Jugando con las palabras

# Autores

Asier Merino Herrán - 20m024

Guillermo Filgueira Fernández - 20M019

Guillermo Fernández Sánchez - 20M082

# Funcionamiento del código

Se sigue la lógica de programa que se recomienda en el enunciado: se pide al usuario tanto las letras en la mano como las letras en el tablero, se generan las posibles palabras y se hace un filtrado según las palabras sean válidas o no. Primero se explicará cómo se generan las palabras y como se establece si una palabra es válida y por último se explicará la parte ejecutable de código.

## Generación de palabras

--Genera todas las palabras posibles

generaPalabrasNum :: Integer -> String -> [String]

generaPalabrasNum n str = nub $ permutaLetras(combinacionesNum n str)

--Calcula las permutaciones de todas las palabras de la lista

permutaLetras :: [String] -> [String]

permutaLetras strs = nub $ concatMap (\str -> permutations str) strs

--Devuelve todas las combinaciones posibles de cierta longitud

combinacionesNum :: Integer -> String -> [String]

combinacionesNum = combinaciones\_2

-- 2ª definición

combinaciones\_2 :: Integer -> [a] -> [[a]]

combinaciones\_2 0 \_          = [[]]

combinaciones\_2 \_ []         = []

combinaciones\_2 k (x:xs) =

    [x:ys | ys <- combinaciones\_2 (k-1) xs] ++ combinaciones\_2 k xs

En primer lugar se calculan las posibles combinaciones de letras mediante las funciones combinaciones\_2 y combinacionesNum. Para conseguir una mayor rapidez en el cálculo (ya que el cálculo de todas las posibles combinaciones es uno de los principales problemas a lo que se puede enfrentar este enunciado), se han sacado ambas funciones de internet de la siguiente [página web](https://www.glc.us.es/~jalonso/vestigium/i1m2014-combinatoria-en-haskell/). Estas funciones calculan todas las posibles combinaciones de una cierta longitud a partir de una letras pasadas en forma de lista (o string, que es equivalente en Haskell). Por ejemplo:

**\*Main>** combinacionesNum 4 "abcdef"

["abcd","abce","abcf","abde","abdf","abef","acde","acdf","acef","adef","bcde","bcdf","bcef","bdef","cdef"]

**\*Main>** combinacionesNum 3 "abcdef"

["abc","abd","abe","abf","acd","ace","acf","ade","adf","aef","bcd","bce","bcf","bde","bdf","bef","cde","cdf","cef","def"]

**\*Main>** combinacionesNum 2 "abcdef"

["ab","ac","ad","ae","af","bc","bd","be","bf","cd","ce","cf","de","df","ef"]

**\*Main>**

En cuanto a las permutaciones se utiliza la función permutations de la librería Data.List de Haskell.

Una vez tenemos las posibles agrupaciones de letras queremos que se reordenen de todas las maneras posibles, para ello tenemos la función permutaLetras , a la que le pasamos una lista de agrupaciones de letras (el resultado de combinacionesNum) y nos devuelve las distintas permutaciones de cada palabra. Para evitar repeticiones se usa la función de Haskell nub. Todo este procedimiento se encapsula en la función generaPalabrasNum que se utilizará en el main del programa.

**\*Main>** generaPalabrasNum 3 "abcde"

["abc","bac","cba","bca","cab","acb","abd","bad","dba","bda","dab","adb","abe","bae","eba","bea","eab","aeb","acd","cad","dca","cda","dac","adc","ace","cae","eca","cea","eac","aec","ade","dae","eda","dea","ead","aed","bcd","cbd","dcb","cdb","dbc","bdc","bce","cbe","ecb","ceb","ebc","bec","bde","dbe","edb","deb","ebd","bed","cde","dce","edc","dec","ecd","ced"]

**\*Main>** generaPalabrasNum 2 "abcd"

["ab","ba","ac","ca","ad","da","bc","cb","bd","db","cd","dc"]

**\*Main>**

## Funciones auxiliares

El análisis de los caracteres ‘ch’ y ‘ll’ debe ser especial por lo que se van a sustituir por 1 y 2 respectivamente para facilitar su manipulación.

--Sustituye letras especiales (como ch o ll) por un codigo

sustLetrasACod :: String -> String

sustLetrasACod [] = []

sustLetrasACod [x] = [x]

sustLetrasACod (x:y:res)

                    | x == 'c' && y == 'h' = ('1':sustLetrasACod res)

                    | x == 'l' && y == 'l' = ('2':sustLetrasACod res)

                    | otherwise = (x:sustLetrasACod (y:res))

--Sustituye todas las palabras de una lista segun la funcion sustLetrasACod

sustListaACod :: [String] -> [String]

sustListaACod = map sustLetrasACod

--Sustituye codigo por sus respectivas letras especiales (como ch o ll)

sustCodALetras :: String -> String

sustCodALetras [] = []

sustCodALetras [x] = [x]

sustCodALetras (x:res)

                    | x == '1' = ('c':'h':sustCodALetras res)

                    | x == '2' = ('l':'l':sustCodALetras res)

                    | otherwise = (x:sustCodALetras res)

--Sustituye todas las palabras de una lista segun la funcion sustCodALetras

sustListaALetras :: [String] -> [String]

sustListaALetras = map sustCodALetras

Además se tiene definidas algunas funciones auxiliares que facilitan el desarrollo del código.

--Devuelve true si es consonante

esVocal :: Char -> Bool

esVocal c = c `elem` "aeiou"

--Devuelve true si es consonante

esConsonante :: Char -> Bool

esConsonante c = not (esVocal c) && (c `elem` ['a'..'z'] || c `elem` ['1', '2'])

--Dado un string devuelve solo las letras y convertidas todas a minuscula

soloLetrasMinusculas :: String -> String

soloLetrasMinusculas str = map toLower (filter isAlpha str)

## Validación de palabras

Una vez hayamos generado todas las posibles combinaciones de letras, queremos descartar aquellas palabras que no tengan una construcción valida en el castellano. Para ello establecemos dos tipos de reglas: las reglas generales, que se aplican sobre toda la palabra y que pueden ser por ejemplo que no puede haber tres o más consonantes seguidas o que después de una ‘q’ debe venir una ‘u’ más vocal; y las reglas relacionadas con las silabas, por ejemplo que consonante al final de sílaba solo puede ser ‘l’, ’n’, ’s’ o ‘r’.

--Devuelve true si la palabra es valida con las reglas definidas

palabraValida :: String -> Bool

palabraValida str = palabraValidaGeneral str && palabraValidaPorSilabas str

### Comprobaciones generales

--Realiza comprobaciones generales sobre la palabra y devuelve si es valida

palabraValidaGeneral :: String -> Bool

palabraValidaGeneral str =

  let

    funciones = [noMasTresConsonantesSeguidas, qMasUMasConsonante, hMasVocal, noConsMasH, dosLetrasIguales, noTerminaDosConsonantes, noMasTresVocalesSeguidas, noCaracEspMasVocal]  -- Lista de funciones de reglas generales a comprobar TODO

  in

    all (\f -> f str) funciones

El español tiene una gran cantidad de reglas a la hora de la construcción de palabras por que la idea del programa es que sea lo más flexible posible a la hora de añadir reglas. Es por eso que se ha definido la función de manera que si se quiere implementar una nueva reglas que se haya olvidado simplemente haya que crear una nueva función y añadirla a la lista ‘funciones’. Estas funciones deben ser todas de la forma regla :: String -> Bool de manera que devuelvan ‘False’ si la palabra no cumple la reglas y ‘True’ si la cumplen. Un ejemplo de este tipo de funciones que implementan reglas generales es:

-- Devuelve False si encuentra alguna 'q' que no esté seguida de una 'u' y una vocal

qMasUMasConsonante :: String -> Bool

qMasUMasConsonante [] = True

qMasUMasConsonante [x] = x /= 'q'

qMasUMasConsonante [x, y] = x /= 'q' && qMasUMasConsonante [y]

qMasUMasConsonante (x:y:z:res) = if x == 'q'

                                   then y == 'u' && esVocal z && qMasUMasConsonante (y:z:res)

                                   else qMasUMasConsonante (y:z:res)

Lista de reglas generales que se han aplicado:

* No puede haber tres o más consonantes seguidas
* La letra ‘q’ debe ir seguida de un ‘u’ y una vocal
* La letra ‘h’ debe estar seguida de una vocal
* La letra ‘h’ no puede estar precedida por una consonante
* …….

### Comprobaciones relacionadas a las sílabas

--Hace un busqueda en profundidad para ver si una palabra es valida separandola por silabas y comprobando cada silaba, supone silabas de hasta 4 letras

palabraValidaPorSilabas :: String -> Bool

palabraValidaPorSilabas [] = True

palabraValidaPorSilabas (x:y:z:w:resto) = (silabaValida [x, y, z, w] && labraValidaPorSilabas resto)

                                         || (silabaValida [x, y, z] && palabraValidaPorSilabas (w:resto))

                                         || (silabaValida [x, y] && palabraValidaPorSilabas (z:w:resto))

                                         || (silabaValida [x] && palabraValidaPorSilabas (y:z:w:resto))

palabraValidaPorSilabas (x:y:z:resto) = (silabaValida [x, y, z] && palabraValidaPorSilabas resto)

                                         || (silabaValida [x, y] && palabraValidaPorSilabas (z:resto))

                                         || (silabaValida [x] && palabraValidaPorSilabas (y:z:resto))

palabraValidaPorSilabas (x:y:resto) = (silabaValida [x, y] && palabraValidaPorSilabas resto)

                                         || (silabaValida [x] && palabraValidaPorSilabas (y:resto))

palabraValidaPorSilabas (x:resto) = silabaValida [x] && palabraValidaPorSilabas resto

La comprobación de reglas relacionadas con las sílabas es el principal reto de la práctica, y es que en el español no hay un algoritmo exacto para separar una palabra en sílabas, y mucho menos si tenemos en cuenta que la palabra puede no seguir ningún tipo de regla de construcción. Por lo tanto se ha tenido que seguir una búsqueda “en profundidad”.

En primer lugar se ha establecido que en el español una sílaba tiene, como máximo, cuatro letras. Para comprobar si una palabra tiene sílabas que cumplen las características del castellano se ha hecho, mediante if-else, una búsqueda en profundidad sobre un hipotético árbol cuyas ramas corresponden a las distintas posibles separaciones de la palabra en silabas, empezando por las sílabas más grandes posibles. Por ejemplo el hipotético árbol para la palabra ‘árbol’ sería:

Diagrama, Esquemático

Descripción generada automáticamente

La función palabraValidaPorSilabas recorre este árbol de manera recursiva de izquierda a derecha y por cada vez que llega a un nodo comprueba si la sílaba es válida mediante una función llamada silabaValida. En este ejemplo empezaría llegando a la sílaba ‘arbo’, que supongamos que es válida. Entonces continuaría con la sílaba ‘l’, que no puede ser válida ya que una sílaba de una sola consonante no es válida en el castellano. Por lo tanto volvería atrás y seguiría con la sílaba ‘arb’, que no puede ser válida ya que la consonante ‘b’ al final de sílaba no es válido. Por lo tanto seguiría con ‘ar’, que parece válida, que junto con ‘bol’ que también es válida conseguiría una separación de sílabas válidas y por lo tanto palabraValidaPorSilabas devolvería True. Para evitar la evaluación completa del árbol se utiliza la evaluación perezosa que implementa Haskell mediante && y ||.

Por lo tanto solo nos queda definir cuando una sílaba es válida o no. Al igual que en la función palabraValidaGeneral el objetivo es que el código sea lo más flexible posible, por lo que se usa la misma estrategia que antes definiendo en una lista las funciones que van a hacer la comprobaciones.

--Comprueba si una silaba es valida, supone que las silabas son de maximo 4 letras

silabaValida :: String -> Bool

silabaValida [x] =   let funciones = [esVocal]

                     in all (\f -> f x) funciones

silabaValida str@[\_, \_] = let funciones = [consonanteFinal, cv\_vc]

                           in all (\f -> f str) funciones

silabaValida str@[\_, \_, \_] = let funciones = [consonanteFinal, dosConsonantesInicio]

                               in all (\f -> f str) funciones

silabaValida str@[\_, \_, \_, \_] = let funciones = [consonanteFinal, dosConsonantesInicio, ccvv]

                               in all (\f -> f str) funciones

silabaValida \_ = False

Las sílabas harán unas comprobaciones u otras dependiendo de su longitud. Por ejemplo la única manera de que una sílaba e una letra sea válida es que la letra sea una vocal.

Un ejemplo de función que se aplica en sílabas de dos, tres y cuatro letras es:

--Devuelve false si la ultima letra es consonante y distinta de l,n,s,r

consonanteFinal :: String -> Bool

consonanteFinal str = let ult = last str in not(esConsonante ult && not(ult `elem` "lnsr"))

Las reglas de sílabas que se utilizan son:

* Una sílaba solo puede acabar en consonante si es ‘l’, ’n’, ’s’ o ‘r’.
* …….

Ejecutable

Por lo tanto ya tenemos dos funciones que engloban las dos tareas que queríamos hacer:

generaPalabrasNum nos genera todas la posibles palabras de longitud N con unas letras que le pasamos como argumento y la función palabraValida que nos hace de filtro para comprobar si una palabra es válida. Combinando ambas funciones definimos:

--Dada una lista de letras te da todas las posibles palabras validas de longitud n

procesarNum :: Integer -> String ->  [String]

procesarNum n str = sustListaALetras(filter palabraValida (sustListaACod (generaPalabrasNum n str)))

Esta función nos dará una lista de todas las posibles palabras de longitud N que se pueden formar con las letras del String que le pasemos como argumento, por lo que solo queda hacer una sencilla interfaz para el usuario.

El funcionamiento de la interfaz es muy sencillo:

1. Primero se imprimirá un mensaje de bienvenida y una breve explicación al usuario.
2. Luego se hará un bucle infinito en la que en cada iteración el usuario escribe las letras que tiene tanto en su mano como en el tablero para que las que se le presenta un resultado o escribe ‘salir’ para finalizar el programa
3. Una vez se tienen las letras con las que hay que generar palabras el programa genera todas las posibles palabras de longitud máxima; si no encuentra ninguna palabra de esa longitud se restará uno a la longitud de búsqueda, y así sucesivamente hasta que encuentre resultados o se llegue a longitud 1 (no existen palabras de longitud 1); si encuentra resultados los imprime.
4. Si en el paso anterior se han encontrado resultados se le da al usuario la opción de buscar más palabras, en cuyo caso se hará un búsqueda de palabras con una longitud menor a la de las palabras ya presentadas.

# Limitaciones

En un principio se iban a presentar todos los resultados posibles, pero a la hora de probarlo el programa consumía mucho tiempo de ejecución a la hora de calcular todas las posibles combinaciones de letras, por lo que se decidió limitar los resultados a una cierta longitud. Aun así si el usuario introduce demasiadas letras, el programa asume una gran carga computacional a la hora de generar todas las posibles combinaciones de letras.

# Testing