry to **`NULL`**, to record that the object is now gone.
- At the end of each of your **`sweep_xxxx()`** functions, call **`pv_compact()`** on the corresponding pointer-vector to remove all **`NULL`** entries created from deleting objects. This way your pointer-vectors won't grow out of control. (You can turn on verbose-output at the top of **`ptr_vector.c`** if you want to make sure your vectors are updated properly.)

Testing!

If you did everything correctly, you should see much better memory performance when you use the Scheme interpreter. Of course, run several commands, and run some more complicated operations, so that you can be sure your interpreter deletes exactly what should be deleted; no more, no less.

To help you ensure that you properly garbage collect all the different kinds of Scheme constructs, you should try the code snippets in **`examples.scm`**. See the comments in this file for details!