Tercera Guerra Mundial - Resolución

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

-- Punto 1

```
data País = País {
  nombre :: Nombre,
  recursos :: Recursos,
  aliados :: [País],
  enemigos :: [País],
  estrategia :: Estrategia
} deriving Show
type Nombre = String
type Recursos = Double
type Estrategia = País -> País

estadosUnidos = País {
  nombre = "Estados Unidos",
  recursos = 1500,
  enemigos = [afganistán, coreaDelNorte, china],
  aliados = [alemania, inglaterra],
  estrategia = bombas 3 . influencia
}

rusia = País {
  nombre = "Rusia",
  recursos = 2100,
  enemigos = [turquía, estadosUnidos],
  aliados = [],
  estrategia = bombas 10 . complot . virus "Witzelsucht"
}

-- Lo de acá abajo es para que no se rompa al probar en GHCi nada más:

paísGenérico = País "" 1000 [] [] (bombas 1)

afganistán = paísGenérico {nombre = "Afganistán"}
coreaDelNorte = paísGenérico {nombre = "Corea del Norte"}
china = paísGenérico {nombre = "China"}
alemania = paísGenérico {nombre = "Alemania"}
inglaterra = paísGenérico {nombre = "Inglaterra"}
turquía = paísGenérico {nombre = "Turquía"}
```

-- Punto 2

```
afectarRecursosCon criterio país = nuevosRecursos ((criterio . recursos) país) país

nuevosRecursos unosRecursos unPaís = unPaís {recursos = min 0 unosRecursos}
```

```
nuevosEnemigosSegún criterio país = país {enemigos = (map potenciar . enemigos) país}

influencia enemigo = enemigo {aliados = []}

bombas cantidad = afectarRecursosCon (* (1 - 0.1 * cantidad))
-- O bien, otra interpretación puede ser: afectarRecursosCon (* 0.9 ^ cantidad)

virus nombre | length nombre < 20 = afectarRecursosCon ((+) (-100))
             | otherwise          = nuevosRecursos 0

complot = nuevosEnemigosSegún potenciar

potenciar = afectarRecursosCon (+ 1000)
```

-- Punto 3

```
resultadoDeGuerra = contraataqueDeEnemigos . atacarAEnemigos

atacarAEnemigos país = nuevosEnemigosSegún (estrategia país) país
```

-- Opción 1 (más fumeta)

```
contraataqueDeEnemigos país = (componerTodo . map estrategia . enemigos) país país

componerTodo = foldl (flip (.)) id
```

-- Opción 2 (más simple)

```
contraataqueDeEnemigos2 país = (foldl (flip estrategia) país . enemigos) país
```

-- Opción 3 (queda con el orden inverso de la lista pero era aceptable)

```
contraataqueDeEnemigos2 país = (foldr estrategia país . enemigos) país
```

-- Punto 4

```
-- Algunas funciones convenía hacerlas en cualquier caso:

recursosAceptables = (>= 500) . recursos

tieneMásRecursosQueSusEnemigos país = (all (tieneMásRecursos país) . enemigos) país

tieneMásRecursos paísConMásRecursos = (> recursos paísConMásRecursos) . recursos

-- Después podemos pensar en 2 opciones para lo que queda. Ambas eran aceptables.
```

-- Opción 1

```
-- Devolver a los países como estaban originalmente antes de la guerra

ganó país =
  (recursosAceptables . resultadoDeGuerra) país &&
  (tieneMásRecursosQueSusEnemigos . resultadtieneMásRecursosoDeGuerra) país
```

```
quiénesGanaron = filter ganó

-- Opción 2
-- Devolver a los países luego de la guerra. Es mejor, pero lo anterior era suficiente.

ganó2 país = recursosAceptables país && tieneMásRecursosQueSusEnemigos país

quiénesGanaron2 = filter ganó . map resultadoDeGuerra
```

-- Punto 5

Se dejan los 3 conceptos principales de los que se podía hablar, que fueron muy importantes al resolver este enunciado:

- **Orden superior**: Se usó al invocar funciones como map, que le mandamos por parámetro otra función. Esto hace que la lógica del map la podamos reutilizar para muchos casos sin tener que redefinirla en otras funciones. Lo mismo pasa en la función afectarRecursosCon y nuevosEnemigosSegún, que son muy genéricas gracias a recibir una función por parámetro.

- **Aplicación parcial**: Es cuando se envían menos argumentos a una función de los que necesita, y eso devuelve una nueva función que espera los parámetros restantes. Por ejemplo, en nuevosEnemigosSegún, que escribimos “map potenciar”, y eso es una nueva función que resulta de aplicar parcialmente a map. O en “bombas 3”, y otros casos más. Esto, al igual que el orden superior, también permite crear nueva lógica fácilmente (nuevas funciones, nuevo comportamiento) sin tener que codificar cosas nuevas.

- **Composición**: Es cuando generamos una nueva función a partir de combinar otras dos, haciendo que el resultado de una se le envíe como entrada a otra. De esta forma, creamos nuevas funcionalidades a partir de combinar otras sin tener que reescribir esas partes de la lógica. No codificamos nada nuevo, sino que combinamos fácilmente lo existente para hacer una nueva funcionalidad. Se usó, por ejemplo, en quiénesGanaron2, en donde componemos el filter parcialmente aplicado, con el map también parcialmente aplicado, haciendo que el resultado de ese map (una lista) se le envíe a ese filter (que espera recibir una lista).