Questions are based on *Stochastic Lambda Calculus and Monads of Probability Distributions* by Norman Ramsey and Avi Pfeffer.

**Comparing designs**

If \( P \) is a probability monad, then the Haskell code presented in this paper offers these functions:

- `return :: a -> P a`
- `(>>=) :: P a -> (a -> P b) -> P b` -- pronounced "bind"
- `choose :: Probability -> P a -> P a -> P a`
- `support :: P a -> [P a]`
- `expectation :: (a -> Double) -> P a -> Double`
- `sample :: RandomGen g => P a -> g -> (a, g)`

Our own design for finite probability distributions \( F \) offers these functions:

- `certainly :: a -> F a`
- `equally :: [a] -> F a`
- `weightedly :: Real w => [(w, a)] -> F a`
- `fchoose' :: Probability -> F a -> F a -> F a`
- `fchoose :: Probability -> a -> a -> F a`
- `pmap :: (a -> b) -> F a -> F b`
- `pfilter :: (a -> Bool) -> F a -> F a`
- `pfilterMap :: (a -> Maybe b) -> F a -> F b`
- `bindx :: F a -> (a -> F b) -> F (a, b)`
- `liftPair :: F a -> F b -> F (a, b)`
- `prob :: (a -> Bool) -> F a -> Probability`
- `vprob :: Eq a => F a -> a -> Probability`
- `expected :: (a -> Double) -> F a -> Double`
- `fsupport :: F a -> [a]`
- `byLikelihood :: Ord a => F a -> [(Probability, a)]`

1. What functions are *identical* in both designs (except for possibly having different names)?
2. What functions in the paper’s design are missing from our design but can be simulated using our functions? When simulations exist, *please show them*.
3. What functions in our design are missing from the paper’s design but can be simulated using the paper’s functions? Just sketch one or two simulations.
4. What functions in the paper’s design, if any, *can’t* be simulated using our functions?
5. What functions in our design, if any, *can’t* be simulated using the paper’s functions?
6. Our design can be used to solve all the dice problems. The paper’s design can’t. What is crucial piece or pieces are missing from the paper’s design that prevent it from being able to do all the dice problems?

**Deeper analysis of the paper**

7. Explain Figure 2. (Think of Figure 2 as the specification for an interpreter; in particular, think of \( P \) as a recursive function that might be called `eval`. You might begin by asking about that function’s type.)
8. In the first paragraph of Section 8 on page 163, what’s being claimed? Are you convinced?
9. Section 5.1 talks about the performance of expectation queries over product spaces. You have intimate experience with product spaces from the dice world. Can you think of an expectation query that would take \( O(|X| \times |Y|) \) time in the expectation monad but could be computed in \( O(|X| + |Y|) \) time using some other technique?

**Questions to take home**

10. Unless you happen to know the related work already, sections about related work are usually boring. But there’s a different reading of Section 7 than simply who did what. Looking through Section 7, what different language-design choices can you identify? What choices appeal to you? What choices do you find distasteful?
11. Section 3.2 mentions sampling functions. For a distribution with finite support, what would a sampling function look like?

12. What role should abstract integration play in a probabilistic programming language?

13. What, if any, is the computational consequence of equation (2) at the bottom of page 156?

14. At the end of Section 4 on page 158, the paper says that in the paper’s calculus, it is not safe to duplicate a redex. Why not? Is there a simple explanation in terms of what kind of language design is being described?