```
import Text.Show.Functions
```

-- PUNTO 1

```
data Alumno = Alumno {
  nombreAlumno :: String,
  dedicación :: Int,
  materiasAprobadas :: [Materia],
  conceptosAprendidos :: [String]
} deriving Show

-- Ejemplo:
pepe = Alumno "Pepe" 0 [] []
```

-- PUNTO 2.a

```
-- Aclaración: Para el punto 6 este tipo cambia, así que esta definición habría que borrarla en un archivo de código real

data Materia = Materia {
  nombreMateria :: String,
  efecto :: EfectoEnAlumno
} deriving Show

type EfectoEnAlumno = Alumno -> Alumno

-- Auxiliares

aumentarDedicación tiempoDedicado alumno = alumno {
  dedicación = tiempoDedicado + dedicación alumno
}

olvidar concepto alumno = alumno {
  conceptosAprendidos = (filter ((/=) concepto) . conceptosAprendidos) alumno
}

aprenderConceptos conceptos alumno = alumno {
  conceptosAprendidos = conceptos ++ conceptosAprendidos alumno
}

conceptosRecursivos = map ((++) "Recursividad " . show) [1..]

agregarPrefijo prefijo alumno = alumno {
  nombreAlumno = prefijo ++ nombreAlumno alumno
}

-- Principales
-- Para el punto 6, las materias también son otras así que habría que borrar lo que está acá:
paradigmas = Materia {
  nombreMateria = "Paradigmas",
```

```
  efecto = aumentarDedicación 100 . aprenderConceptos ["polimorfismo", "orden superior"]
}

sistemasOperativos reentregas = Materia {
  nombreMateria = "Sistemas Operativos",
  efecto = agregarPrefijo "Excelentísim@" . aumentarDedicación (1000 * reentregas)
}

recursividadAFull = Materia {
  nombreMateria = "Recursividad a Full",
  efecto = aprenderConceptos conceptosRecursivos
}

desastre2 = Materia {
  nombreMateria = "Desastre 2",
  efecto = olvidar "polimorfismo"
}
```

-- PUNTO 2.b

```
agregarMateria materia alumno = alumno {
  materiasAprobadas = materia : materiasAprobadas alumno
}

aprobar materia = agregarMateria materia . efecto materia
```

-- PUNTO 3

```
{-
> (aprobar (sistemasOperativos 3) . aprobar recursividadAFull . aprobar paradigmas) pepe
-}
```

-- PUNTO 4

```
type Cursada = [Materia]

hacer cursada alumno = foldr aprobar alumno cursada
```

-- PUNTO 5

```
-- Auxiliares
type CriterioDePuntos = Alumno -> Int

cantidadDeConceptos :: CriterioDePuntos
cantidadDeConceptos = length . conceptosAprendidos

dedicaciónNegativa :: CriterioDePuntos
dedicaciónNegativa = (* (-1)) . dedicación
```

```
tamañoDePrimerConcepto :: CriterioDePuntos
tamañoDePrimerConcepto = length . head . conceptosAprendidos

puntosLuegoDeCursarSegún criterioDePuntos cursada =
  criterioDePuntos . hacer cursada

-- Función principal
esMejorSegún :: CriterioDePuntos -> Cursada -> Cursada -> Alumno -> Bool

esMejorSegún criterioDePuntos cursadaGrosa cursadaFea alumno =
  puntosLuegoDeCursarSegún criterioDePuntos cursadaGrosa alumno >
  puntosLuegoDeCursarSegún criterioDePuntos cursadaFea alumno

{-
  Sería posible ver si es mejor una cursada que incluye Recursividad a Full, si es que comparamos por dedicación o si comparamos por los caracteres del primer concepto aprendido, ya que no necesitamos que se termine de generar la lista infinita de conceptos que hace que aprenda.

  En cambio, si comparamos por cantidad de conceptos aprendidos, ahí se cuelga porque nunca termina de terminar la lista como para saber su cantidad total.

  Ambos casos, son gracias a la evaluación diferida, ya que sin ella, no se podría ni siquiera empezar a evaluar algo con una entidad que aún no se terminó de construir y traer entera a memoria.
-}
```

-- PUNTO 6

```
data Materia = Materia {
  nombreMateria :: String,
  efecto :: EfectoEnAlumno,
  condiciónParaAprobar :: CondiciónParaAprobar
} deriving Show

type CondiciónParaAprobar = Alumno -> Bool

-- Auxiliares

sabe :: String -> CondiciónParaAprobar
sabe concepto = elem concepto . conceptosAprendidos

aprobó :: String -> CondiciónParaAprobar
aprobó unNombreDeUnaMateria = any ((==) unNombreDeUnaMateria . nombreMateria) .
materiasAprobadas

-- Materias nuevas

paradigmas = Materia {
  nombreMateria = "Paradigmas",
```

```
  efecto = aumentarDedicación 100 . aprenderConceptos ["polimorfismo", "orden superior"],
  condiciónParaAprobar = sabe "parametrización"
}

sistemasOperativos reentregas = Materia {
  nombreMateria = "Sistemas Operativos",
  efecto = agregarPrefijo "Excelentísim@" . aumentarDedicación (1000 * reentregas),
  condiciónParaAprobar = (\_ -> reentregas < 5)
}

recursividadAFull = Materia {
  nombreMateria = "Recursividad a Full",
  efecto = aprenderConceptos conceptosRecursivos,
  condiciónParaAprobar = aprobó "Recursividad a Full"
}

desastre2 = Materia {
  nombreMateria = "Desastre 2",
  efecto = olvidar "polimorfismo",
  condiciónParaAprobar = (\_ -> True)
}

aprobarSiEsQuePuede materia alumno
  | condiciónParaAprobar materia alumno = aprobar materia alumno
  | otherwise = error "No podía aprobar esta materia!"
```