1a Suppose we are writing a Scala program to find the first element of a list that satisfies a given predicate. Because the list might contain no elements that satisfy that predicate, we must use the Option data type. Fill in the blanks in the function definition:

```scala
def firstSuchThat[A](l: List[A], p: A => Boolean): Option[A] = l match {
  case Nil => None
  case x :: xs => {
    if (p(x)) Some(x)
    else firstSuchThat(xs, p)
  }
}
```

1b Suppose you wanted to transmute the definition above into a function streamFirstSuchThat whose first argument would be a Stream[A]. Obviously, you need to replace all occurrences of firstSuchThat with streamFirstSuchThat, and you need to replace List[A] with Stream[A]. There are two other changes that are necessary. What are they?

i. Replace Nil with Stream.Empty

ii. Replace `::` with `#::`.

iii. Prove that if a stream is infinite then streamFirstSuchThat can never return None.

We’ll prove that if streamFirstSuchThat(p, l) does return None, then l is finite. The only way it can return None is if the first clause of the match matches, which means that for some recursive call n deep, Stream.Empty is encountered. But the recursive calls are all through the second clause of the match, on the tails of l. That in turn implies that l.tail^n = Stream.Empty. This is the definition of “finite.” QED

Getting this right proved difficult for many students. Some tried proving it by induction. But induction is a tool for proving a recursive program’s result has a certain property, assuming one of its arguments was constructed by applying some
“constructor operations” a finite number of times.¹ That’s exactly what isn’t true here.

We invoked a lemma above that if a recursive function returns x, then x must have been the value of some recursive call n deep for some finite number n. Might this be proved by induction? But induction on what? Not on n: an induction on n assumes n is a natural number, i.e., either 0 or the successor of a natural number; but that’s what we want to prove.²

² Which variables occur bound and free in the following formulas? Remember that the same variable may occur both bound and free in the same formula. The only variables here are p, x, y, and z; the other lower-case letters, a, b, etc., are constants.

<table>
<thead>
<tr>
<th>Formula</th>
<th>Bound</th>
<th>Free</th>
</tr>
</thead>
<tbody>
<tr>
<td>(a) ( \forall x (P(x,y) \land \exists y (Q(x,y))) )</td>
<td>x, y</td>
<td>y</td>
</tr>
<tr>
<td>(b) ( P(0,x) \lor P(x,b) )</td>
<td>None</td>
<td>x</td>
</tr>
<tr>
<td>(c) ( \forall p (p(x)(a) \land \text{social_relation}(p) \oplus p=\lambda x.y.(\text{love}(x,y) \land \neg \text{love}(y,x))) )</td>
<td>p, x, y</td>
<td>x</td>
</tr>
</tbody>
</table>

¹ In more complex cases, you must put some or all of its arguments together into a tuple that can be shown to be finitely constructible.

² So how does a proof of something so obvious go? Perhaps like this: Let F be the recursive function in question. We’ll assume it’s directly recursive in that it calls itself, rather than calling something that calls it; because streamFirstSuchThat is directly recursive. We’ll assign a number \( n(c) \) to every call \( c \) of \( F \), where \( n \) may be defined as “the depth of the recursive call at which the value of \( c \) is computed.” Whatever a recursive function returns must either be something requiring no computation (a constant or one of its parameters, for example); or be the value of the last thing it calls. Let \( c_0 \) be a call to \( F \). The last thing it calls is either itself or some other function \( G \), that by hypothesis results in no further calls to \( F \). The chain of calls to itself cannot go on forever, or no value would be returned. So assume it stops. Have I proved it yet? No? Then the value of \( n(c_l) \) for the last call \( c_l \) is 0. The call that called the last call gets value \( n(\text{caller}(c_l)) = 1 \). And so forth, until we reach the original call \( c_0 = \text{caller}^{n(c_0)}(c_l) \). So the \( n \) in the main proof is \( n(c_0) \). I don’t think any questions are begged in this proof, although several are raised; formalizing it completely would be challenging.
3. Connect left-column item(s) to equivalent right-column item(s)

<table>
<thead>
<tr>
<th>Statements about theorems</th>
<th>Statements about models</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 “The theory has no theorems”</td>
<td></td>
</tr>
<tr>
<td>2 “Every formula is a theorem of the theory”</td>
<td>A “The theory has no models”</td>
</tr>
<tr>
<td>3 “The theory is inconsistent”</td>
<td>B “Every interpretation is a model of the theory”</td>
</tr>
<tr>
<td>4 “The theory is vacuous”</td>
<td></td>
</tr>
</tbody>
</table>

Solution:

1-B, 2-A, 3-A, 4-B

(using notation “number-letter”).

Explanation: Every formula is a theorem in an inconsistent theory because from \( p \) and \( \neg p \) it is hard to keep \( q \) from following. Every model of an inconsistent theory would have to make both \( p \) and \( \neg p \) (alternatively, every formula) true; but no model can make \( p \) and \( \neg p \) true (the value assigned to \( \neg p \) is by definition the opposite of the value assigned to \( p \)). So there are no models of inconsistent theories.

At the opposite extreme are vacuous theories with no theorems at all, except tautologies, which are “vacuously” true. (An example of a tautology, in a theory including predicates \( \text{red} \) and \( \text{loves} \), is “Everything is red and loves something, or is not red, or loves nothing.”) Every interpretation makes tautologies true, so every interpretation is a model of a vacuous theory.

Terminology reference: A theory is a set of formulas. An interpretation explains how symbols map to function and predicate constants. An interpretation is a model of a theory if and only if it makes all the theorems of the theory true.