Monad functions are described for the Maybe monad as follows,

```
    (i)
        return :: a -> Maybe a
        return x = Just x
    (ii)
        (>>=) :: Maybe a -> (a -> Maybe b) -> Maybe b
        m >>= f = case m of
                        Just x  -> f x
                        Nothing -> Nothing
```

1. Left identity: (return a >>= f) = (f a)

```
Proof:
    LHS = return a >>= f
        = Just a >>= f      {by definition of return}
        = f a               {by definition of >>=}
        = RHS
    Hence proved.
```

2. Right identity: (m >>= return) = (m)

```
Proof:
    (i) m = Nothing,
        LHS = m >>= return
            = Nothing >>= return
            = Nothing              {by definition of >>=}
            = m
            = RHS
    (ii) m = Just a,
        LHS = m >>= return
            = Just a >>= return
            = return a             {by definition of >>=}
            = Just a               {by definition of return}
            = m
            = RHS
    Hence proved.
```

3. Associativity: ((m >>= f) >>= g) = (m >>= (\x -> f x >>= g))

```
Proof:
    (i) m = Nothing,
        LHS = (m >>= f) >>= g
            = (Nothing >>= f) >>= g
            = Nothing >>= g                    {by definition of >>=}
            = Nothing                          {by definition of >>=}
        RHS = m >>= (\x -> f x >>= g)
            = Nothing >>= (\x -> f x >>= g)
            = Nothing                          {by definition of >>=}
        LHS = RHS
    (ii) m = Just a,
        LHS = (m >>= f) >>= g
            = (Just a >>= f) >>= g
            = (f a) >>= g                      {by definition of >>=}
        RHS = m >>= (\x -> f x >>= g)
            = Just a >>= (\x -> f x >>= g)
            = (\x -> f x >>= g) a              {by definition of >>=}
            = (f a) >>= g                      {by beta reduction}
        LHS = RHS
    Hence proved.
```

Monad functions are described for the List monad as follows,

```
    (i)
        return :: a -> [a]
        return x = [x]
    (ii)
        (>>=) :: [a] -> (a -> [b]) -> [b]
        m >>= f = concat (fmap f m)
```

We will be using the following well known identities,

```
(1) (concat a) ++ (concat b) = concat (a ++ b)
(2) fmap f (x ++ y) = (fmap f x) ++ (fmap f y)
```

Now we will prove the 3 given monad laws.

```
1. Left identity: (return a >>= f) = (f a)
Proof:
    LHS = return a >>= f
        = [a] >>= f                     {by definition of return}
        = concat (fmap f [a])           {by definition of >>=}
        = concat [f a]                  {by definition of fmap}
        = f a                           {by definition of concat}
        = RHS
    Hence proved.

2. Right identity: (m >>= return) = (m)
Proof: Induction on length of list m,
      Base case: m = []
      Inductive step: m = (x : xs),
        LHS = m >>= return
            = (x : xs) >>= return
            = concat (fmap return (x : xs))                 {by definition of >>=}
            = concat (return x : fmap return xs)            {by definition of fmap}
            = (return x) ++ (concat (fmap return xs))       {by definition of
concat}
            = (return x) ++ (xs >>= return)                 {by definition of >>=}
            = (return x) ++ xs                              {by inductive
hypothesis}
            = [x] ++ xs                                     {by definition of
return}
            = (x : xs)                                      {by definition of ++}
            = m
            = RHS
    Hence proved.

3. Associativity: ((m >>= f) >>= g) = (m >>= (\x -> f x >>= g))
Proof: Induction on length of list m,
    (i) Base case: m = [],
    (ii) Inductive step: m = (x : xs),
```

```
        LHS = (m >>= f) >>= g
            = ((x : xs) >>= f) >>= g
            = (concat (fmap f (x : xs))) >>= g                                {by
definition of >>=}
            = (concat (f x : fmap f xs)) >>= g                                {by
definition of fmap}
            = ((f x) ++ (concat (fmap f xs))) >>= g                           {by
definition of concat}
            = concat (fmap g ((f x) ++ (concat (fmap f xs))))                 {by
definition of >>=}
            = concat ((fmap g (f x)) ++ (fmap g (concat (fmap f xs))))        {by
identity (2)}
            = (concat (fmap g (f x))) ++ (concat (fmap g (concat (fmap f xs)))) {by
identity (1)}
            = (concat (fmap g (f x))) ++ ((concat (fmap f xs)) >>= g)         {by
definition of >>=}
            = (concat (fmap g (f x))) ++ ((xs >>= f) >>= g)                   {by
definition of >>=}
            = (concat (fmap g (f x))) ++ (xs >>= (\x -> f x >>= g))           {by
inductive hypothesis}
            = (concat (fmap g (f x))) ++ (concat (fmap (\x -> f x >>= g) xs))   {by
definition of >>=}
            = concat ((fmap g (f x)) ++ (fmap (\x -> f x >>= g) xs))          {by
identity (1)}
        RHS = m >>= (\x -> f x >>= g)
            = (x : xs) >>= (\x -> f x >>= g)
            = concat (fmap (\x -> f x >>= g) (x : xs))                        {by
definition of >>=}
            = concat (((\x -> f x >>= g) x) : (fmap (\x -> f x >>= g) xs))     {by
definition of fmap}
            = concat ((f x >>= g) : (fmap (\x -> f x >>= g) xs))              {by
beta reduction}
            = concat ((concat (fmap g (f x))) : (fmap (\x -> f x >>= g) xs))   {by
definition of >>=}
            = (concat (fmap g (f x))) ++ (concat (fmap (\x -> f x >>= g) xs))   {by
definition of concat}
            = concat ((fmap g (f x)) ++ (fmap (\x -> f x >>= g) xs))          {by
identity (1)}
        LHS = RHS
    Hence proved.
```