1. Why is heap fragmentation not a problem in a traditional Lisp implementation where "cons" is the only heap storage allocation primitive?

   Because each cons cell is the same size, there will be no internal or external fragmentation. There is no internal fragmentation because the cons cell is completely filled by the head and tail. There is no external fragmentation because there will not be any block too big to fit in any available memory space.

2. Consider the following OCaml program. Its names have been changed to protect the innocent.

```
let s x = x + 1

let rec i min max =
  if max < min then [] else min :: i (s min) max

let rec f p = function
 | [] -> []
 | a::r ->
   let fpr = f p r in if p a then a::fpr else fpr

let x n =
  f (fun m -> not (m mod n = 0))

let t max =
  let rec f1 = function
    | [] -> []
    | n::r ->
       n :: if max < n*n then r else f1 (x n r)
  in
    f1 (i 2 max)
```

   (a) What are the types of the top-level OCaml identifiers `s`, `i`, `f`, `x`, `t`?

```
s : int -> int
i : int -> int -> int list
f : ('a -> bool) -> 'a list -> 'a list
x : int -> int list -> int list
t : int -> int list
```

   (b) What does the expression "`t 20`" evaluate to? Show your work.

   What do the functions do?

   `s` increments its argument.

   `i` creates a list of integers from min to max (inclusive on both ends, because when min=max, the else branch is taken, adding min(=max) to the list)

   `f` acts as a filter. It accepts a predicate and a list, and iterates through the elements of the list. When an element passes "p a", it is added to the result list, otherwise it is dropped.

   `x` creates a curried function, by partially evaluating f with a predicate which returns true if some element of the list is not divisible by the parameter, n. So (x n l) would return a list which is l with all elements divisible by n removed.

   `t` is the Sieve of Eratosthenes. It accepts an integer max, creates a list from 2 to max, and applies it to f1. f1 puts the first prime number, 2, on the result list, and filters out all elements in the rest of the list divisible by 2. It does this on each successive prime remaining in the list until max ¡ n*n, which results in a list of primes from 2 to max (inclusive).

$$\texttt{t 20} = [2, 3, 5, 7, 11, 13, 17, 19]$$

3. The paper "The Twisted Network Framework" gives the following as one of the disadvantages of Twisted.

   "As each callback must be finished as soon as possible, it is not possible to keep persistent state in function-local variables."

What does this remark mean? Give an example of why it might be nice to keep persistent state ina function-local variable, and describe what would happen to a Twisted server if you tried to do that.

This remark is saying that you cannot keep persistent state in a function-local variable in Twisted the same way you can in a multithreaded app, because functions must finish quickly so as to not block the single thread.

A desirable use case for a function-local variable may be a file descriptor or a database handle which you are using while running a long- running query. If you did this with a Twisted server, it would hang while the query ran.

4. Consider the following argument: The programmers of the world are still using the first major imperative language (Fortran), the first major functional language (Lisp), and the first major logic-programming language (Prolog), but they no longer use the first major object-oriented language (Simula 67). That is because object-oriented languages are fundamentally different. Here's why:

   If you agree with this argument, finish it by adding some justifications and examples. If not, argue the contrary.

   You can argue either way. Some might say the OOP crowd is more trendy. A better reason is that OOP languages like C++ and Java have much more dependence on external libraries, due to the fact that OOP concepts like classes, subclassing, etc, make it very easy to create and maintain external modules. But as time passes, people will begin to write their modules in more modern OOP languages, which causes earlier OOP languages to be unusable due to lack of library support.

5. Consider the static-scoping semantics given in pages 512-513 of Webber. (508-509 in the 2nd edition)

   (a) Can these semantics be implemented easily on a machine in which all storage is allocated on the stack, or do they assume the existence of a heap? If the former, give a nontrivial example and show why the heap is not needed for it; if the latter, give an example of why a heap is necessary.

   Take a look at the rule $(fn(x, E), C) \rightarrow (x, E, C)$
   The static scoping semantics carry the context of the caller into the function body, i.e. all its variable bindings. This assumes existence of a heap, since it would not be easy to store a caller's variables on the new stack frame of a called function.

   (b) This question is the same as (a), except use the dynamic-scoping semantics.

   Take a look at the rule $(fn(x, E), C) \rightarrow (x, E)$
   With dynamic scoping, the callee does not carry the caller's context into the function body, so it would be easier to allocate on the stack, because all the callee's data will live in the callee's stack frame.

6. Generally speaking, languages that use call-by-name or call-by-need parameter-passing conventions avoid hangs and/or crashes that would occur if the same program were evaluated using call-by-value. Can you think of any counterexamples? That is, can you think of a program that has well-defined behavior with call-by-value, but which hangs with call-by-name or call-by-need? If so, give a counterexample; if not, explain why it's not possible to give a counterexample.

   In call-by-value, the parameter is evaluated before the function is executed. In call-by-name, the parameter is evaluated each and every time it is accessed. Call-by-need is a memoized version of call-by-name, where the parameter is evaluated the first time it is accessed its value is cached for future accesses.

   Call-by-name could cause a hang if the parameter was a function that changed some global state. For example,

```
int count = 0; //global state
int bar() {
  count++;
  if (count > 1) {
    while (true) { ... }
  }
  return 1;
}

void foo(int a) {
  printf("%d", a);
  printf("%d", a);
}

foo(bar());
```

In this example, bar will change global state and hang depending on that state. In call-by-value, bar is called only once, updates count to 1, and returns 1. In call-by-name, bar is called each time foo's parameter is accessed, which results in bar being called twice, and hanging on the second call.

This example would not work for call-by-need. Another example, which would be fine for call-by-value but would hang for call-by-name AND call-by-need:

```
bool __is_foo_called = false; //global state
int bar() {
  if (__is_foo_called) {
    while (true) { ... }
  }
  return 1;
}

void foo(int a) {
  __is_foo_called = true;
  print("%d", a);
}

foo(bar());
```

Here, call-by-value would succeed, because bar is evaluated before any part of the body of foo. In call-by-name and call-by-need, foo sets the global state to true before its parameter is accessed. Then, once its parameter is accessed, bar is evaluated and goes into a loop.

7. Write implementations of the following Prolog predicates. Use a clear and simple style. Do not assume that the input arguments are ground terms. You may assume the standard Prolog predicates like member/2 and append/3, but avoid them in favor of unification if you can. Define auxiliary predicates as needed.

   (a) shift_left(L, R) succeeds if R is the result of "shifting left" the list L by 1. The leading element of L is lost. For example, shift_left([a,b,c], [b,c]).

   ```
   shift_left([_|R], R).
   ```

   (b) shift_right(L, R) is similar, except it shifts right. For example, shift_right([a,b,c], [a,b]).

   ```
   shift_right([_], []).
   shift_right([H|L], [H|R]) :- shift_right(L,R).
   ```

   (c) shift_left_circular(L, R) is like shift_left, except the leading element of L is reintroduced at the right. For example, shift_left_circular([a,b,c], [b,c,a]).

   ```
   shift_left_circular([H|L], R) :- append(L, [H], R).
   --or--
   shift_left_circular([H|L], R) :- slc(L, H, R).
   slc([], X, [X]).
   slc([Y|L], H, [Y|R]) :- slc(L, H, R).
   ```

   (d) shift_right_circular(L, R) is similar, except it shifts right. For example, shift_right_circular([a,b,c], [c,a,b]).

   ```
   shift_right_circular(L,R) :- shift_left_circular(R,L).
   ```

   (e) Which of your implementation of the predicates (a)-(d) might be nondeterministic? For each such predicate, give an example call that succeeds more than once.

   There will be nondeterminism with any predicate which has multiple rules, because the unification process can choose either and backtrack to try another. So (b), (c), and (d) will all have nondeterminism, because (b) has multiple rules, append/3 in (c) has multiple rules, and (d) is defined in terms of (c).

8. What is the set of possible behaviors for each of the following Scheme top-level expressions? Assume that each one is typed in to a fresh instance of a Scheme interpreter. Justify any tricky or obscure parts of your answers by citing particular parts of the formal semantics of Scheme.

```
(not (if #f #f))
```

You are not responsible for knowing the two-argument if statement, but FYI it will nondeterministically return #t or #f if the conditional is #f. So this can either evaluate to #f or #t.

```
(car (begin (set! car cdr) (list car cdr)))
```

See Dybvig section 2.3 for comments on order of evaluation of Scheme expressions. To summarize, different implementations may evaluate in different orders, possibly right-to-left or left-to-right. The value of 'car' may be looked up to be the 'car' function before the right expression (begin ...) is evaluated, or the begin expression may be evaluated first.

In the first case (left-to-right), the first 'car' would be evaluated to the 'car' function, and then the 'car' identifier would be set to the 'cdr' function. Then, the list of the 'car' identifier (looked up to be 'cdr') and the 'cdr' function would be created. The 'car' from the beginning would access the first element, which is the 'cdr' function (from the 'car' identifier).

In the second case (right-to-left), the (list...) subexpression would get evaluated first, created a list of the 'car' function and the 'cdr' function. Then the 'car' identifier would get bound to the 'cdr' function, and finally the leftmost 'car' identifier would be looked up to be the 'cdr' function. So (cdr (car cdr)) would evaluate to the 'cdr' function.

Note, however, that it is not even guaranteed that if an expression is evaluated left-to-right, its subexpressions will also be evaluated left-to-right. Directionality may change in subexpressions, so you may get some combination of the above behavior.

9. The abstract syntax for Scheme in R5RS section 7.2.1 lists only constants, identifiers, procedure calls, lambda, if, and set!. The following expressions aren't any of these things, so why are they valid expressions in Scheme? Or if they're not valid expressions, say why not.

```
(let ((a '())) a)
(cond ((eq? '() '()) 1) (#t 0))
```

The above two are macros that are replaced with the abstract syntax defined in R5RS.

(1) - This is not valid because 1 is not a function.

10. Consider the following Java code snippet, taken from BioJava. Translate it as best you can into idiomatic Python. For parts that you cannot easily translate, say what the problems are, and how you'd go about addressing these problems if you actually had to translate all of BioJava into Python.

```
package org.biojava.bio.program.ensembl;
import java.util.*;
import java.lang.ref.*;
import org.biojava.bio.*;
...
class CloneDB extends ImmutableSequenceDBBase
       implements SequenceDB {
  private Ensembl ensembl;

  private Map cloneCache;
  private Set ids_cache;

  { cloneCache = new HashMap(); }

  public CloneDB(Ensembl db) {
      this.ensembl = db;
  }

  public String getName() {
      return "http://www.ensembl.org/";
  }

  Sequence getCloneSequence(Clone cont)
      throws EnsemblException
  {
    Sequence seq = (Sequence)
      cloneCache.get(cont.getID());
    if (seq == null) {
      if (ensembl.getUseHeavyClones()) {
        seq = new EnsemblHeavyCloneSequence
          (cont, ensembl);
      } else {
        seq = new EnsemblCloneSequence
          (cont, ensembl);
      }
      cloneCache.put(cont.getID(), seq);
    }
    return seq;
  }
  ...
}
```

```
from java.util import *
from java.lang.ref import *
from org.biojava.bio import *

class CloneDB(ImmutableSequenceDBBase):
  def __init__(self, db):
    self.ensembl = db
    self.cloneCache = HashMap()

  def getName(self):
    return "http://www.ensembl.org/"

  def getCloneSequence(self, cont):
    seq = self.cloneCache.get(cont.getID())
    if seq == None:
      if self.ensembl.getUseHeavyClones():
        seq = EnsemblHeavyCloneSequence(cont, self.ensembl)
      else:
        seq = EnsemblCloneSequence(cont, self.ensembl)
      self.cloneCache.put(cont.getID(), seq)
    return seq
```

There are many issues.

- There is not an equivalent to the package keyword. To achieve the same package semantics, you have to use directory-local __init__.py files as described here: https://docs.python.org/2/tutorial/modules.html#packages
- There are no interfaces in Python, so you have to assume or manually check that your CloneDB class is implementing the correct functions.
- Python does not really have a concept of private/public fields and methods. There is a trick you can do, detailed here: http://stackoverflow.com/questions/70528
- Type information cannot be translated to Python because Python is dynamically typed.
- The { `cloneCache = new HashMap();` } line is an "initializer block", which is not really an issue, because in Java, the line is simply copied into every constructor by the compiler. It simply allows you to share large code blocks between different constructors. To translate to Python, just stick it in the __init__ method.
- Python has no exception annotations, but you throw and catch similar to Java, so the "throws EnsemblException" is just left off.
- You would have to assume that the Java libraries imported have also been ported to Python.

11. For each of the following seemingly-arbitrary language rules, explain what performance gains (if any) arise because of the rules. Analyze both time and space performance. List the rules roughly in decreasing order of importance, performance-wise.

    (a) Arrays are origin-zero, not origin 1.

    Slightly faster. For origin-zero, to get the address of $a[4]$, you will calculate $a + 4 \times N$, where N is the size of the array cell. For origin-1, to get the address of $a[5]$, you calculate $a + (5 - 1) \times N$, which is extra arithmetic.

    (b) Arrays must be allocated statically or on the stack; they cannot be allocated on the heap.

    This allows you to do away with any runtime memory manager (e.g. garbage collector), because all your memory will be cleaned up when a stack frame is popped off.

    (c) Each object in an array must have the same size.

    Faster, because you can do arithmetic indexing. However, this may waste space, because you have to make the size the maximum of whatever objects you want to put in.

    (d) Subscript violations result in undefined behavior; an exception may or may not be raised.

    Performance gain because you don't have any bounds-checking overhead. You also get a small space gain because you don't have to store the array length anywhere.

    (e) The array must have just one dimension.

    Performance loss due to the extra index calculation of both dimensions. Also the possibility of external fragmentation preventing the allocation of a large array (which would otherwise be smaller chunks given a C++ or Java way of doing multidimensional arrays).

    (f) The number of elements in an array is fixed at compile-time.

    If you're using stack allocation of arrays, fixed size makes this faster. However, you may have to allocate a large array according to the max size it would become if it was dynamically allocated.

    (g) Arrays must always be accessed directly; there are no pointers to arrays, and arrays cannot be passed by reference to procedures.

    Very space-wasting, because you must copy arrays when passing them into functions. However, this avoids aliasing, which allows for better compiler optimizations, including constant propagation and code reordering.

12. Suppose we add a new statement "return[N]EXPR;" to our C implementation, as an extension to C. N must be an integer constant expression with a nonnegative value, and EXPR can be any valid C expression. This new statement acts like "return EXPR;", except that it causes the Nth caller of the current function to return with the value of the expression EXPR. For example, suppose F calls G which calls H, and suppose H then executes "return[2]5;". This causes F to return immediately, with the value 5; G and H are silently exited in the process. By definition, "return[0]EXPR;" is equivalent to "return EXPR;".

What problems do you see in implementing "return[N]EXPR;"? Are these problems inherent to C, or can they be worked around by changing the semantics of the new statement slightly? Would it be easier to implement this new statement in some of the other languages that we have studied? If so, which ones and why? If not, why not?

A huge problem with this is centered around static typing. Because some function F does not know how deep in the call chain its return value will come from, it cannot know the type of that return value. Imagine the functions F, G, and H in the problem have return types void, int, and double, respectively. H executing "return[2]1.0" would cause F to try to return 1.0. These problems are inherent to C.

This would be much easier to implement in Python, for example, because of dynamic typing. The return type of the function is not specified, so returning some alternative value would not be a problem.