Question 1:

Assuming static scoping and call by reference, we go through the following code:

```
1  {
2      int x = 0;
3      int A(reference int y) {
4          int x = 2;
5          y = y + 1;
6          return B(y) + x;
7      }
8
9      int B(reference int y) {
10         int C(reference int y) {
11             int x = 3;
12             return A(y) + x + y;
13         }
14         if (y == 1)
15             return C(x) + y;
16         else
17             return x + y;
18     }
19     write(A(x));
20 }
```

Initially `x = 0` and `A()` is called. Within `A()`, `x` is now aliased as the parameter name, `y`, meaning the local `x` is initialized as 2 and the aliased `x` is incremented to 1. `B()` is then called with a reference to the same variable, which is now 1.

Within `B()`, it is checked whether the input parameter `y == 1`, which as just mentioned, it is. This means `C()` is called with `x`, but no local `x` is defined within `B()`. Due to static scoping, this `x` must then refer to the one defined in `B()`'s containing block. Thus, `x` and `y` within `C()` now refer to the same variable with value 1.

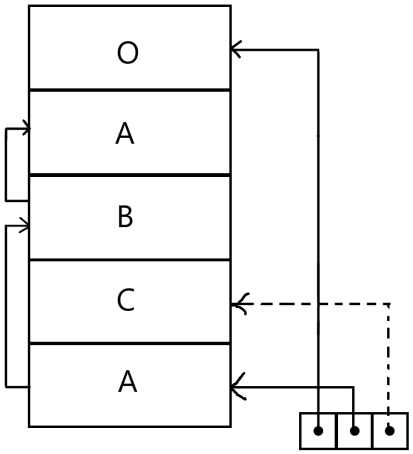
Within `C()`, the parameter `y` refers to this variable with the value 1. Locally, `x` is initialized to 3. In the return statement, `A()` is now called with the same variable as the first call, but now with a different value. This variable is now incremented to 2 and `B()` is called again. Now, `y != 1` so the **`else`** block is executed. Again, `x` and `y` still both refer to the same variable with value 2, so 4 is returned.

In `A()` this is added to 2, meaning it returns 6 to `C()`. In `C()`, the line `A(y) + x + y` then becomes `6 + 3 + 2`, since `x` is defined locally and `y` references the variable just incremented in `A()` to 2. The result returned to `B()` is 11.

This is now added to `y`, which once again refers to the same variable with value 2, meaning 13 is returned to `A()`. Finally, the local `x`, which is 2, is added and the result of 15 returned.

So the program in the end prints the value 15.

With a *display*, a vector is used with an element for each nested block. Naming the outer block O and the function blocks with the function names, we get the following display and activation record stack:



The display has an element for the outermost level, for the level containing `A()` and `B()`, and for the level containing `C()`. The pointers to `O` and `C` are unaltered throughout, but the pointer to `A` first points to `A`, then to `B` when `A()` calls `B()` and then back to `A` when `C()` calls `A()`. For both these switches, the previous display is stored in the activation record.

Question 2:

Call by name is (almost) simply textually replacing all occurrences of the parameters in the function body with the arguments input. This means that variables can be *captured*. As an example:

```
int func(name int n) {
    int m = 1;
    return n + 2;
}
int m = 0;
func(m);
```

Calling `func(m)` results in the textual replacement:

```
int func(name int n) {
    int m = 1;
    return m + 2;
}
int m = 0;
func(m);
```

Because of this simple replacement, `m` now seems to refer to the `m` declared within the function body. The actual parameter was *captured* by the local variable.

A simple but bad solution is to disallow the programmer to ever use the same variable name twice. Instead, a better technique is to never let a variable be captured. This is done by always evaluating the parameter within the environment of the caller instead of the function body.

We now look at the following code:

```
{
    int x = 5;
    int P(name int m) {
        int x = 2;
        return m + x;
    }
    write(P(x++) + x);
}
```

This uses call by name and static scoping. Initially, `x = 5`. Upon calling `write()` the expression `P(x++) + x` is evaluated. This is done by first evaluating the left side, then the right, and then summing. So, first `P(x++)` is called, producing within the function the code:

```
int x = 2;
return x++ + x;
```

Again, first the left-hand side is evaluated. The `x` at the left is not captured but is evaluated within its own environment. With static scoping, this environment is defined by the block which is was declared within, meaning the value here is 5. It is then incremented, meaning it will be 6 when used again. The right-hand side refers to `x` within its own environment, meaning its value is 2. The returned value is thus 7.

Now the right-hand side within `write()` is evaluated to 6, since this `x` was incremented within the `P()`. The results are added and the final result printed is 13.

Question 3:

Mark and sweep is a technique used in garbage collection. It has the phases *mark* and *sweep*. When marking, the garbage collector first traverses all objects in the heap and marks each as unused. Then all active pointers on the stack are followed, traversing all the used data structures in the heap recursively. Each of these are then marked as used. Now all unused objects are known and the heap is *swept* and each unused block is put back in the free list.

When traversing the data structures during the mark phase, the collector follows the pointers within each block recursively. To get back to the parent of a block, the return pointers need to be stored meanwhile on the stack. The problem is, that the garbage collector is meant to use the technique when the memory is almost used up, meaning there might be little space left for the stack to grow. To use less memory during the mark phase, *pointer reversal* can be used. When visiting a parent's subtree, the pointer in the parent to its child is reversed so it now points to the parent's own parent. Doing this while traversing makes it very easy to retrace a node further up in the tree by following the pointer to its parent. Upon retracing, the pointer is once again flipped and point to the original child. This allows the tree to be traversed in both directions while reusing the space already used by the structures instead of the stack. For this to work, pointers to the current node and its parent must be stored, but just these two needs space on the stack.

Mark and compact has the exact same mark phase as mark and sweep. The second phase is the difference. Instead of just sweeping the unused blocks, the used blocks are moved next to each other, compacting them together, leaving one large block of used memory and one large block of free.

The pros of using this is that fragmentation is eliminated. This also makes the handling of the free list simpler, as it is simply a contiguous block. It also helps locality of reference, which can make programs more efficient. The cons are the compaction phase, which requires multiple passes over the heap.

Pros of using the mark and sweep are that it runs faster than mark and compact and is simpler to implement. Cons are that it creates external fragmentation, that it still is inefficient, and that used blocks are scattered all around, decreasing locality of reference.

Question 4:

Type equality is when two types are formally different, but can be considered interchangeable.

Type equality is split into equivalence by name and structural equivalence. Equivalence by name is really only if two types have the same name, but *weak* equivalence by name is when renaming of a type just creates an alias, rather than a new type. Structural equivalence says that two types are structurally equivalent if their structure is identical.

Type compatibility is when a value of one type can be used whenever the value of another is to be used. It allows a value or expression to be used in the context of another type. It may not be true the other way around. An example where it is not true is subclassing in Object-Oriented languages. An example in Java could be:

```
1  class Person {
2      String name;
3      int age;
4  }
5
6  class Student extends Person {
7      float avg_grade;
8  }
```

Here, `Student` can be used anywhere a `Person` is expected, since it will inherit the fields of `Person`. But the field `avg_grade` is specific to the `Student` class. Attempting to access this from a non-student `Person` will result in an error.

Question 5:

This answer depends on your project, how you have implemented the data types and how you parse the input string to the game configuration state type.

Question 6:

When flipping an image horizontally, each subsquare within a parent square stays within its immediate parent. Within the parent, the subsquares are swapped. The top squares are swapped with the squares immediately below and vice versa. Each subsquare must also recursively be flipped. The base case is reaching a leaf, which should remain untouched. This function is created and named `flipQT`. With this function, `myFlip` should simply `map` over the input list with `flipQT`:

```
1 | myFlip :: [QT a] -> [QT a]
2 | myFlip = map flipQT
3 |
4 | flipQT :: QT a -> QT a
5 | flipQT (C a) = C a
6 | flipQT (Q tl tr br bl) = Q (flipQT bl) (flipQT br) (flipQT tr) (flipQT tl)
```

So the base case is a single colour, which is left untouched. In the case of a subtree, the top-left and bottom-left squares are swapped and the top-right and bottom-right swapped. Each subsquare runs the function recursively.

Question 7:

The function `zipWith` needs to be applied recursively to all parts of the two trees. Upon reaching a leaf in both trees, the function should be applied to the values in the leaves, resulting in a new leaf. If in one tree a leaf was reached but in the other, the node still has children, the leaf encodes the information in all the same pixels as all the children of the node in the other tree do. This means the function should be applied recursively to each child of the other tree with the leaf as the other argument.

The final case is two subtrees. Here `zipWith` should be applied to the same nodes in each subtree recursively:

```
1  zipWith :: (a -> b -> c) -> QT a -> QT b -> QT c
2
3  zipWith f (C a) (C b) = C $ f a b
4
5  zipWith f (Q tl1 tr1 bl1 br1) (Q tl2 tr2 bl2 br2)
6      = Q (zipWith f tl1 tl2) (zipWith f tr1 tr2) (zipWith f bl1 bl2) (zipWith f br1 br2)
7
8  zipWith f leaf (Q tl tr bl br)
9      = Q (zipWith f leaf tl) (zipWith f leaf tr) (zipWith f leaf bl) (zipWith f leaf br)
10
11 zipWith f (Q tl tr bl br) leaf
12     = Q (zipWith f tl leaf) (zipWith f tr leaf) (zipWith f bl leaf) (zipWith br f leaf)
```

Question 8:

The keyword `Monad` is a type class, which is used to wrap imperative behaviour in a functional setting. Monads allow programmers to compute things sequentially, which makes it easier to structure functional programs and is used for e.g. state and I/O.

The possible operations to define for the monad `M` would be the following:

- `return`: Takes a value and wraps it in the monad
- >>=: The bind function takes a monadic value and a function from a normal to a monadic value, applies the function to the normal value and returns the result as a monadic value. This makes it easy to bind monadic computations
- >>: Used as >>=, except it does not take a function. It is used whenever the function specified would have ignored its input and always would output a predetermined value - so it can bind computations where the input is ignored in the function.
- `fail`: What to do in case of failure, e.g. a failure in pattern matching in a `do` block.

These have the following type signatures:

- `return :: a -> m a`
- `(>>=) :: m a -> (a -> m b) -> m b`
- `(>>) :: m a -> m b -> m b`
- `fail :: String -> m a`

The operations for the minimal definition of a monad is `return` and >>=. `fail` is as a default simply `error` and >> is defined in terms of >>=:

```
a >> b = a >>= \_ -> b
```

We now take a look at the expression:

```
guard (5 > 2) >> return "cool" :: [String]
```

The type annotation at the very end, `:: [String]`, is used after the entire expression has been evaluated.

The comparison `(5 > 2) = True`, so the `guard True` pattern is matched, resulting in `return ()`. The return type is `m ()`, so this is actually a `MonadPlus` and thus a `Monad`. The type of this is inferred later. Thus, we now have:

```
return () >> return "cool" :: [String]
```

The `return` call is used on the list `"cool"`. For lists this returns a list of the input, resulting in

```
return () >> ["cool"] :: [String]
```

Since the bind operator, >>, takes two monads of the same type, `return ()` must also be wrapped in a list. This is fine, since it should be wrapped in a `MonadPlus`, which lists are. We get the result:

```
[()] >> ["cool"] :: [String]
```

The bind operator >> is defined in terms of >>=, which for lists calls `map` with the given function and list and then flattens the result with `concat`. This means >> for lists is defined as:

```
xs >> ys = xs >>= \_ -> ys
         = concat $ map (\_ -> ys) xs
```

Using this with the previous result, we get:

```
concat $ map (\_ -> ["cool"]) [()]
= concat [["cool"]]
= ["cool"]
```

The result is thus `["cool"] :: [String]`, which is just `["cool"]`

Following the same logic for

```
guard (1 > 2) >> return "cool" :: [String]
```

we now get `mzero` when calling `guard` with `(5 > 2) = False`:

```
mzero >> return "cool" :: [String]
```

We get the same result for `"cool"` and thus `mzero` is the result defined in the `MoandPlus` class for the list. This is the empty list, resulting in:

```
[] >> ["cool"] :: [String]
```

This is then evaluated as:

```
concat $ map (\_ -> ["cool"]) []
= concat []
= []
```

So the result is the empty list, type annotated by `:: [String]` to ensure the type of the list is still a list of strings, just empty.

Finally, we have the following code:

```
do
    x <- [1, 2]
    y <- [1, 2]
    guard $ x <= y
    return (x + y)
```

Intuitively, `x` and `y` are assigned all possible combinations of `1` and `2`, then each is checked with `guard`. If the first is no greater than the second, their sum is put in the final list. If it is, an empty list is put inside. The result is concatenated and the empty lists thus disappear.

In more detail, we can start to desugar:

```
[1, 2]         >>= (\x ->
[1, 2]         >>= (\y ->
guard $ x <= y >>
return (x + y)))
```

Which again is:

```
[1, 2] >>= (\x -> [1, 2] >>= (\y -> guard $ x <= y >> return (x + y)))
```

For the left-most bind, we map the function inside over the first list and concatenate the result. So for each of `[1, 2]`, the function inside does:

```
[1, 2] >>= (\y -> guard $ x <= y >> return (x + y))
```

Continuing with the first element, 1, as an example, this becomes:

```
[1, 2] >>= (\y -> guard $ 1 <= y >> return (1 + y))
```

This maps the function after the bind over each element in `[1, 2]`. For each, the following function is run:

```
guard $ 1 <= y >> return (1 + y)
```

So again with the first element in the next list, 1, as an example, the following is run:

```
guard $ 1 <= 1 >> return (1 + 1)
```

Just as with the examples before, this compares the values, here `1` and `1`. If `1 <= 1`, the value from `return (1 + 1)` is used. If not, `mzero` is used. Since the first bind operators operate on lists, the `return` here is also the one defined for lists. This means either `[1 + 1]` or `[]` is returned.

The result for the first element in the first list is thus:

```
concat [[1 + 1], [1 + 2]]
```

and for the second:

```
concat [[], [2 + 2]]
```

The first element here is empty, since `2 > 1`. In total, the result becomes:

```
concat [concat [[1 + 1], [1 + 2]], concat [[], [2 + 2]]]
= concat [[2, 3], [4]]
= [2, 3, 4]
```

So the final result is `[2, 3, 4]`