– Pregunta 1:

```
type Conjunto a = a -> Bool

– a
miembro :: Conjunto a -> a -> Bool
miembro conjunto elemento = conjunto elemento

– b
vacio :: Conjunto a
vacio = \_ -> False

– c
singleton :: (Eq a) => a -> Conjunto a
singleton x = \y -> x == y

– d
desdeLista :: (Eq a) => [a] -> Conjunto a
desdeLista lista = \x -> x elem lista

– e
complemento :: Conjunto a -> Conjunto a
complemento conjunto = \x -> not (conjunto x)

– f
union :: Conjunto a -> Conjunto a -> Conjunto a
union conjunto1 conjunto2 = \x -> conjunto1 x || conjunto2 x

– g
interseccion :: Conjunto a -> Conjunto a -> Conjunto a
interseccion conjunto1 conjunto2 = \x -> conjunto1 x && conjunto2 x

– h
diferencia :: Conjunto a -> Conjunto a -> Conjunto a
diferencia conjunto1 conjunto2 = \x -> conjunto1 x && not (conjunto2 x)

– i
transformar :: (b -> a) -> Conjunto a -> Conjunto b
transformar f conjuntoA = \y -> conjuntoA (f y)
```

Pregunta 2.a:

```
Vacio :: ArbolMB a
RamaB :: a -> ArbolMB a -> ArbolMB a -> ArbolMB a
```

– Pregunta 2.b:

```
transformarVacio :: b
transformarRamaB :: a -> b -> b -> b
```

– Pregunta 2.c:

```
plegarArbolMB :: b -- El tipo de transformarVacio
            -> (a -> b -> b) -- El tipo de transformarRamaM.
            -> (a -> b -> b -> b) -- El tipo de transformarRamaB.
            -> ArbolMB a -- El arbol a plegar.
            -> b -- El resultado del plegado.


plegarArbolMB transVacio transRamaM transRamaB = plegar
  where
    plegar Vacio = transVacio
    plegar (RamaM x y) = transRamaM x (plegar y)
    plegar (RamaB x y z) = transRamaB x (plegar y) (plegar z)
```

Pregunta 2.d:

```
sumarArbolMB :: (Num a) => ArbolMB a -> a
sumarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = 0
    transRamaM a b = a + sumarArbolMB  b
    transRamaB a b c = a + sumarArbolMB  b + sumarArbolMB c
```

Pregunta 2.e:

```
aplanarArbolMB :: ArbolMB a -> [a]
aplanarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = []
    transRamaM a b = aplanarArbolMB b ++ [a]
    transRamaB a b c = aplanarArbolMB b ++ [a] ++ aplanarArbolMB c
```

Pregunta 2.f:

```
analizarArbolMB :: (Ord a) => ArbolMB a -> Maybe (a, a, Bool)
analizarArbolMB = plegarArbolMB transVacio transRamaM transRamaB
  where
    transVacio = Nothing  -- En caso de un árbol vacío, devolvemos Nothing.
```

transRamaM x (minVal, maxVal, isSorted) = Just (min x minVal, max x maxVal, isSorted && x >= minVal)
transRamaB x (minIzq, maxIzq, isSortedIzq) (minDer, maxDer, isSortedDer) = Just (minimum [x, minIzq, minDer], maximum [x, maxIzq, maxDer], isSortedIzq && isSortedDer && x >= maxIzq && x <= minDer)

Pregunta 2.e:

La función plegarGen debería tomar n+1 funciones de transformación como argumentos, además del valor de tipo Gen a que se desea plegar. Cada función de transformación corresponde a un constructor diferente del tipo Gen a.

Pregunta 2.f:

La función predefinida sobre listas en Haskell que tiene una firma y un comportamiento equivalente al de implementar una función de plegado es la función foldr.

-- Pregunta 2.g :

-- La función PlegarGen debe tomar n + 1 funciones como argumento.

-- La razón es que la función PlegarGen debe tener un caso para cada constructor del tipo de datos Gen a.
-- Cada caso debe llamar a una función diferente para transformar el valor del constructor en el tipo de datos b que se desea obtener.

-- Pregunta 2.h:

-- La función predefinida foldr del Preludio de Haskell tiene una firma y un comportamiento equivalente al de implementar una función de plegado para el tipo [a]:
foldr :: (a -> b -> b) -> b -> [a] -> b

– Pregunta 3.a:
Se tomó (Secuencial s) como la instancia para el monad y no simplemente Secuencial porque el tipo (Secuencial s) tiene un argumento, el tipo del estado s. Por lo tanto, es necesario que la instancia del monad también tenga un argumento, que en este caso es el tipo del estado.

– Pregunta 3.b:

return :: a -> Secuencial s a
(>>=) :: Secuencial s a -> (a -> Secuencial s b) -> Secuencial s b
>> :: Secuencial s a -> Secuencial s b -> Secuencial s b
fail :: a -> Secuencial s a

– Pregunta 3.c:

```
return a = Secuencial $ \estado -> (a, estado)
```

– Pregunta 3.d:

```
(Secuencial programa) >>= transformador =
        Secuencial $ \estadoInicial ->
        let (resultado, nuevoEstado) = programa estadoInicial
            (Secuencial nuevoPrograma) = transformador resultado
        in nuevoPrograma nuevoEstado
```

– Pregunta 5.a:

```
subs (id const) const id -> (id const) id (const id)
                            const id (const id)
                                const id id
                Resultado:          id
```