The switch/case construct

Given int expression e, constants c1, c2, ..., cn, the following are roughly equivalent:

```c
switch (e) {
    case c1:
        <statement 1>
        break;
    case c2:
        <statement 2>
        break;
    ...
    case cn:
        <statement n>
        break;
    default:
        <default statement>
        break;
}
```

Note: the 'break;' statements in the cases of the switch are important: without a 'break;', control falls through to the following case. This design choice can easily lead to unexpected behaviors. Consider:

```c
switch (x % 2) {
    case 0:
        printf("x is even\n");
    case 1:
        printf("x is odd\n");
}
```

You can make hella elegant code sometimes by exploiting the fallthrough behavior of cases:

```c
switch(x) {
    case 1:
    case 4:
    case 9:
    {
        int sx = isqrt(x);
        printf("sqrt(x) = %d\n", sx);
        break;
    }
    default:
        printf("not a perfect square...\n");
}
```

The entire main loop of your VM will be one big switch statement. Be wary of fallthrough!
Structs that aren’t pointers

In C0 and in much of our C programming, we have accessed structs through pointers almost exclusively.

typedef struct stack * stack;
struct stack {
    list top;
};

... stack S = stack_new(); ...
... S->top ...

Sometimes it’s convenient to access structs directly, though, like when they’re stored in an array. To access a field f of a struct expression e, one uses the syntax ‘e.f’.

stack stack_array[] = ...

... stack_array[i].top ...

In fact, the familiar notation ‘p->f’ for accessing a struct field through a pointer p is just syntactic sugar for ‘(*(p).f)’, and since array indexing is just syntactic sugar for pointer arithmetic, the above expression really amounts to ‘(*(stack_array+i)).top’, or (stack_array+i)->top.

Accessing structs directly will be useful when dealing with various pools.

Flexible array members

Usually, structs aren’t allowed to contain arrays directly, only pointers, but C99 allows the last member of a struct to be an array of unknown size. Just as with stack-allocated arrays, the array will be contiguous in memory with the earlier struct elements. Flexible array members are used by the "bare" C0 runtime to implement the struct c0_array, an array stored with its length and the size of its elements:

```
struct c0_array {
    int count;
    int elt_size;
    char elems[];
};
```

The sizeof operator on a struct with a flexible array member returns the size the struct would have if the array had length 0. To allocate a struct with a flexible array member, request space for the struct itself plus however much space you want for the array. For instance:

```
struct c0_array *a = xcalloc(1, sizeof(struct c0_array) + 23);
```

allocates space for the c0 representation of an array containing 23 bytes, perhaps a char array. The fields can be accessed as usual using struct-pointer notation:

```
a->count = 23;
a->elt_size = 1;
... a->elems[17] ...
```

An alternative that would be equivalent under most implementations is to simply allocate sizeof(int) bytes for the first int plus sizeof(int) bytes for the second int plus 23 bytes for the array, giving you a block of memory laid out as follows:

```
[<- 4b ->][<- 4b ->][------------- 23 bytes -------------]
```
Casting pointers \(< - >\) integers, signed \(< - >\) unsigned

C99 leaves the behavior of casting pointers to integral types and the representation of signed integers up to the implementation -- they are both "implementation-defined" behaviors. For this assignment, we look to GCC’s documentation:

GCC supports only two’s complement integer types, and all bit patterns are ordinary values. (http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Integers-implementation.html)

A cast from pointer to integer discards most-significant bits if the pointer representation is larger than the integer type, sign-extends1 if the pointer representation is smaller than the integer type, otherwise the bits are unchanged.

A cast from integer to pointer discards most-significant bits if the pointer representation is smaller than the integer type, extends according to the signedness of the integer type if the pointer representation is larger than the integer type, otherwise the bits are unchanged. (http://gcc.gnu.org/onlinedocs/gcc-4.3.5/gcc/Arrays-and-pointers-implementation.html)

We encourage you to leverage these implementation decisions to implement reliable signed modular arithmetic using unsigned modular arithmetic and to store integer types directly onto a stack that holds elements of type void *.

The macros INT(p) and VAL(x) cast back and forth between void* pointers and ints, and we always have INT(VAL(x)) == x. INT(p) is not defined, unless p was obtained by casting an int (or is equal to NULL). You can see the definition of these macros in the file c0vm.h.

Coping with unused variables

When compiling with `-Wall -Wextra`, a warning will be generated if any variable is unused, and when compiling with `-Werror`, that warning turns into an error:

```
cc1: warnings being treated as errors
foo.c: In function 'bar':
foo.c:12: error: unused variable 'x'
```

Sometimes this indicates a bug in your program, but sometimes it just indicates that you’re program is incomplete, and you may want to compile regardless in order to test the partial functionality you have written thus far. You might think to suppress the error simply by mentioning the unused variable on a line by itself:

```
x;
```

but that just triggers another warning/error:

```
cc1: warnings being treated as errors
foo.c: In function 'bar':
foo.c:12: error: statement with no effect
```

The proper solution is to mention the unused variable on a line by itself but additionally indicate your intent to ignore its value by casting it to void:

```
(void) x;
```

That should eliminate the warning and get you back on track to compiling and testing your code.