1.

(a) **Answer:**
Because \( \oplus \) will add the elements of the first argument (the result from the left sub-tree) to the second argument one at a time, so each element is copied a number of times equal to the number of left branches between the root and the node containing the element.

(b) **Answer:**

```ocaml
define treeToListApp2 : tree -> list
let rec treeToListApp2 t acc = match t with
| Leaf -> acc
| Node (l, x, r) -> (treeToListApp2 l (x::treeToListApp2 r acc))
```

(c) **Answer:**

regionContains checks whether a region contains a point. For example:

```
regionContains myRegion (2.0, 2.0)
```
returns true because (2.0, 2.0) is in the rectangle with corners (1.0, 1.0) (4.0, 5.0) in the second part of the union. Similarly:

```
regionContains myRegion (-1.0, -1.0)
```
returns true because (-1.0, -1.0) is outside the rectangle that is inverted in the first part of the union. (Note: your explanations need not be quite so detailed here.)

It is easy to implement because regions are represented by functions that return true when applied to points in the region, and false otherwise - hence the region itself essentially already does what regionContains needs to do when applied to the region.

(d) **Answer:**

```ocaml
let regionInverse reg pt = not (reg pt)
let myRegion = regionUnion (rectangle (1.0, 1.0) (4.0, 5.0)) (regionInverse ( rectangle (0.0, 0.0) (2.0, 3.0) ))
```

2.

(a) **Answer:** In this code, each time an element is added to the path so far, the result (newPath) is used for both recursive calls, which results in sharing between the lists in the set that is constructed.

This leads to a major advantage: the space used to store all the lists in the result is roughly the number of nodes in the input. Without sharing each list would require space equal to its length, so the total space required would be \( k \) times as large without sharing, where \( k \) is the average length of the paths.
(b) **Answer:** The imperative version has one major disadvantage compared to the original one: to work correctly when the different lists in the result are modified, the sharing must be removed. This means that much more space is required, as explained in the previous question.

(c) Alas, to be closer in difficulty to what you might have on the exam, I changed this question a couple of times, but I seem to have changed the answer without changing the question properly.

Originally it was “find the largest path containing only even numbers”. (Hence the List.forAll in the hint.)

Then I changed it to the question that appears on the sample exam handout. For that question, here’s two solutions. The second is nicer, but I certainly don’t expect you to produce similar code under exam situations.

**Answer:**

```ml
// First version
let maxEvenPath t =  
    seq { for p in allPaths t ->  
       ( seq{for n in p do if n%2=0 then yield n} |> Seq.length, p)  
    } |> Seq.max |> snd |> Seq.length

// Second version
let numEvens = Seq.filter(fun x -> x%2 = 0) >> Seq.length
let maxEvenPath = allPaths >> Seq.maxBy numEvens >> Seq.length
```

Finally I decided this was still too difficult compared to the actual exam, so I chose an easier question “find the maximum number of even numbers in a path”. For that question, here’s my the answer (but attempt it yourself first):

**Answer:**

```ml
let maxEvenPath t =  
    Seq.max (seq {for p in allPaths t ->  
        Seq.length (seq {for n in p do if n%2=0 then yield n}))
```

3.

(a) **Answer:** Aliasing is when there are many different ways of referring to the same modifiable location, such as when two variable names refer to the same thing.

```ml
let incBoth (a:int ref) (b:int ref) =  
a := !a + 1  
b := !b + 1  // When a and b are the same cell, we end up adding 2.
```
The above would be a sufficient answer, but the below is probably a better example of “complicated behaviour”.

```fsharp
let modifyArray (a:int ref[]) =
a.[0] := !a.[0] * 2
a.[1] := !a.[1] + 1

let arr = Array.create 5 (ref 2)
modifyArray arr // In fact, each location in arr contains the same cell.

// The cell contains 5 after the above call.
```

(b) Yes, we can roll our own objects in Java, but the resulting code will be much more complicated than the given F# code. This is because building objects that don’t directly depend on the way objects work in Java would require a class for each kind of method. This might be desirable when attempting to interface with code that uses a kind of objects quite different to Java’s objects.

(c) (i) \text{revAppend}([1,2,3],[], ZS) \text{ succeeds with } ZS=[3,2,1].

Main steps:

```fsharp
\text{revAppend}([1,2,3],[], ZS) :- \text{revAppend}([2,3],[1],ZS). // 2nd rule
\text{revAppend}([2,3],[1],ZS) :- \text{revAppend}([3],[2,1],ZS). // 2nd rule
\text{revAppend}([3],[2,1],ZS) :- \text{revAppend}([], [3,2,1],ZS). // 2nd rule
\text{revAppend}([], [3,2,1],[3,2,1]). with ZS = [3,2,1] // 1st rule
```

(ii) \text{revAppend}(XS, YS, [1,2]) \text{succeeds three times, then loops forever.}

```fsharp
\text{revAppend}(XS,YS,[1,2]) :- \text{revAppend}([], [1,2],[1,2]). // 1st rule
SUCCEEDS
BACKTRACK (for more solutions)

\text{revAppend}([X1|Y1],YS,[1,2]) :- \text{revAppend}(Y1,[X1|YS],[1,2]). // 2nd rule
with XS=[X1|Y1]

\text{revAppend}([], [1,2],[1,2]). with Y1=[], X1=1, YS=[2], XS=[1] // 1st rule
SUCCEEDS
BACKTRACK (for more solutions)

\text{revAppend}([X2|Y2],[X1|YS],[1,2]) :- \text{revAppend}(Y2,[X2|X1|YS],[1,2]). // 2nd rule
with Y1=[X2|Y2]

\text{revAppend}([], [1,2],[1,2]). with Y2=[], X2=1, X1=2, YS=[], XS=[2,1] // 1st rule
SUCCEEDS
BACKTRACK (for more solutions)
revAppend([X3|Y3],[X2|X1|YS],[1,2]) :- revAppend(Y3,[X3|X2|X1|YS],[1,2]). // 2nd with Y2=[X3|Y3]
// This then continues using the 2nd rule forever

(This is a little harder than you'll get on the exam.

(iii) revAppend(XS, [2,3], [1,2,3]). Succeeds once, then loops forever.

revAppend([X1|Y1],[2,3],[1,2,3]) :- revAppend(Y1,[X1,2,3],[1,2,3]). // 2nd rule with XS=[X1|Y1]

revAppend([], [1,2,3], [1,2,3]). with Y1=[], X1=1, XS=[1] // 1st rule SUCCEEDS
BACKTRACK (for more solutions)

revAppend([X2|Y2],[X2,X1,2,3],[1,2,3]) :- revAppend(Y2,[X2|X1|YS],[1,2]). // 2nd with Y1=[X2|Y2]

// This then continues using the 2nd rule forever

4.

(a) A thread is a sequence of instructions from a program that are executed in order, one after another.

Having multiple threads means there is more than one such execution sequence involved in a program, with each independently progressing through its own sequence of instructions. Multiple threads are useful because some programs need to do more than one thing at a time (or at very similar times). Reasons for this include dealing with real world events that may be concurrent, servers responding to many clients, keeping user interfaces responsive, keeping the CPU busy while I/O is done, and using multiple cores.

[You wouldn’t have to include all of these examples for full marks.]

(b) (i) Answer:

No, because the loop at the start of NewRequest is outside the lock. Thus, another thread could take the resource, making inCharge false before the lock is acquired just after the loop exits.

This can be fixed by moving the lock line to the first line, so that it includes the whole of verb!NewRequest!.

(ii) The interleaving involves threads for clients 0, 1 and 2:

Thread 0 calls NewRequest, and obtains the lock on Client 0, then proceeds through the Sleep and releases the lock.
Thread 2 calls NewRequest, and takes the resource from client 0. Thread 0 calls NewRequest, and obtains the lock on Client 0.
Thread 0 calls CanIHaveIt on Client 0, which fails.
Thread 1 calls NewRequest, and obtains the lock on Client 1.
Thread 1 calls CanIHaveIt on Client 0, which waits for the lock on Client 0.
Thread 0 calls CanIHaveIt on Client 1, which waits for the lock on Client 1.
Deadlock: Threads 0 and 1 are each waiting for a lock held by the other.

(iii) Increasing the granularity will not help in this case, since we are already locking just a single client, and clients don’t contain other objects that we could lock instead. Decreasing the granularity can resolve the deadlock: we can just have all clients lock the same object (e.g., a special reference cell created just for this purpose).

5.

(a) **Answer:**

<table>
<thead>
<tr>
<th>Account</th>
<th>Other balance</th>
<th>Balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>Account 1</td>
<td>-9000</td>
<td>1000</td>
</tr>
<tr>
<td>Account 2</td>
<td>-9000</td>
<td>1000</td>
</tr>
<tr>
<td>Account 1</td>
<td>1000</td>
<td>1000</td>
</tr>
<tr>
<td>Account 2</td>
<td>1000</td>
<td>1000</td>
</tr>
</tbody>
</table>

This can happen when:

(i) All 100 transfer messages are queued for each agent, followed by the balance request from the thread for the other account.

(ii) Each agent receives 100 transfer messages, reduces its balance, and sends 100 deposit messages to the other agent.

(iii) Each agent receives the request for the balance from the thread for the other account and returns -9000.

(iv) The request for the balance from each agents own thread is placed in its queue.

(v) Each agent receives 100 deposit messages, increasing its balance back to 1000.

(vi) Each agent responds to it’s own balance request with 1000.

(b) **let!** is used to run one asynchronous computation as part of a larger one. **let** is used to run a normal F# computation synchronously, as usual.

Both are needed because it is often necessary to have both asynchronous code and ordinary F# synchronous code as part of an asynchronous computation.

(c) STM is a technique for coordinating concurrency that allows the programmer to designate transactions that should run atomically (without interference). It generally uses a log of
reads and writes by each transaction to detect interference, and to rollback the actions of transactions that interfere.

It prevents deadlocks because threads do not wait for each other. Instead they continue optimistically, and are rolled back and retried if interference occurs.

6.

(a) Starting with 
\[(\text{fun } a \ b \rightarrow \text{if } a \ b \text{ False}) \ True\]

Substituting the definitions gives:

\[(\text{fun } a \ b \rightarrow (\text{fun } x \ y \ z \rightarrow x \ y \ z) \ a \ b \ (\text{fun } x \ y \rightarrow y)) \ (\text{fun } x \ y \rightarrow x)\]

Now, doing normal-order reduction means repeatedly finding the leftmost \text{fun} that is applied to an argument, and putting the (first) argument in place of the (first) variable for that \text{fun}. So, the reduction steps are:

\[
\begin{align*}
\text{fun } a \ b \rightarrow (\text{fun } x \ y \ z \rightarrow x \ y \ z) \ a \ b \ (\text{fun } x \ y \rightarrow y)) \ (\text{fun } x \ y \rightarrow x) \\
= \text{fun } b \rightarrow (\text{fun } x \ y \ z \rightarrow x \ y \ z) \ (\text{fun } x \ y \rightarrow x) \ b \ (\text{fun } x \ y \rightarrow y) \\
= \text{fun } b \rightarrow (\text{fun } y \ z \rightarrow (\text{fun } x \ y \rightarrow x) \ y \ z) \ b \ (\text{fun } x \ y \rightarrow y) \\
= \text{fun } b \rightarrow (\text{fun } z \rightarrow (\text{fun } x \ y \rightarrow x) \ b \ z) \ (\text{fun } x \ y \rightarrow y) \\
= \text{fun } b \rightarrow (\text{fun } x \ y \rightarrow x) \ b \ (\text{fun } x \ y \rightarrow y) \\
= \text{fun } b \rightarrow (\text{fun } y \rightarrow b) \ (\text{fun } x \ y \rightarrow y) \\
= \text{fun } b \rightarrow b
\end{align*}
\]

(b) A most general unifier is the most general type that is an instance of each of two given types.

As an example, the m.g.u. of \('a*'b \rightarrow 'b*'a\ and \('c \rightarrow 'c\) is \('d*'d \rightarrow 'd*'d\). It is an instance of the first by substituting \('a = 'd\ and \('b = 'd). It is an instance of the second by substituting \('c = 'd*'d\). And, for each of these substitutions, choosing something more general results in a type that is no longer an instance of the other type.

(c) Main steps:

(i) Allocate each subexpression/pattern a different type variable.

(ii) Generate type equations relating the type variables for each construct used in the program.

(iii) Solve the equations using unification, reporting an error if unification fails.

(d) (i) Reference counting.

(ii) Mark and Sweep
(iii) Copying

One main difference is that reference counting keeps a count of the number of references to each cell, and immediately frees a cell when this reaches zero, while the other two instead periodically trace the reachable cells and free the rest.

Mark and Sweep differs from Copying in that it marks each reachable cell while tracing and then frees all unmarked cells, while Copying copies the reachable cells to another area of memory so that the whole of the original area is no longer needed.