Sample answers:

1. Explain what the functional programming paradigm is, and cite one or more rules or conventions that distinguish it from traditional procedural programming.

   This definition is rather long
   The essence is in the first paragraph. The key notions are that the program consists of evaluating functions starting at the top level, and avoiding side effects that change data values.

   Here’s a more concise answer:
   - A program consists of a hierarchy of functions.
   - Execution consists of executing (evaluating) the top-level function, which may then invoke other functions.
   NOTE: Since the two characteristics above are common to other paradigms, including ordinary procedural, you also needed to say something about the characteristics below:
   - Results consist (mainly or even exclusively) of the returned function value.
   - Wherever possible:
     - functions are pure; i.e. they have no side effects. The code is read-only,
     - to obtain a succession of values, recursion is preferred to iteration

2. Explain briefly how forward-chaining inference works, and cite an example of its application

   We start with predicates that we know to be true. We’re especially interested in implications of the form \( P \rightarrow Q \) (or a logical equivalent such as \( Q \mid \neg P \)). Once we know that \( P \) is true, if necessary after substituting values of variables, we know that \( Q \) is true, too. This rule is called *modus ponens*.

   Then we look for an implication of the form \( Q \rightarrow R \), where \( Q \) is the same as before. We continue until we
   - (a) produce the desired query conclusion, or
   - (b) produce the negation of the desired conclusion, or
   - (c) produce a complete list of all possible conclusions (possibly none).

   Examples of its application include:
   - MYCIN the prototype experimental medical diagnosis system.

3. Explain the differences among: functions, macros, and special forms. Why do Lisp and Clojure support all three?
- **Special forms** are lower-level facilities that provide access to the environment that couldn’t be done with a Clojure function. The comparatively small repertoire of special forms, not coded in Clojure, itself, allowed Clojure’s developers to “bootstrap” the implementation.

- **Functions** are the basic unit of programs. Execution consists of evaluating the top-level function. Clojure compiles functions into executable Java bytecode.

- **Macros** are higher-level constructs that cater to ease-of-use, naturalness, and readability. They can use (generate invocations of) functions and other macros, but they cannot access values of data items. Macros help to keep the core language simple and easy to implement. They sometimes also help the programmer to define special versions of the language suited to specific applications.

- LISP and Clojure rely on all three levels in order to keep the core language simple to learn and easy to implement.

4. Explain the essential conceptual difference between:
   
   (a). **if** statements (or other conditionals) in a procedural programming language, such as C or BASIC.
   
   When the condition is true, the computer executes the imperative statements in the **then** clause. When the condition is false, it executes the statements in the **else** clause, if any. In either case some result is produced.

   (b). **if** forms (or other conditionals) in a functional programming language such as LISP or Clojure.
   
   When the condition is true, the computer assumes the **then** (usually true-false or Boolean) expression to be true. When the condition is false, it assumes the **else** expression to be true. In either case the result is either **true** or **false** (in Lisp or Clojure, it's either **true or nil**).


   ```java
   public int Sum (int num)
   {int result;
    if (num == 1)
     result = 1;
    else  result = num + Sum(num-1);
    return result;
   }
   ```

   Never fear. The author concedes two pages later that the recursive version is extremely inefficient. He then proposes a more efficient **iterative** version:

   ```java
   public int Sum (int num)
   {int  result = 0;
    for (int number = 1; number <= num; number++)
     result += number;
    return result;}
   ```
Some students wondered what this question has to do with artificial intelligence. Version 1 is an example of misuse of recursion, which is a powerful and useful technique in AI, but shouldn’t be assumed when simpler, more direct, or more efficient techniques are appropriate. Version 2 reminds us that even iterative solutions can be wasteful and unnecessary. (Surprising in a book from a respected author and publisher!)

(a). Will both versions produce the correct answer for any specified \textit{num}? If not, what failure(s) might occur?

If \textit{num} is negative, the first version will invoke itself recursively until memory is exhausted, since the base case will never occur. The second version will yield 0 for negative \textit{num}, since the loop won’t be executed.

(b). Would the second (iterative) version be acceptable in either an organization’s program module library or in a production application? If not, why?

It is an extremely costly and inefficient method that would be inappropriate in any program. See the pink sidebar at http://www.idinews.com/factorial.html

Also there’s no commentary.

Afterthought: A surprising number of students didn’t recognize the obvious closed-form solution! They got full credit, but that was elementary algebra, which any serious computer-science student should have mastered long ago. A working programmer (and that book’s author) should be embarrassed by not noticing this.

6. Two formulas (logical expressions) contain both constant terms and variable terms.

(a). What purpose would be served by a query interpreter trying to \textit{unify} them?

Determining values, if any, for which both formulas are true, either yielding answers to a query or allowing computation to continue using those values. If no such values can be found, unification fails and the query yields false.

(b) Where would the query interpreter find/keep/store additional information that it needs about the problem domain?

A rule base consisting of inference rules (conditionals).
A data base consisting of assertions of facts.

(c). How should such a program present/convey the results of the query? That is, what form would the \textit{results} of unification usually take?

By a list of bindings, i.e. assignments of values to the variables in the predicates, so that the predicates then imply the desired conclusion(s).

7. Cite some reasons why today’s programmers developing a large application that includes forward-chaining inference logic are likely to prefer using Clojure to doing the equivalent
things in Lisp. In other words, cite some positive features of Clojure or negative features of Lisp that aren’t shared by the other language.

Here are three:

(a) By eliminating some of the parentheses, Clojure code is easier to read (and debug) than Lisp code.

(b) By being implemented on top of Java, a widely (almost universally) supported object-oriented language, Clojure generates reasonably efficient code and allows the programmer to use Java for parts of a program that need operations Clojure doesn’t directly support, such as input-output or user interfaces.

(c) Lisp has evolved into multiple incompatible versions that may confuse programmers or introduce subtle bugs. “Common LISP” is an attempt to correct that, but other versions persist and are favored by some users.

8. Given the following Clojure predicates about human or animal relationships:
   Male(x)   :- x is male
   Female(x)  :- x is female
   Child(y,x) :- x is a child of y (or y is a parent of x)

Write logical expressions (use conventional symbolic notation or Clojure expressions or clear unambiguous English) in terms of those primitives for:

We’ll use symbolic logic notation in the sample answers below. Clojure equivalents should be obvious.

Note: I didn’t worry about those existential quantifiers. If you coded an unbound variable, I assumed it was appropriately scoped.

(a) x is a grandmother of y.
   \[ \text{grandmother}(x,y) \leftrightarrow \text{Female}(x) \land \exists z (\text{Child}(x,z) \land \text{Child}(z,y)) \]

(b) x is the brother of y
   \[ \text{brother}(x,y) \leftrightarrow \text{Male}(x) \land x \neq y \land \exists z (\text{Child}(z,x) \land \text{Child}(z,y)) \]

(c) x and y are first cousins
   Various ways of expressing this call for definition of more auxiliary predicates. Here’s one that doesn’t:
   \[ \text{cousins}(x,y) \leftrightarrow \exists u \exists v (u \neq v \land \text{Child}(u,x) \land \text{Child}(v,y) \land \text{Child}(z,u) \land \text{Child}(z,v)) \]