Entrevistando por un sueño

Una consultora de RRHH, además de agregarnos a LinkedIn, nos pidió que le hagamos un sistema que los ayude con sus tareas habituales.
La idea es básicamente saber si alguien es apto para un puesto.

De cada postulante se conoce su nombre, su edad, su remuneración pretendida y una lista de conocimientos.

```
data Postulante = UnPostulante {nombre :: String, edad ::
Int, remuneracion :: Float, conocimientos :: [String]}

pepe = UnPostulante "Jose Perez" 35 15000.0 ["Haskell",
"Prolog", "Smalltalk", "C"]
tito = UnPostulante "Roberto González" 20 12000.0 ["Haskell",
"Php"]
```

También hay algunos puestos a cubrir, con sus correspondientes conocimientos necesarios:

```
jefe = UnPuesto "gerente de sistemas" ["Haskell", "Prolog",
"Smalltalk"]
chePibe = UnPuesto "cadete" ["ir al banco"]
```

1) **Recursos humanos**

La empresa tiene una política de recursos humanos que se basa en las siguientes requisitos:

a. *tieneConocimientos* recibe un puesto y un postulante y devuelve si el postulante posee todos los conocimientos requeridos para el puesto.
b. *edadAceptable* recibe una edad mínima y una edad máxima y un postulante y devuelve si el postulante tiene la edad requerida para el puesto.

c. *sinArreglo* recibe a un postulante y verifica que no tenga el apellido del dueño de la empresa. (se asume que el apellido son las últimas letras del nombre y que ya existe una función que retorna el apellido del dueño)

```
apellidoDuenio :: String
```

2) Preselección

En base a lo anterior, y previendo la existencia de nuevos criterios, definir una función que recibe una lista de postulantes y una lista con los requisitos de selección que devuelva a los postulantes que cumplen con todo lo indicado.

a. Mostrar un ejemplo de consultas para el puesto de gerente de sistemas, con edad entre 30 y 40 años, con los conocimientos correspondientes y que no esté arreglado.
b. Sin definir una nueva función, añadir a la consulta anterior, un nuevo requisito, que consiste en que entre los conocimientos del postulante no esté "repetir lógica".

3) Esperando que te llamen

Representar el paso de un año de espera entre los postulantes. Sabiendo que existe una función *incrementarEdad* que incrementa en 1 la edad del postulante y una función *aumentarSueldo* que recibe un porcentaje y un postulante y le aumenta en ese porcentaje el sueldo que pretende cobrar, y sin hacer nuevas funciones auxiliares, definir la función *actualizarPostulantes* que reciba una lista con postulantes y, que además de aumentar la edad a todos, actualice sus sueldos pretendidos en un 27%. Realizarlo de dos maneras:

a. Utilizando listas por comprensión.
b. Utilizando composición y aplicación parcial.

¿Cual de las dos soluciones preferís? ¿por qué?

¿Qué sucede si la lista de postulantes es infinita? Justificar. Mostrar un ejemplo de consulta donde suceda lo indicado.

4) Capacitaciones

La empresa decide comenzar a hacer capacitaciones a los postulantes, pero además de los que se presentaron, convoca a los alumnos de la UTN anotados en la bolsa de trabajo, para que también se nutran y en un futuro entren a la empresa.

a. Definir la función *capacitar* que dado un postulante y un conocimiento, lo agregue a su lista de conocimientos.
b. Definir la misma función pero para un estudiante, del cual sólo se conoce el legajo y sus conocimientos, y que cuando se lo quiere capacitar, aprende el nuevo conocimiento, pero se olvida del último.
c. Hacer una función *capacitación*, que reciba un puesto y una persona, ya sea estudiante o postulante, y lo capacite en todos los conocimientos necesarios para dicho puesto. Definir su tipo y mostrar al menos dos ejemplos de invocación.
d. ¿Qué más habría que definir para que se pueda hacer una preselección entre los estudiantes? Justificar

SOLUCION

--ENTREVISTANDO POR UN SUEÑO
--Definiciones generales de tipos de datos y ejemplos

--Simplemente por expresividad
type Conocimiento = String
type Nombre = String

data Postulante = UnPostulante {nombre :: Nombre, edad :: Int, remuneracion :: Float, conocimientos :: [Conocimiento]} deriving Show

pepe = UnPostulante "Jose Perez" 35 15000.0 ["Haskell", "Prolog", "Smalltalk", "C", "repite logica"]
tito = UnPostulante "Roberto Gonzalez" 20 12000.0 ["Haskell", "Php"]

data Puesto = UnPuesto {puesto :: String, conocimientosRequeridos :: [Conocimiento]}

jefe = UnPuesto "gerente de sistemas" ["Haskell", "Prolog", "Smalltalk"]
chePibe = UnPuesto "cadete" ["ir al banco"]

-- 1) Recursos humanos

--La empresa tiene una política de recursos humanos que se basa en las siguientes requisitos:

-- A) tieneConocimientos recibe un puesto y un postulante y devuelve si el postulante posee todos los conocimientos requeridos para el puesto.

tieneConocimientos:: Puesto->Postulante->Bool
tieneConocimientos puesto alguien = all (tieneConocimiento alguien) (conocimientosRequeridos puesto)

tieneConocimiento:: Postulante->Conocimiento->Bool
tieneConocimiento alguien conocimiento = elem conocimiento (conocimientos alguien)

-- B) edadAceptable recibe una edad mínima y una edad máxima y un postulante y devuelve si el postulante tiene la edad requerida para el puesto.

edadAceptable :: Int-> Int -> Postulante -> Bool
edadAceptable edadMin edadMax alguien = edad alguien > edadMin && edad alguien < edadMax
--Variante con pattern matching

```
--edadAceptable edadMin edadMax (UnPostulante _ edad _ _) = edad > edadMin &&
edad < edadMax

-- C) sinArreglo recibe a un postulante y verifica que no tenga el apellido del dueño de la
empresa. (se asume que el apellido son las últimas letras del nombre y que ya existe una
función que retorna el apellido del dueño)

sinArreglo :: Postulante -> Bool
sinArreglo alguien = not (terminaCon (nombre alguien) apellidoDueno)

apellidoDueno::Nombre
apellidoDueno = "Gonzalez"

terminaCon:: Nombre->Nombre -> Bool
terminaCon nombre apellido = apellido == drop (length nombre - length apellido) nombre

-- Algunas variantes posibles
{-
terminaCon nombre apellido = reverse apellido == take (length apellido) (reverse nombre)

terminaCon [] apellido = False
terminaCon (_:ultimos) apellido
        | ultimos == apellido = True
        | otherwise = terminaCon ultimos apellido

terminaCon [] apellido = False
terminaCon nombre apellido = ultimos == apellido || terminaCon ultimos apellido
        where ultimos = tail nombre
-}

--------------------------------------------------------------------------------
-- 2) Preselección

--Para el uso que se le da en este item, podria ser:
--preseleccion::[Postulante]->[Postulante->Bool]->[Postulante]
--pero en realidad, su tipo es mucho mas genérico:
preseleccion::[a]->[a->Bool]->[a]
preseleccion postulantes requisitos = filter (cumpleTodos requisitos) postulantes

cumpleTodos requisitos postulante = all (\requisito -> requisito postulante) requisitos

--Variantes
{-
cumpleTodos [] _ = True
```

```
cumpleTodos (requisito:requisitos) postulante =
        requisito postulante && cumpleTodos requisitos postulante

cumpleTodos requisitos postulante = all ($ postulante) requisitos
-}

-- A)
-- Ejemplo de consulta
-- preseleccion [tito,pepe] [edadAceptable 30 40, tieneConocimientos jefe, sinArreglo]

-- B)
-- Ejemplo de consulta
-- preseleccion [tito,pepe] [edadAceptable 30 40, tieneConocimientos jefe, sinArreglo,
not.elem "repite logica".conocimientos]

-- O también, con una expresión lambda
-- preseleccion [tito,pepe] [edadAceptable 30 40, tieneConocimientos jefe, sinArreglo,
(\(UnPostulante _ _ _ conocimientos) -> notElem "repite logica" conocimientos)]

-------------------------------------------------------------------------------
-- 3) Esperando que te llamen

--Representar el paso de un año de espera entre los postulantes. Sabiendo que existe
una función incrementarEdad que incrementa en 1 la edad del postulante y una función
aumentarSueldo que recibe un porcentaje y un postulante y le aumenta en ese porcentaje
el sueldo que pretende cobrar, y sin hacer nuevas funciones auxiliares, definir la función
actualizarPostulantes que reciba una lista con postulantes y, que además de aumentar la
edad a todos, actualice sus sueldos pretendidos en un 27%. Realizarlo de dos maneras:

-- A) Utilizando listas por comprensión.

pasoDelTiempo postulantes =
        [incrementarEdad (aumentarSueldo 27 alguien) | alguien <- postulantes]

-- B) Utilizando composición y aplicación parcial.

pasoDelTiempo' postulantes = map (incrementarEdad.(aumentarSueldo 27)) postulantes

incrementarEdad (UnPostulante n edad r c) = UnPostulante n (edad +1) r c
aumentarSueldo inc (UnPostulante n e sueldo c) = UnPostulante n e (sueldo*(1+inc/100))
c

-- ¿Cual de las dos soluciones preferís? ¿por qué?
-- Cualquiera...
```

```
-- ¿Qué sucede si la lista de postulantes es infinita? Justificar. Mostrar un ejemplo de consulta donde suceda lo indicado.
-- Devuelve otra lista infinita, gracias a la evaluación diferida.
-- Ejemplo de consulta:
-- pasoDelTiempo (repeat pepe)
-- Retorna:
-- [UnPostulante {nombre = "Jose Perez", edad = 36, remuneracion = 19050.0, conocimientos = ["Haskell","Prolog","Smalltalk","C","repite logica"]},...]

-------------------------------------------------------------------------------
-- 4) Capacitaciones

--La empresa decide comenzar a hacer capacitaciones a los postulantes, pero además de los que se presentaron, convoca a los alumnos de la UTN anotados en la bolsa de trabajo, para que también se nutran y en un futuro entren a la empresa.

-- A) Definir la función capacitar que dado un postulante y un conocimiento, lo agregue a su lista de conocimientos.

--(Para este item bastaba con hacer lo siguiente. La solución terminada esta integrada con el item B, para lo cual se mantiene la implementación y sólo se modifica la definicion del tipo de dato)
{-
capacitar :: Postulante -> Conocimiento -> Postulante
capacitar (UnPostulante n e r conocimientos) conocimiento =
        UnPostulante n e r (conocimiento:conocimientos)
-}

-- B) Definir la misma función pero para un estudiante, del cual sólo se conoce el legajo y sus conocimientos, y que cuando se lo quiere capacitar, aprende el nuevo conocimiento, pero se olvida del último.
class Persona a where
        capacitar:: a->Conocimiento->a

instance Persona Postulante where
        capacitar (UnPostulante n e r conocimientos) conocimiento =
                UnPostulante n e r (conocimiento:conocimientos)

data Estudiante = UnEstudiante {legajo :: String, conocimientosEstudiante :: [Conocimiento]} deriving Show
instance Persona Estudiante where
        capacitar (UnEstudiante l conocimientos) conocimiento =
                UnEstudiante l (conocimiento:init conocimientos)
```

-- C) Hacer una función capacitación, que reciba un puesto y una persona, ya sea estudiante o postulante, y lo capacite en todos los conocimientos necesarios para dicho puesto. Definir su tipo y mostrar al menos dos ejemplos de invocación.

```
capacitacion :: Persona a => Puesto -> a -> a
capacitacion puesto alguien = foldl capacitar alguien (conocimientosRequeridos puesto)
```

```
-- Ejemplos de invocación
-- capacitacion jefe pepe
-- UnPostulante {nombre = "Jose Perez", edad = 35, remuneracion = 15000.0,
conocimientos = ["Smalltalk","Prolog","Haskell","Haskell","Prolog","Smalltalk","C","repite logica"]}

-- capacitacion chePible (UnEstudiante "123.456-7" ["aprobar materias"])
-- UnEstudiante {legajo = "123.456-7", conocimientosEstudiante = ["ir al banco"]}
```

-- D) ¿Qué más habría que definir para que se pueda hacer una preselección entre los estudiantes? Justificar
-- Estrictamente, sólo basta definir funciones de selección para los estudiantes, de tipo Estudiante->Bool. Si se quisiera usar criterios similares que para los postulantes, es decir funciones con el mismo nombre y diferente implementación, se las debería declarar como parte de la type class persona y al definir el tipo Estudiante hacer su correspondiente implementación.