Monsters Inc

Introducción y advertencia

El siguiente texto presenta el parcial de Monsters Inc, y una solución al mismo con variantes. No pretende ser exhaustivo, hay probablemente decenas de variantes igualmente buenas o mejores que las aquí expuestas.

Además, subraya algunos conceptos teóricos y marca errores comunes. Nuevamente, tampoco pretende cubrir todos los casos.

No debe entenderse a este parcial explicado como una guía para la resolución de un parcial, sino más bien como un apunte complementario. Y si bien la amplitud de la explicación se corresponde al alcance de la materia Paradigmas de Programación, también se incluyen pistas para profundizar un poco más y explorar otras posibilidades que el paradigma provee.

Por último, se recomienda encarecidamente no leer la misma hasta haber resuelto el parcial anteriormente.

Y para quienes son de otra generación, recomendamos mirar un poco
http://www.disney.es/FilmesDisney/Monstruos/index2.html

Problema

La reconocida empresa que transforma gritos de terror infantiles en energía limpia quiere organizar el trabajo de su monstruoso personal.

1) Obtener la energía que produce un grito, que se calcula como el nivel de terror por la intensidad al cuadrado, en caso de que moje la cama, si no, sólo es el triple del nivel de terror más la intensidad. El nivel de terror es equivalente al largo de la onomatopeya.
De los gritos
sabemos gritos están representados por una tupla (Onomatopeya, Intensidad, mojóLaCama).
Ejemplo:

```
energiaDeGrito ("AAAAAHG", 10, True)
700
energiaDeGrito ("uf", 2, False)
8
```

2) Obtener el grito que cada monstruo (un empleado de la empresa) le arranca a un niño (su víctima). Hay muchos monstruos y cada uno tiene su forma de trabajar, que consiste precisamente en recibir a un niño y devolver su grito. De los niños se sabe el nombre, la edad y la altura y se los representa con una tupla con dichas componentes.

Monstruos en la empresa hay muchos. En particular, se desea implementar los siguientes, pero podría haber más:

- Sullivan (el protagonista) produce un grito "AAAGH" con tantas A como letras tenga el nombre del niño y una intensidad como como 20 / edad; hace mojar la cama si el niño tiene menos de 3 años.
- Randall Boggs (la lagartija) produce un grito de “¡Mamadera!”, con tanta intensidad como vocales en el nombre de la persona; hace mojar la cama en los niños que miden entre 0,8 y 1,2 de altura.
- Chuck Norris produce siempre un grito que dice todo el abecedario, con 1000 de nivel de intensidad y siempre hace que mojen la cama.
- Un Osito Cariñoso produce un grito “uf”, con un nivel de intensidad igual a la edad del niño y nunca moja la cama.

3) Hacer una función que reciba una lista de funciones y un elemento, y que devuelva la lista que resulta de aplicar cada función sobre el elemento. Definir dominio e imagen.
Por ejemplo:

```
pam [doble, triple, raizCuadrada] 9
[18, 27, 3]
```

4) Los monstruos a veces trabajan en equipo, por lo que van varios a la casa de un niño y todos lo asustan. Obtener el conjunto de gritos que logra el equipo.
Por ejemplo:

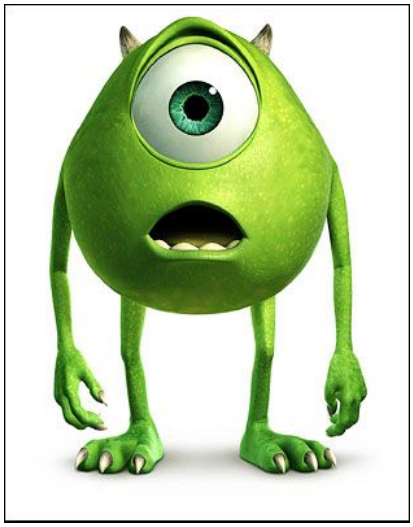
```
gritos [sullivan, osito, chuck] (“kevin”, 2, 1.1)
[("AAAAAHG", 10, True), (“uf”,2,False), (“abcdefghijklmnopqrstuvwxyz”,1000,True)]
```

5) Saber la producción energética de un equipo de monstruos que va a un campamento y asusta a todos los niños que están durmiendo en las carpas. Asumir que los campamentos son listas de niños y ya están definidos como constantes.
Por ejemplo:

```
produccionEnergeticaGritos [sullivan, osito, chuck] campamentoDeExploradores
999999
```

6) Ante la declinación en la producción energética de Monsters inc debido a las nuevas generaciones que ya no se asustan de monstruos obsoletos, la empresa ha decidido poner en marcha un proyecto de transformación de risas en energía, para lo que contrató a comediantes.
Las risas se representan por tuplas en las que se indica la duración y la intensidad. La energía que producen es la duración elevada a la intensidad. Cada comediante recibe un niño y devuelve una risa.
Por ejemplo:

- Capusotto produce una risa de duración igual al doble de la edad del niño y la misma intensidad.



Hacer la función **`produccionEnergeticaRisas`** que devuelve el total de energía producido por enviar a un equipo de comediantes a un campamento. Si es necesario, hacer funciones auxiliares pero no modificar las ya hechas.

7) Hacer una única función **`produccionEnergetica`** que reemplace a **`produccionEnergeticaRisas`** y a **`produccionEnergeticaGritos`** y que aplicada con los argumentos adecuados permita obtener lo mismo que ellas, es decir, que reciba un conjunto de monstruos o un conjunto de comendiantes y devuelva la energía producida.

(Una posible) Solución explicada

Este parcial nos plantea el modelo de dominio a medida que avanzamos con los puntos. En el primero de ellos nos muestra uno de estos elementos: el grito, el cual es modelado como una tupla de 3 elementos (terna), cuyo tipo[1] es:

```
(String, Int, Bool)
```[2]

Alternativamente, para ganar en expresividad y poder tratar este tipo a más alto nivel y en términos del dominio, también podríamos definir este tipo como un Grito, mediante un sinónimo de tipo (`type`)[3]:

```
type Grito = (String, Int, Bool)
```

Finalmente, para abstraernos del orden de las componentes, definiremos funciones que accedan a los mismos, que más adelante utilizaremos:

```
onomatopeya (o,_,_) = o

intensidad (_,i,_) = i

mojoLaCama (_,_,m) = m
```

Punto 1. Obtener la energía que produce un grito.

1) Obtener la energía que produce un grito, que se calcula como el nivel de terror por la intensidad al cuadrado, en caso de que moje la cama, si no, sólo es el triple del nivel de terror

---

[1] Tipo, o firma, o prototipo. La segunda forma, si bien correcta, es mas usual en el paradigma de objetos. La tercera debería evitarse, ya que proviene del vocabulario de los prototipos de C, y no es usada por la comunidad Haskell. En lo que resta de este texto, usaremos la primera forma.

[2] Dado que el tipo de los literales numéricos como 10 y 2 son en realidad de tipo `Num a => a`

```
Hugs> :t 2
2 :: Num a => a
```

sería también válido decir que los gritos tienen tipo `(String, Integer, Bool)` o `(String, Float, Bool)`, etc. Acá usaremos `Int` de forma (casi) arbitraria

[3] `type` es precisamente eso: un sinónimo, y no introduce un tipo nuevo. Solo existe para ganar expresividad y no afecta al funcionamiento del programa.

más la intensidad. El nivel de terror es equivalente al largo de la onomatopeya.
Los gritos están representados por una tupla (Onomatopeya, Intensidad, mojóLaCama).
Ejemplo:

```
energiaDeGrito ("AAAAAHG", 10, True)
700
energiaDeGrito ("uf", 2, False)
8
```

Esta función toma un grito tal como lo modelamos antes, y por el ejemplo vemos que devuelve un valor numérico entero. En particular, como este valor está en función de la intensidad del grito que es de tipo Int, veremos que esto restringe la imagen de la función a también Int:

```
energiaDeGrito grito
      | mojoLaCama grito = nivelTerror grito * (intensidad grito)^2
      | otherwise = 3 * nivelTerror grito + intensidad grito

nivelTerror grito = length (onomatopeya grito)
```

Esta solución presenta los siguientes elementos destacables:

- Uso de guardas: el requerimiento del punto 1 nos plantea una función partida de dos ramas: si moja la cama y no moja la cama.
- Delegación del nivel de terror: el nivel de terror es un concepto del dominio, y por tanto debería estar encapsulado en su propia función

A continuación, algunas variantes igualmente buenas o mejores:

Variante 1. Empleando función composición

La definición de nivelDeTerror conceptualmente[4] está componiendo la función onomatopeya con la función length. Entonces una buena idea es poner de manifiesto esto mediante la función composición `(.)`:

```
nivelTerror  grito =(length.onomatopeya) grito
```

Variante 2: Estilo Point Free

En este caso es posible "cancelar" el argumento grito a ambos lados de la ecuación de nivelDeTerror:

```
nivelTerror  ~~grito~~ =(length.onomatopeya) ~~grito~~
```

con lo que queda:

```
nivelTerror       = length.onomatopeya
```

Ambas definiciones son equivalentes.

Variante 3 (avanzada): Empleando pattern matching

Es posible definir de forma más sucinta la función `energiaDeGrito` empleando pattern matching y @:

```
energiaDeGrito grito@(_, intensidad, mojoLaCama)
       | mojoLaCama = nivelTerror grito * intensidad^2
       | otherwise = 3 * nivelTerror grito + intensidad
```

La contrapartida de esta solución es que nos estamos acoplando más fuertemente al formato de la tupla

[4] El concepto de composición está presente siempre que un problema se resuelva a través de la aplicación de una función al resultado de otra. Sin embargo, normalmente, cuando hablemos de composición nos estaremos refiriendo al uso de la función que *encapsula* este concepto:
`f(g(x)) = (fog)(x)`

Variante 4: Empleando pattern matching y `where`.

Otra versión similar a la anterior:

```
energiaDeGrito (onomatopeya, intensidad, mojoLaCama)
       | mojoLaCama = nivelTerror *intensidad^2
       | otherwise = 3 * nivelTerror + intensidad
            where nivelTerror = length onomatopeya
```

Esta solución tiene los mismos beneficios que la anterior, pero tiene una diferencia: `nivelTerror` pasa a estar definida localmente, con lo que no es reutilizable en otros contextos. Esto puede ser malo (si tengo que volver a definir `nivelTerror` de forma exactamente igual) o bueno (por ejemplo si, en otro contexto, también existe la idea de `nivelTerror`, pero significa algo distinto a lo que significa en este contexto).

A continuación, algunas variantes con errores comunes:

Error Comun 1: Uso redundante de los valores booleanos

```
energiaDeGrito grito
      | (mojoLaCama grito == True) = nivelTerror grito * (intensidad grito)^2
      | otherwise = 3 * nivelTerror grito + intensidad grito
```

El uso de la comparación `== True` es completamente redundante y por tanto incorrecto, dado que la imagen de `mojoLaCama` ya es un valor `Bool`. En definitiva, estamos escribiendo una expresión (lo que está resaltado) que se resuelve como un valor booleano, siendo este valor exactamente el mismo que el de la expresión que se encuentra a la izquierda del **==**.

Error Común 2: No delegar apropiadamente

```
energiaDeGrito grito
       |mojoLaCama grito = (length.onomatopeya) grito * (intensidad grito)^2
       |otherwise    = 3 * (length.onomatopeya) grito + intensidad grito
```

Esta solución, si bien funciona, es conceptualmente incorrecta no solo por la repetición de lógica, sino fundamentalmente porque oculta un concepto propio del dominio que está explicitado en el enunciado.

Punto 2: Obtener el grito que cada monstruo le arranca a un niño

2) Obtener el grito que cada monstruo (un empleado de la empresa) le arranca a un niño (su víctima). Hay muchos monstruos y cada uno tiene su forma de trabajar, que consiste precisamente en recibir a un niño y devolver su grito. De los niños se sabe el nombre, la edad y la altura y se los representa con una tupla con dichas componentes.

Lo que se pide en este punto no es más que obtener, para un niño, el grito correspondiente según el monstruo que lo asuste. Es decir, a partir de un niño dado, según el efecto que causa un monstruo, obtener el correspondiente grito. ¿A qué suena eso? Debería evocarnos la idea de función. Justamente, es la idea de este punto: modelar los monstruos como funciones que tienen un dominio `Niño` y una imagen `Grito`.

Además, debemos modelar a los niños, de los que se conoce nombre edad y altura. A partir de ello podríamos definir, por ejemplo, el siguiente sinónimo:

```
type Niño = (String, Int, Double)
```
[5]

Y algunas funciones que abstraigan de la constitución de los Niños:

```
nombre (nombreNiño, _, _) = nombreNiño
edad (_, edadNiño, _) = edadNiño
altura (_, _, alturaNiño) = alturaNiño
```

Ahora sí, vamos con los monstruos, recordando lo que decía el enunciado para cada uno de ellos:

- *Sullivan (el protagonista) produce un grito "AAAGH" con tantas A como letras tenga el nombre del niño y una intensidad como 20 / edad; hace mojar la cama si el niño tiene menos de 3 años.*

```
sullivan victima = (gritoGenerado, intensidadGenerada, haceMojarLaCama)
  where gritoGenerado      = gritoGenerado = map (\ _ -> 'A') (nombre victima) ++ "GH"
        intensidadGenerada = div 20 edadVictima
        haceMojarLaCama    = edad victima < 3
```

- *Randall Boggs (la lagartija) produce un grito de “¡Mamadera!”, con tanta intensidad como vocales en el nombre de la persona; hace mojar la cama en los niños que miden entre 0,8 y 1,2 de altura.*

[5] Hay quienes llevarían esto mas allá y definirán sinónimos de tipo también para las componentes de tipos simples, para dotar de mayor expresividad la definición. En el mundo real, se cuentan con mecanismos de definición de tipos algebraicos y abstractos de datos `(data)` que permiten resolver todos estos problemas de una forma más práctica y con mayor rigurosidad de chequeo de tipos que las definiciones `type`. Ver anteúltima sección.

```
randall victima = ("¡Mamadera!", intensidadGenerada, haceMojarLaCama)
  where intensidadGenerada = (length . filter esVocal . nombre) victima
        haceMojarLaCama    = altura victima >= 0.8 && altura victima <= 1.2
esVocal = flip elem "AEIOUaeiou"
```

- *Chuck Norris produce siempre un grito que dice todo el abecedario, con 1000 de nivel de intensidad y siempre hace que mojen la cama.*

```
chuck victima = ("abcdefghijklmnopqrstuvwxyz", 1000, True)
```

- *Un Osito Cariñoso produce un grito "uf", con un nivel de intensidad igual a la edad del niño y nunca moja la cama.*

```
osito victima = ("uf", edad victima, False)
```

Tanto `chuck` como `osito` son funciones simples y no requieren demasiada explicación.

Nos vamos a detener un poco en `sullivan` y `randall`. En ambos casos, con el fin de lograr una buena expresividad en la solución, se usaron definiciones locales de algunos componentes de la tupla. ¿Por qué definiciones ***locales***? Porque esos nombres que usamos para los componentes de las tuplas tienen sentido únicamente dentro de ese contexto. Notemos que el mismo nombre `intensidadGenerada`, por ejemplo, dentro del contexto de la función `sullivan` tiene un significado y dentro del contexto de la función `randall` tiene otro. A diferencia de esto, saber si "algo" es una vocal (función `esVocal`) es algo que muy probablemente podamos usar en otro contexto distinto, pero que lo vayamos a implementar de forma exactamente igual.

Variante 1: usando `repeat` (en `sullivan`)

`repeat` es una función definida en el Prelude, que simplemente devuelve una lista infinita de lo que sea que reciba como parámetro. En este caso, una lista infinita de 'A'. Podríamos definir la localmente la función `gritoGenerado` de la siguiente forma:

```
gritoGenerado = take ((length . nombre) victima) (repeat 'A') ++ "GH"
```

o equivalente:

```
gritoGenerado = (take ((length . nombre)  victima) . repeat) 'A' ++ "GH"
```

Variante 2: usando una lista definida a partir de un rango infinito (en `sullivan`)

También podríamos lograr lo mismo en `gritoGenerado` definiendo una lista potencialmente infinta en el rango 'A' a 'A':

```
gritoGenerado = take ((length.nombre) victima) ['A', 'A'.. ] ++ "GH"
```

Variante 3: eliminando paréntesis con `$` (en `sullivan`)

Es posible eliminar paréntesis superfluos empleando la función infija `$`[6]

```
sullivan victima = (gritoGenerado, intensidadGenerada, haceMojarLaCama)
  where gritoGenerado      = (take (length . nombre $ victima) . repeat $ 'A') ++ "GH"
        intensidadGenerada = div 20 edadVictima
        haceMojarLaCama    = edad victima < 3
```

Variante 4: empleando un rango finito (en `chuck`)

Dado que el tipo caracter (`Char`) es enumerable (esto es, pertence a la clase `Enum`), es posible emplearlo en listas por comprensión de la siguiente forma:

```
chuck victima = (['a'..'z'], 1000, True)
```

Variante 5: empleando notación infija (en `randall`)

El "acento hacia atrás" (`) puede usarse para convertir una función con identificador (es decir, con un nombre formado por letras) a la notación infija, en lugar de usarla en forma prefija como hacemos habitualmente. Esto permite aplicar parcialmente su segundo argumento como cuando hacemos, por ejemplo, (+4).

```
esVocal = (`elem` "AEIOUaeiou")
```

Variante 6: con parámetros explícitos (en `randall`)

```
esVocal letra  = elem letra "AEIOUaeiou"
```

[6] Ver resolución del punto 3 para una discusión más detallada de esta función

Punto 3: Implementar `pam`

(`map` al revés, para los amigos)
3) Hacer una función que reciba una lista de funciones y un elemento, y que devuelva la lista que resulta de aplicar cada función sobre el elemento. Definir dominio e imagen.
Por ejemplo:

```
pam [doble, triple, raizCuadrada] 9
[18, 27, 3]
```

Recordemos que las funciones tienen siempre un dominio e imagen bien definido, el cual constituye el tipo de la función. Aquí entonces se está pidiendo exactamente eso.

```
pam :: [a -> b] -> a -> [b]
```

Una posible solución, empleando listas por comprensión:

```
pam funciones valor = [ funcion valor | funcion <- funciones ]
```

Variante 1: con `map` y lambda

Sin embargo, mas allá del nombre, la reminisencia que `pam` tiene a `map` puede conducirnos a otras variantes interesantes que emplean orden superior.

Ambas funciones tienen en común que toman una lista, y aplican una transformación. En este caso, la transformación es justamente, aplicar la función al valor, tal como queda en evidencia en la primera solución[7]:

```
[ funcion valor | funcion <- funciones ]
```

Entonces, una variante posible, usando una expresión lambda es la siguiente:

```
pam funciones valor = map (\funcion -> funcion valor) funciones
```

Variante 2: con `where`

```
pam funciones valor = map aplicar funciones
            where aplicar funcion = funcion valor
```

Variante 3: con $ y aplicación parcial

El prelude define una función infija `($)` (se pronuncia “apply” o aplicar) que recibe dos

[7] En realidad, esa implementación no es más que un caso particular de la definición usando listas por comprensión de `map f xs = [ f x | x <- xs ]`

argumentos (una función y un valor de su dominio) y aplica la función al valor. Su definición es trivial:

```
($) f x = f x
```

No parece muy útil[8], pero en este caso es justo lo que necesitamos. Con lo que nos queda:

```
pam funciones valor = map ($ valor) funciones
```

Variante 4: recursivamente

Uff. recursividad.. volvimos al bajo nivel.... Normalmente evitaremos las soluciones de “aplicar” la función que sí pone en evidencia la función `($)`, pero cumple exactamente con lo pedido.

Punto 4. Obtener el conjunto de gritos que logra un equipo de monstruos

4) Los monstruos a veces trabajan en equipo, por lo que van varios a la casa de un niño y todos lo asustan. Obtener el conjunto de gritos que logra el equipo.
Por ejemplo:

```
gritos [sullivan, osito, chuck] (“kevin”, 2, 1.1)
[("AAAAAHG", 10, True), (“uf”,2,False), (“abcdefghijklmnopqrstuvwxyz”,1000,True)]
```

Aunque en la mayoría de los casos no es necesario definir los tipos en Haskell[9], aquí lo haremos solamente para fijar ideas: tenemos un niño y conjunto de monstruos, modelados mediante una lista, y lo que queremos es obtener los gritos que cada monstruo la arrebata a la víctima:

```
gritos ::[Niño -> Grito] ->  Niño -> [Grito]
gritos monstruos niño = pam monstruos niño
```

o equivalente:

```
gritos ::[Niño -> Grito] ->  Niño -> [Grito]
gritos = pam
```

Error común : no delegar apropiadamente

En este parcial resulta bastante evidente por lo “colgado” del punto anterior que `pam` será

[8] A veces se la utiliza para eliminar la necesidad de paréntesis. Ver las soluciones al punto 2.
[9] Esto es posible debido al mecanismo de inferencia de tipos de Haskell, que nos permite preservar un tipado estático sin necesidad de agregar metadatos sobre el tipo de las variables.

utilizado luego para algo, pero podría no haber sido tan directo. De hecho, normalmente esto no ocurrirá.

Siempre debemos aplicar nuestro criterio (no hay formulas mágicas) y pensar si podemos reutilizar lo anterior, y en caso de que no sea así, porqué: quizás esté bien, o quizás se nos haya escapado algún concepto mas general.

No hay que tener miedo de cambiar algo que funciona si pensamos que ese cambio es útil por algún motivo[10].

Punto 5.

5) Saber la producción energética de un equipo de monstruos que va a un campamento y asusta a todos los niños que están durmiendo en las carpas. Asumir que los campamentos ya están definidos y son funciones constantes que devuelven una lista con niños.
Por ejemplo:
`produccionEnergeticaGritos [sullivan, osito, chuck] campamentoDeExploradores 999999`

Este punto puede ser pensado de varias formas. Una de ellas es la siguiente:
"la energía que los monstruos obtienen de un campamento es la sumatoria de las energías de los gritos obtenidos al asustarlo".

En base a esta idea, podemos resolver el problema así:

```
produccionEnergeticaGritos monstruos niños =
      (sumatoriaEnergias.asustarCampamento monstruos) niños

sumatoriaEnergia gritos =
      foldl (\acum grito -> acum + energiaDeGrito grito) 0 gritos

asustarCampamento monstruos niños =
      (aplanar.map (\niño -> gritos niño monstruos)) niños

{-
aplanar toma una lista de listas y la transforma en una lista de los elementos
de las listas. Ejemplo:
aplanar [[4,5], [8], [1,2,6]] ==  [4,5,8,1,2,6]
-}
```

[10] Leer sobre refactorización y pruebas de regresión, temas que se tratarán en otras materias.

```
aplanar ~~listaDeListas~~ = foldl (++) [] ~~listaDeListas~~
```

Variante 1: utilizando `concat y sum`

El Prelude ya nos provee la función `aplanar`, y la llama `concat`, con lo que bien la podríamos haber utilizado en la solución anterior.
Por otro lado, la sumatoria de energías también podría descomponerse en una combinación de justamente esos dos conceptos: sumar compuesto con mapear las energías:

```
sumatoriaEnergia gritos =
      (sumatoria.map energiaGrito) gritos

sumatoria =  foldl (+) 0
```

Finalmente, esta función también es provista por el Prelude: `sum`. Combinando todos estos elementos, nos quedaría:

```
produccionEnergeticaGritos monstruos niños =
      (sumatoriaEnergias.asustarCampamento monstruos) niños

sumatoriaEnergia gritos =
      (sum.map energiaGrito) gritos

asustarCampamento monstruos niños =
      (concat.map (\niño -> gritos niño monstruos)) niños
```

Variante 2: Combinando las funciones de forma diferente, definiendo una función de orden superior

```
produccionEnergeticaGritos monstruos  ~~niños~~ =
      sumarListasDe (energiaCampamento monstruos) ~~niños~~

sumarListasDe f ~~lista~~ = (sum.concat.map f) ~~lista~~

energiaCampamento monstruos niño = map energiaDeGrito (gritos niño monstruos)
```

Variante 3: Otra combinación

```
produccionEnergeticaGritos monstruos =
      sum.map energiaDeGrito.concat.map (gritos monstruos)
```

O bien:

```
produccionEnergeticaGritos monstruos =
      sum.concat.map (map energiaDeGrito.gritos monstruos)
```

Aunque esta solucione puede parecer mejor que las anteriores por ser la más breve, lo cierto es que es también la menos delegada de las tres, con lo que perdemos tanto en declarativdad como en expresividad.

Variante 4: Con `concatMap`

La composición de `concat` y `map f` es tan común que ya viene definida en el Prelude: `concatMap`[11]

```
produccionEnergeticaGritos monstruos =
      sum.map energiaDeGrito.concatMap (gritos monstruos)
```

O como en la variante anterior, cambiando el orden:

```
produccionEnergeticaGritos monstruos =
      sum.concatMap (map energiaDeGrito.gritos monstruos)
```

Punto 6.

6) Ante la declinación en la producción energética de Monsters inc debido a las nuevas generaciones que ya no se asustan de monstruos obsoletos, la empresa ha decidido poner en marcha un proyecto de transformación de risas en energía, para lo que contrató a comediantes. Las risas se representan por tuplas en las que se indica la duración y la intensidad. La energía que producen es la duración elevada a la intensidad. Cada comediante recibe un niño y devuelve una risa.
Por ejemplo:

- Capusotto produce una risa de duración igual al doble de la edad del niño y la misma intensidad.

Hacer la función **`produccionEnergeticaRisas`** que devuelve el total de energía producido por enviar a un equipo de comediantes a un campamento. Si es necesario, hacer funciones auxiliares pero no modificar las ya hechas.

```
produccionEnergeticaRisas comediantes niños =
      (sumatoriaEnergiasRisas.hacerReirCampamento comediantes) niños

sumatoriaEnergiaRisas risas =
      (sum.map energiaRisa) risas

hacerReirCampamento comediantes niños =
```

[11] La función concatMap tiene un rol importante en otro posible enfoque, más general, de la transformación de listas. El curioso puede investigar sobre la función (>>=) (se pronuncia “bind”)

```
      (concat.map (\niño -> pam niño comediantes)) niños
```

Punto 7.

7) Hacer una única función **`produccionEnergetica`** que reemplace a **`produccionEnergeticaRisas`** y a **`produccionEnergeticaGritos`** y que aplicada con los argumentos adecuados ermita obtener lo mismo que ellas, es decir, que reciba un conjunto de monstruos o un conjunto de comendiantes y devuelva la energía producida

La solución del punto anterior es estúpidamente parecida a la variante 1 de produccionEnergéticaGritos, lo cual es malo, malo, muy malo. Estamos repitiendo lógica, es decir, subyace un concepto más general que no estamos encapsulando apropiadamente: producciónEnergética. Dicho de otra forma, toda lógica debería estar implementada en un único lugar [12].

Lo que nos pide este punto es justamente remover esta duplicación.
Para ellos, debemos definir una función más genérica, que resuelva el problema de la producción enérgetica, ya sea de gritos o risas.

Si miramos nuevamente el código anterior, veremos que lo que está en negrita es igual en ambas soluciones

```
produccionEnergeticaRisas comediantes niños =
      (sumatoriaEnergiasRisas.hacerReirCampamento comediantes) niños

sumatoriaEnergiaRisas risas =
      (sum.map energiaRisa) risas

hacerReirCampamento comediantes niños =
      (concat.map (\niño -> pam niño comediantes)) niños
```

Es decir, estas soluciones está acopladas en sus nombres de variables y la función que aplican (energiaRisas o energíaGritos). Entonces, simplemente, parametricemos esas funciones, y de paso, utilicemos nombres más genéricos:

```
produccionEnergetica productores niños =
      (sumatoriaEnergias.extraerEnergiaCampamento productores) niños
```

[12] Principios conocidos en la jerga como DRY (don't repeat yourself) y Once and only once. Son fundamentales, dado que lógica repetida (y potencialmente implementada una y otra vez de la misma forma, en el mejor de los casos, y de forma distinta, en el peor) resultará indefectiblemente en código inmantenible y dolores de cabeza a futuro, para el desdichado que tenga que modificar nuestro código (y uno mismo es frecuentemente ese desdichado).

```
sumatoriaEnergia producciones energiaProduccion =
      (sum.map energiaProduccion) producciones

extraerEnergiaCampamento productores niños =
      (concat.map (\niño -> pam niño productores)) niños
```

Modelado Alternativo con `data`

Este parcial nos propone modelar nuestros datos compuestos empleando tuplas. Es una forma válida, que como ventaja, es simple, pero tiene las siguientes contras:

- Pérdida de expresividad: la tupla (10, 10, 10) podría representar tanto una fecha como las cantidades de harina, leche y azúcar de un budín.
- Pierdo capacidad de chequeo de tipos: usando el ejemplo anterior, podría pasarle a una función, erróneamente, una tupla con un formato adecuado pero con una semántica incompatible con la misma, dado que type es tan solo un sinónimo[13].
- En general, con tuplas hay muchas estructuras que no puedo modelar (por ejemplo, las estructuras recursivas)
- `data` abre la puerta a varias técnicas avanzadas que las tuplas no.

Otra alternativa es, en lugar de usar simples tuplas, emplear `data`:

```
data Grito = Grito String Int Bool
```

o, aún mejor, emplear data + sintaxis de registros

```
data Grito = Grito {
       onomatopeya :: String,
       intensidad :: Int,
       mojoLaCama :: Bool } deriving Show
```

que tiene la ventaja de que no hay que definir las funciones que acceden a las componentes, porque vienen gratis:

```
> onomatopeya (Grito “ahhhh” 50 True) -- onomatopeya no hay que definirla
“ahhhh”
```

Luego, para los niños:

[13]Si bien no siempre querremos cargar al sistema de tipos con más aún más responsabilidades, por ejemplo, embebiendo verificaciones semánticas además de sintácticas, esto es Haskell es posible. El curioso puede consultar sobre newtype y phantom types

```
data Niño = Niño {
      nombreNiño::String,
      edadNiño:: Int,
      alturaNiño:: Double } deriving Show
```

Y para la risa:

```
data Risa = Risa { duracionRisa  ::Int,
                   intensidadRisa::Int } deriving Show
```

Luego las funciones anteriores quedan iguales, salvo aquellas que deconstruyan la tupla en sus componentes. Por ejemplo la siguiente función:

```
energiaDeGrito (onomatopeya, intensidad, mojoLaCama)
       | mojoLaCama = nivelTerror *intensidad^2
       | otherwise = 3 * nivelTerror + intensidad
            where nivelTerror = length onomatopeya
```

ahora pasa a ser:

```
energiaDeGrito (Grito onomatopeya intensidad mojoLaCama)
       | mojoLaCama = nivelTerror *intensidad^2
       | otherwise = 3 * nivelTerror + intensidad
            where nivelTerror = length onomatopeya
```

Clases: Más allá del punto 7 y del parcial.

Este parcial puede servir también como punto de partida para explicar algunos conceptos más allá del ejercicio: implementación de polimorfismo mediante clases de tipos (typeclassess), e instance[14].

Partiendo de las definiciones data del punto anterior, podríamos agregar las siguientes definiciones de instancias.

```
class GeneradorEnergia a where
      energia :: a -> Int

instance GeneradorEnergia Grito where
      energia = energiaDeGrito
```

[14] Para el interesado en esta sección, recomendamos leer algún buen tutorial o libro

```
instance GeneradorEnergia Risa where
      energia = energiaDeRisa
```

Con esto ahora podríamos modificar la función de generación de energía del punto 7, para que, en lugar de pasarle por parámetro la función extractora de energia, utilice la correspondiente al tipo de producción (Risa o Grito)

```
produccionEnergetica :: GeneradorEnergia a => (Niño -> a) -> [a] -> Int
produccionEnergetica productores niños =
      (sumatoriaEnergias.extraerEnergiaCampamento productores) niños

extraerEnergia Campamento productores niños =
      (concat.map (\niño -> pam niño productores)) niños
```

Nota: las funciones subrayadas son las mismas que las utilizadas en las soluciones anteriores