The goal of this section is to expose you to logic programming and type inference. We’ll start by looking at prime numbers. Complete the following program:

**Answer:**

```prolog
divisible(X, Y) :- Z is X / Y, integer(Z).
composite(X, Y) :- Y > 1, divisible(X, Y).
composite(X, Y) :- Y < (X / 2), composite(X, Y + 1).
prime(X) :- X > 1, not(composite(X, 2)).
```

Now let’s take a closer look at the definition of environment lists from lecture. The ‘typeof’ predicate checks if the identifier I has type T in a given environment:

```prolog
defn(I, T, [def(I, T) | _]).
defn(I, T, [def(I0, _) | R]) :- dif(I, I0), defn(I, T, R).
typeof(X, T, Env) :- defn(X, T, Env).
```

Translate these lines of Prolog into English in your own words. Be precise.

**Answer:** I is defined to have type T if the environment list starts with such a definition, or if I isn’t the same as the identifier in the first ‘def’ but matches the next suitable definition further down the list.

For this problem, we’ll come up with a few typing rules for the following language. Then we’ll experiment with type inference.

```prolog
e : ID | INT | ‘[‘ ((e ‘,’)* e)? ‘]’ | ‘lambda’ ‘(‘ ID ‘,’ e ‘)’ | e ‘+’ e
```

The type of an ID is dependent on the current environment, and INT expressions type the same way regardless of the environment. We can express these rules with:

```prolog
typeof(X, int, _) :- integer(X).
typeof(X, T, Env) :- defn(X, T, Env).
```

Write typing rules that enforce the following constraints:
1. The type of an empty list is unbounded (e.g. \([T]\)). \textbf{Answer:}

\[
\text{typeof([], [T], \_)}.
\]

2. The type of a list is \([T]\), where all list elements are of type \(T\). \textbf{Answer:}

\[
\text{typeof([E | R], [T], Env) :- typeof(E, T, Env), typeof(R, [T], Env).}
\]

3. The type of a lambda is \(T_1\rightarrow T_2\), where \(T_2\) is the return type and \(T_1\) is the type \(X\) is bound to within the body of the lambda. \textbf{Answer:}

\[
\text{typeof(lambda(X, E), T1\rightarrow T2, Env) :- typeof(E, T2, [def(X, T1) | Env]).}
\]

4. The type of \(L + R\) is \(\text{int}\) when \(L\) and \(R\) are both of type \(\text{int}\). \textbf{Answer:}

\[
\text{typeof(L + R, \text{int}, Env) :- typeof(L, \text{int}, Env), typeof(R, \text{int}, Env).}
\]

5. The type of \(L + R\) is \([T]\) when \(L\) and \(R\) are both of type \([T]\). \textbf{Answer:}

\[
\text{typeof(L + R, [T], Env) :- typeof(L, [T], Env), typeof(R, [T], Env).}
\]

6. The type of \(L + R\) is \(T_2\) when \(L\) is of type \(T_1\rightarrow T_2\) and \(R\) is of type \(T_1\). \textbf{Answer:}

\[
\text{typeof(L + R, T2, Env) :- typeof(R, T1, Env), typeof(L, T1\rightarrow T2, Env).}
\]

At this point we could enter our rules into Prolog and have it infer types for our programs. For the next few questions, try solving for \(T\) manually:

1. \text{typeof(lambda(x, x + x) + 1, T, []). Answer: } T = \text{int}.

2. \text{typeof(lambda(x, x + x) + [1], T, []). Answer: } T = [\text{int}].

3. \text{typeof(f + g, T, [def(f, int\rightarrow [\text{int}]), def(g, int)]). Answer: } T = [\text{int}].

4. \text{typeof(lambda(x, x + x) + [lambda(x, x + x)], T, []). Answer: } T = [(T_1\rightarrow T_2)].

Write a list reversal predicate in Prolog. Hint: Use a helper predicate. \textbf{Answer:}

\[
\begin{align*}
\text{lrevaux([], A, A).} \\
\text{lrevaux([X | Y], A, R) :- lrevaux(Y, [X | A], R).} \\
\text{lrev(X, R) :- lrevaux(X, [], R).}
\end{align*}
\]