Laboratorio de Programación Funcional 2019
Analizador de GIFT*

La tarea consiste en implementar un analizador de cuestionarios. Un cuestionario es una colección de ejercicios de autoevaluación que pueden ser importados por una plataforma moodle (como EVA) para ser usados en los cursos. Los cuestionarios se escriben en texto plano usando el formato GIFT* que introduciremos, que es una versión simplificada de GIFT, un lenguaje de especificación de ejercicios de moodle. El analizador debe permitir la salida en dos formatos diferentes: HTML, y LaTeX.

Algunos términos usados en este documento: Blanco o espacio es el caracter reconocido por la función `isSpace`. **Línea en blanco** es un renglón que solamente tiene blancos.

1 Sintaxis de GIFT*

El formato GIFT* es usado para escribir los ejercicios de autoevaluación de EVA. Este formato es una variante del formato GIFT documentado en `docs.moodle.org/all/es/Formato_GIFT`. Cada archivo consta de una lista de ejercicios separados por líneas en blanco. A su vez, cada ejercicio tiene un nombre y un cuerpo rodeados por comentarios. Cada comentario ocupa una línea prefijada por `"//"`. El nombre aparece en una línea nueva inmediatamente antes del cuerpo, parentizado entre dos listas `"::"`. El cuerpo consiste de una lista de preguntas y respuestas. Las preguntas son textos que pueden contener fragmentos matemáticos y de código. Las respuestas, que siempre aparecen parentizadas por llaves, toman distintos aspectos dependiendo de si son preguntas de múltiple opción, de desarrollo, o de falso/verdadero.

El resto de la sección describe la sintaxis de GIFT* usando EBNF.

De la sintaxis EBNF. Los no terminales se escriben con paréntesis angulares, como en ⟨*ejercicio*⟩, ⟨*nombre*⟩, etc. Los terminales se escriben entre comillas simples. Las listas se escriben usando llaves {...}, no se confunda las llaves usadas para representar listas con las llaves `'{'` y `'}'` usadas como terminales en ⟨*respuesta*⟩. Todas las listas que aparecen en esta gramática son no vacías.

De los no terminales. El no terminal ⟨*ident*⟩ es una tira de letras, dígitos, blancos y puntos. Los no terminales ⟨*texto*⟩, ⟨*math*⟩ y ⟨*code*⟩ son tiras arbitrarias de caracteres. El no terminal ⟨*linea*⟩ es una tira de caracteres que no tiene fin de línea. El no terminal ⟨*blancos*⟩ es una tira de blancos.

Gramática.

⟨*cuestionario*⟩ ::= { ⟨*ejercicio*⟩ }
⟨*ejercicio*⟩ ::= { ⟨*comentario*⟩ } ⟨*nombre*⟩ ⟨*cuerpo*⟩ { ⟨*comentario*⟩ }
⟨*comentario*⟩ ::= `'//'` ⟨*linea*⟩ `'\n'`
⟨*nombre*⟩ ::= `'::'` ⟨*ident*⟩ `'::'`
⟨*cuerpo*⟩ ::= { ⟨*pregunta*⟩ | ⟨*respuesta*⟩ }
⟨*pregunta*⟩ ::= { ⟨*fragmento*⟩ }
⟨*respuesta*⟩ ::= `'{}'` | `'{'` ⟨*blancos*⟩ `'}'` | `'{'` ⟨*fv*⟩ `'}'` | `'{'` { ⟨*opcion*⟩ } `'}'`
⟨*fv*⟩ ::= `'T'` | `'V'` | `'TRUE'` | `'VERDAD'` | `'VERDADERO'` | `'F'` | `'FALSE'` | `'FALSO'`
⟨*fragmento*⟩ ::= ⟨*texto*⟩ | `'$'` ⟨*math*⟩ `'$'` | `'‘'` ⟨*code*⟩ `'’'`
⟨*opcion*⟩ ::= `'='` ⟨*fragmento*⟩ | `'~'` ⟨*fragmento*⟩

Ejemplo. El siguiente es un formulario GIFT* con seis preguntas.

```
// ejemplo de verdadero falso
::fv.1:: Grant murió en 1886 { FALSO }
// fin de pregunta

// ejemplo de multiple opcion
::mo.1:: {=Grant ~el gato de Grant ~Tumbledore} está sepultado en la
tumba de Grant.
// fin de pregunta

// ejemplo de multiple opcion
::mo.2:: Indique quién está sepultado en la tumba de Grant.{
   =Grant
   ~el gato de Grant
   ~Tumbledore
   }
// fin de pregunta

// ejemplo de desarrollo
::desarrollo.1:: Cuente acerca de las becas que recibió Grant como estudiante. {}
// fin de pregunta

// ejemplo de respuesta corta
::corta.1:: ¿Quién está sepultado en la tumba de Grant?{=Grant =Ulysses
S. Grant =Ulysses Grant}
// fin de pregunta

// ejemplo de respuesta corta
::corta.2:: {=Grant =Ulysses S. Grant =Ulysses Grant} está sepultado en
la tumba de Grant.
// fin de pregunta

//
::ejemplo del enunciado:: Complete el programa ‘foo x = "Si un quilo de papas
cuesta $" ++ x ++ " entonces dos quilos cuestan" ++ ... ‘ {}.
¿Cuál es el tipo de ‘foo‘?
{~‘Papa -> Papa -> Papa‘ ~‘$ -> $‘ ~$ -> $ =String -> String}
//
```

Al importarlo en EVA obtenemos una serie de preguntas para las cinco primeras (Figura 1). La última no se visualiza correctamente porque no responde al formato GIFT hoy aceptado, aunque sí responde al formato GIFT* que usted debe reconocer.

2 ¿Qué es un analizador (parser)?

Un analizador sintáctico (parser) es un programa que inspecciona una secuencia de entrada, decidiendo si ésta es admisible de acuerdo a un lenguaje formal. Los parsers se implementan combinando parsers más simples, que reconocen terminales o producciones particulares. En este contexto, un parser es una función que reconoce al primer elemento significativo de la entrada. Por ejemplo, un parser que lee un entero desde una entrada formada por una tira de caracteres puede tener el siguiente tipo

```
getEntero :: String -> Maybe (Integer, String)
```

y cumplir las siguientes igualdades

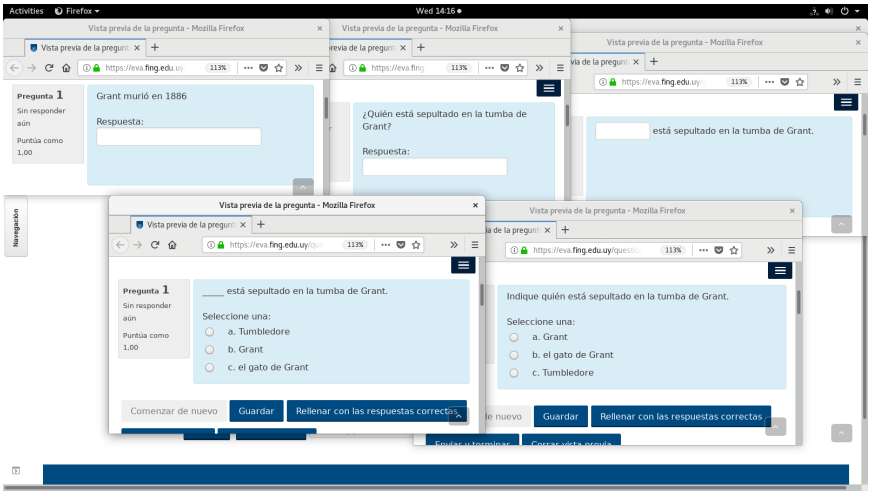

Figure 1: Preguntas desplegadas en EVA

```
getEntero "2134fjdhksjgsh"    == Just (2134, "fjdhksjgsh")
getEntero "2134 23fjdhksjgsh" == Just (2134, " 23fjdhksjgsh")
getEntero "j121dhksjgsh"      == Nothing
```

En esta tarea se construirán analizadores con dos tipos de entradas: usando como entrada un string y usando como entrada una lista de strings.

3 Etapa 1. Lectura por líneas del archivo de entrada.

En la primera parte de la tarea se lee un archivo línea por línea y se construye un `Cuestionario`. Para esta tarea se proporcionan (total o parcialmente) distintos analizadores.

Para la realización de esta primera parte se proporcionan los siguientes archivos.

`Tipos.hs` Declaración de tipos.

`Main1.hs` Programa principal.

`Salida.hs` Publicación de los cuestionarios.

`Etapa1.hs` Funciones del analizador. *Debe completarlo.*

En el archivo `Tipos.hs` se encuentra la siguiente declaración de tipos, que es la que debe usar en esta etapa.

```
type Nombre          = String
type Cuestionario a  = [ Ejercicio a ]
type Cuerpo        a = [ a ]
data Ejercicio     a = Ejercicio Comentario Nombre (Cuerpo a) Comentario
data Comentario      = COM [ String ]
```

Además, en `Tipos.hs` se encuentra la definición de la clase `CCuerpo a`. Su uso se aclarará en la Etapa 2 del laboratorio.

Para reconocer el `Cuestionario` usamos los analizadores `getCuestionario`, `getEjercicio`, `getEjercicios`, `getComs`, `getNombre`, y `getCuerpo`. Los analizadores `getCuerpo` y `getNombre` se explicitan al mostrar las funciones auxiliares `leeMX` y `leeMO`.

A continuación comentamos el analizador `getEjercicio` que está implementado en `Etapa1.hs`. En cambio, el analizador `getEjercicios` es uno de los programas que debe implementar.

```
getEjercicio :: (CCuerpo a) => [ String ] -> Maybe (Ejercicio a, [ String ])
getEjercicio xs =  do (c1 , ys)     <- getComs   xs
                      (nm , ws)     <- getNombre ys
                      (qas, zs)     <- getCuerpo ws
                      (c2 , tl)     <- getComs   zs
                      return (Ejercicio c1 nm qas c2, skipNLs tl)
```

A partir de una lista de renglones, este analizador almacena los comentarios iniciales del ejercicio en `c1`, el nombre del ejercicio en `nm`, y así sucesivamente con cada componente de un ejercicio. La notación `do` nos permite trabajar cómodamente con los distintos analizadores, que devuelven computaciones en la mónada `Maybe`. Al resto de la lista de renglones que queda por analizar se le eliminan las líneas que solamente contienen blancos usando la función `skipNLs` (que no hay que implementar como parte de la tarea).

De no usar esta notación, el programa tomaría el siguiente aspecto:

```
getEjercicio xs = case (getComs xs) of
                   Nothing       -> Nothing
                   Just (c1, ys) ->
                    case (getNombre ys) of
                     Nothing        -> Nothing
                     Just (nm, ws)  ->
                      case (getCuerpo ws) of
                       Nothing        -> Nothing
                       Just (qas, zs) ->
                        case (getComs zs) of
                         Nothing       -> Nothing
                         Just (c2, tl) -> Just (Ejercicio c1 nm qas c2, skipNLs tl)
```

3.1 Ejemplo.

Consideremos la entrada con ocho renglones

```
// ejemplo de verdadero falso
::fv.1:: Grant murió
en 1886 { Falso }
///////
// fin de pregunta

// ejemplo de respuesta corta
::corta.1:: ¿Quién está sepultado en la tumba de Grant?{=Grant =Ulysses
```

El resultado de aplicar `getEjercicio` al ejemplo (visto como lista de renglones donde no aparece `\n`) debe ser `Just (b, out)` donde:

```
b == Ejercicio (COM [" ejemplo de verdadero falso"])
               "fv.1"
               "Grant murió\nen 1886 { Falso }"
               (COM ["     ", " fin de pregunta"])
out == [ ""
       , "// ejemplo de respuesta corta"
       , "::corta.1:: ¿Quién está sepultado en la tumba de Grant?{=Grant =Ulysses"
       ]
```

3.2 Tareas para la Etapa 1

Debe programar los fragmentos `undefined` del archivo `Etapa1.hs` proporcionado, pudiendo usar las funciones auxiliares que se proporcionan, y sin modificar la sección de importaciones. Tampoco debe modificar ninguno de los demás archivos, dado que éstos no se entregarán.

Se garantiza que dentro de cada ejercicio no aparecen líneas en blanco.

Su implementación debe cumplir los siguientes requisitos.

1. La entrada puede constar de un cuestionario sin preguntas.
2. Pueden aparecer líneas en blanco al comienzo y al final del archivo.
3. Dos ejercicios consecutivos deben estar separados por una o más líneas en blanco.
4. Todos los comentarios inmediatamente anteriores al cuerpo de un ejercicio se devuelven en un único `Comentario`. Cada renglón de comentario se modifica, eliminando las primeras dos barras `"/"` del renglón, y reemplazando las barras inmediatamente contiguas por `" "`. Por ejemplo,

```
getComs (["//////",
          "////inicio␣//␣de␣pregunta",
          "::pepe::␣aca␣hay␣texto..."])
    == Just (COM ["␣␣␣␣", "␣␣inicio␣//␣de␣pregunta"],
             ["::pepe::␣aca␣hay␣texto..."])
```

5. Análogo al anterior para los comentarios inmediatamente posteriores.

3.3 Funciones auxiliares de lectura con marcas

Hemos usado funciones auxiliares que particionan una lista en tres de acuerdo a si encontramos o no determinada marca en ella. La primera parte será el máximo prefijo de la lista sin marca, la segunda parte es el primer elemento de la lista que está marcado, y la tercera el resto de la lista. Ser una marca es una función booleana (o propiedad) de los elementos de la lista. Un caso parecido es el resuelto por las funciones `span` y `break`.

En esta sección mostramos y ejemplificamos el uso de dos de esas funciones auxiliares, `leeMX` y `leeMO`.

Observemos la siguiente implementación de una función que lee hasta encontrar una Marca que eXigimos que esté presente:

```
leeMX :: (a -> Bool) -> [a] -> Maybe ([a], a, [a])
leeMX m    []  = Nothing
leeMX m (x:xs)
   | m x       = Just ([], x, xs)
   | otherwise = case (leeMX m xs) of
                      Nothing          -> Nothing
                      Just (ws, z, zs) -> Just (x:ws, z, zs)
```

O la siguiente, que es la misma usando la notación `do`.

```
leeMX m    []  = Nothing
leeMX m (x:xs)
   | m x       = Just ([], x, xs)
   | otherwise = do (ws, z, zs) <- leeMX m xs
                    return (x:ws, z, zs)
```

La función tiene por recorrido un tipo `Maybe`; si la marca no aparece en la lista, devuelve `Nothing`.

Otra función semejante, pero donde la lectura de la marca es Opcional, es la siguiente (incompleta):

```
leeMO :: (a -> Bool) -> [a] -> ([a], Maybe a, [a])
leeMO m []     = ([], Nothing, [])
leeMO m (x:xs) = undefined
```

Observemos que el tipo que se devuelve no es `Maybe`, así que en este caso no podemos usar la notación `do`. Esta función aparece así en `Etapa1.hs` y usted debe, como parte del laboratorio, reemplazar la expresión `undefined` adecuadamente.

Las funciones `leeMX` y `leeMO` podrían ayudarnos en este laboratorio. Veamos como ejemplos dos funciones que servirán para el laboratorio.

Ejemplo 1. Agrupar el cuerpo de un ejercicio La función `getCuerpo` recibe una lista de renglones y devuelve un par cuyo primer componente es la concatenación de todos los renglones previos al primer comentario, y su segundo componente el resto de la entrada que aún queda por analizar. Esta función es declarada en la clase `CCuerpo` que usamos para segmentar mejor la presentación del laboratorio. Al instanciar `Char` como un tipo de la clase `CCuerpo`, el tipo de la función `getCuerpo` es

```
getCuerpo :: [ String ] -> Maybe ( Cuerpo Char , [ String ])
```

y presenta el siguiente comportamiento:

```
getCuerpo ["lin 1    ", "       ", "lin 3", "//// com", ...]
   == Just ("lin 1    \n       \nlin 3", ["//// com", ...])
```

Mostramos dos posibles implementaciones, una de ellas usando la siguiente instancia

```
leeMX :: (String -> Bool) -> [ String ]
                  -> Maybe ([ String ] , String , [ String ])
```

La primera implementación no usa la función auxiliar `leeMX`.

```
getCuerpo xs =
   case (break esComentario xs) of
        ( _ , [] ) ->  Nothing
        (qas, com) ->  Just (concat qas, com)
```

La segunda implementación usa la función auxiliar `leeMX`.

```
getCuerpo xs =
   do (qas, z, zs) <- leeMX esComentario xs
      return (concat qas, z:zs)
```

Ejemplo 2. Lectura del nombre de un ejercicio. La función `getNombre` recibe una lista de renglones y devuelve el nombre del ejercicio que se está analizando. Este nombre se encuentra en el primer renglón de la lista, parentizado entre dos `':'`. Si no se respeta la sintaxis `::nm::mas texto ...` devuelve `Nothing`. En caso contrario, devuelve un par cuyo primer componente es el nombre del ejercicio, y su segundo componente es el resto de la entrada, eliminando el resto de la primera línea si es que en ella solamente quedan blancos.

El tipo de esta función es

```
getNombre :: [ String ] -> Maybe ( Nombre , [ String ])
```

y presenta el siguiente comportamiento:

```
getNombre ["::nom::lin", "otro renglon", ...]
   == Just ("nom", ["lin", "otro renglon", ...])
getNombre ["::nom::   ", "otro renglon", ...]
   == Just ("nom", ["otro renglon", ...])
```

Mostramos dos posibles implementaciones, una de ellas usando las siguientes instancias

```
leeMX :: ( Char -> Bool) -> String
                         -> Maybe ( String , Char , String )
leeMO :: ( Char -> Bool) -> String
                         -> ( String , Maybe Char , String )
```

La primera implementación no usa ninguna función auxiliar.

```
getNombre (':' : ':' : nmqas) : ys)
      = let (nm , qas) = break (== ':') nmqas
            f (w : ws) = if (all isSpace w) then ws else (w:ws)
        in case qas of
             _:':':zs   -> Just (nm, f (zs:ys))
             otherwise -> Nothing
getNombre _
      = Nothing
```

La segunda implementación usa las funciones auxiliares `leeMX` y `leeMO`.

```
getNombre ((':' : ':' : nmqas) : ys)
      = do (nm, _, ':':zs) <- leeMX (== ':') nmqas
           case (leeMO (not . isSpace) zs) of
                ( _ , Just mk, ws) -> return (nm, (mk:ws):ys)
                _                  -> return (nm, ys)
getNombre _
      = Nothing
```

4 Etapa 2. Análisis del cuerpo

En la segunda parte de la tarea se transformará el `Cuerpo` de cada ejercicio para interpretar las diferentes preguntas del formato GIFT*. El objetivo de esta etapa es la escritura de un analizador donde el cuerpo de los cuestionarios permita un tratamiento sencillo de los mismos.

Refinamiento. Esta etapa vuelve a inspeccionar el cuerpo de cada ejercicio. Antes, se devolvía un `String` con algunas modificaciones menores. En esta etapa analizamos esa tira de caracteres y estructuramos su contenido usando los siguientes tipos:

```
data QA           = Q Pregunta | A Respuesta
type Pregunta     = [ Fragmento ]
data Respuesta    = ESSAY | MO [ Opcion ] | FV Bool
data Opcion       = OK [ Fragmento ] | NOK [ Fragmento ]
data Fragmento    = TXT String | MATH String | CODE String
```

El `Cuerpo` es una secuencia de preguntas y respuestas. Las preguntas son `Fragmento`s que pueden tener partes de texto (`TXT`), de notación matemática (`MATH`), o de código (`CODE`). Por ejemplo, el siguiente texto

```
Complete el programa ‘foo x = "Si un quilo de papas
cuesta $" ++ x ++ " entonces dos quilos cuestan" ++ ... ‘ {}
¿Cuál es el tipo de ‘foo‘?
{~‘Papa -> Papa -> Papa‘ ~ ‘$ -> $‘ ~ $ -> $ = String -> String}
```

se almacenará como

```
[
Q [
  TXT "Complete el programa ",
  CODE "foo x = \"Si un quilo de papas\ncuesta $\" ++ x ++ \" entonces dos quilos cuestan\
A ESSAY,
Q [
  TXT "¿Cuál es el tipo de ",
  CODE "foo",
  TXT "?"],
A (MO [
      NOK [CODE "Papa -> Papa -> Papa"],
      NOK [CODE "$ -> $"],
      NOK [MATH " -> "]
      OK  [TXT " String -> String"] ] ) ]
```

Clase de tipos. En `Tipos.hs` aparece la definición siguiente:

```
type Cuerpo a = [ a ]
```

y en consecuencia los tipos `Cuestionario` y `Ejercicio` también están parametrizados. En la primera etapa de la tarea hemos usado como parámetro el tipo `Char`, y ahora refinaremos esa implementación usando el tipo `QA`.

Hemos definido una clase `CCuerpo` para implementar dicho refinamiento, cambiando solamente el Contenido del Cuerpo. Esta clase tiene una única función

```
getCuerpo :: [ String ] -> Maybe ( [ a ] , [String])
```

que se encarga de leer el cuerpo de un ejercicio dependiendo de la instancia particular; en la etapa anterior `Char`, en esta `QA`.

En `Etapa1.hs` se declaró que `Char` es una instancia de `CCuerpo`

```
instance CCuerpo Char where
   getCuerpo xs = do (qas, z, zs) <- leeMX esComentario xs
                     return (unlines qas, z:zs)
```

Ahora se agrega la siguiente instancia:

```
instance CCuerpo QA where
   getCuerpo xs = do (ys, zs) <- getCuerpo xs
                     qas <- str2qas (ys::Cuerpo Char)
                     return (qas, zs)
```

Para obtener el cuerpo de un ejercicio usando el tipo `QA` se realiza lo siguiente: se obtiene dicho cuerpo usando `getCuerpo` para `Char`, y se le aplica la función `str2qas` que se ocupa de la transformación. Este código sabe a qué instancia se está refiriendo el uso de `getCuerpo` gracias a que se explicitó que la salida `ys` debía tener tipo `Cuerpo Char`. En el programa principal se ha usado `c::Cuestionario Char` y `c::Cuestionario QA` de igual forma.

4.1 Otras características de los textos `QA`

Preste atención a las siguientes observaciones.

Fragmentos para matemáticas y códigos. Cada fragmento `MATH` comienza y termina con la marca `'$'`. Cada fragmento `CODE` comienza y termina con la marca `''`. Los caracteres contenidos entre estas marcas se almacenan en el `String` del `Fragmento` correspondiente.

Fragmentos de texto. Cada parte de un `String` que no sea fragmentos para matemáticas o código, es un fragmento de texto.

Espacios en blanco. Puede suceder que en este momento le queden el tratamiento de los blancos genere cosas como `[ TXT "  ", MATH "2+2=4 ", TXT ""]`. Nos ocuparemos de esto en la etapa siguiente.

Caracter de escape. Se dispone del caracter de escape `'\'`. Este caracter

- no sirve de escape dentro de fragmentos matemáticos y de código
- en los fragmentos de texto, el escape permite usar los siguientes caracteres: `{}$‘=~`

Formas de `Respuestas`. En esta etapa nos preocupa identificar las siguientes formas de respuesta representadas en el tipo: `ESSAY`, `MO`, `FV`. Esta identificación surge de analizar el texto que aparece entre las llaves que delimitan cada respuesta.

Ensayos Cuando el texto de la respuesta solamente contiene blancos nos encontramos ante un `ESSAY`.

Falso/Verdadero Cuando el único texto de la respuesta es solamente alguna de las palabras definidas en la sintaxis de $\langle fv \rangle$, eventualmente rodeadas por blancos, nos encontramos ante un `FV`. El valor `Bool` asociado es el que adecuadamente corresponde.

Múltiple opción Cuando el texto es una secuencia de fragmentos antecedidos por `=` o `~`, nos encontramos ante una pregunta `MO`, donde cada fragmento se corresponde con una opción correcta (`OK`) si es antecedida por `=`, y con una opción incorrecta (`NOK`) si es antecedida por `~`.

4.2 Tareas para la Etapa 2

Debe programar los fragmentos `undefined` del archivo `Etapa2.hs` proporcionado, pudiendo usar las funciones auxiliares que se proporcionan, y sin modificar la sección de importaciones.

Su implementación debe cumplir los siguientes requisitos.

1. La función `str2qa` debe convertir la tira que servía de `Cuerpo Char` en la etapa anterior en un `Cuerpo QA` que respete las observaciones planteadas en las secciones previas.

2. En caso de que la entrada no respete la sintaxis ni las anteriores observaciones, devuelve `Nothing`.

4.3 Más funciones auxiliares

4.3.1 Lectura con marcas

En esta sección comentamos acerca de dos funciones semejantes a `leeMX` y `leeMO` con el siguiente tipado:

```
leeMXE :: (a -> Bool) -> (a -> Bool) -> [a] -> Maybe ([a],       a, [a])
leeMOE :: (a -> Bool) -> (a -> Bool) -> [a] -> Maybe ([a], Maybe a, [a])
```

La invocación `leeMXE m e xs` busca la primera ocurrencia de la marca `m` en `xs` que no esté antecedida inmediatamente por el escape `e`. Debe cumplir, por ejemplo,

```
leeMXE (== 'n') (== 'e') "cenenbcsnn"   == Just ("cenenbcs"  , 'n', "n")
leeMXE (== 'n') (== 'e') "cenenbcsenn"  == Just ("cenenbcsen", 'n', "")
leeMXE (== 'n') (== 'e') "cenenbcsenen" == Nothing
```

La invocación `leeMOE m e xs` busca la primera ocurrencia de la marca `m` en `xs` que no esté antecedida inmediatamente por el escape `e`. Debe cumplir, por ejemplo,

```
leeMOE (== 'n') (== 'e') "cenenbcsnn"     == Just ("cennennbcs", Just 'n', "n")
leeMOE (== 'n') (== 'e') "cenenenenbcsn"  == Just ("cennennennennbcs", Just 'n', "")
leeMOE (== 'n') (== 'e') "enen"           == Just ("enen", Nothing , "")
```

Observe que el caracter de escape presenta una nueva complejidad. Si ese caracter aparece al final de la entrada, esa entrada es incorrecta. En consecuencia, se deben cumplir las siguientes ecuaciones[1].

```
leeMXE (== 'n') (== 'e') "cenenbcsnne"   == Just ("cenenbcs", 'n', "ne")
leeMXE (== 'n') (== 'e') "cenenbcsenene" == Nothing
leeMOE (== 'n') (== 'e') "cenenbcsnne"   == Just ("cenenbcs", Just 'n', "ne")
leeMOE (== 'n') (== 'e') "cenenbcsenene" == Nothing
```

En el archivo `Etapa2.hs` se proporcionan algunos fragmentos para implementar estas funciones.

4.3.2 Iteración de analizadores

En esta sección comentamos acerca de la función `getSeq` que puede usarse para iterar analizadores. Vemos como ejemplo su uso al analizar una `Pregunta`. Recordemos que

```
type Pregunta = [ Fragmento ]
```

Para analizar una secuencia de `Fragmento`s me basta usar un analizador de `Fragmento` como argumento de `getSeq`. El programa y las declaraciones de tipos relevantes son:

```
  ---- str2q procesa una Pregunta.
getFragmento :: String -> Maybe ( Fragmento , String )
str2q        :: String -> Maybe ( Pregunta  , String )
str2q = getSeq getFragmento
```

En el archivo `Etapa2.hs` se proporciona el código de `getSeq`. Puede usar la misma, aunque su uso no es obligatorio.

[1] Debido a esto es el tipo devuelto por `leeMOE`.

Observación. El siguiente "programa" ni siquiera compila.

```
str2qTrucho = getSeq . getFragmento
```

Tenga siempre en cuenta la diferencia entre la aplicación de una función a un argumento y la composición de funciones.

4.4 Integración con la Etapa 1

El archivo `Etapa1.hs` completado en la etapa anterior es usado en esta etapa.

El archivo `Main2.hs` de esta etapa se diferencia del `Main1.hs` de la anterior en los siguientes aspectos:

1. además de `Etapa1.hs` también se importa `Etapa2.hs`; y
2. el `Cuestionario` está instanciado por el tipo `QA` en lugar de `Char`.

5 Etapa 3. Transformaciones.

Como lectura exitosa de un archivo GIFT* se obtiene un `Cuestionario`. Debe transformar este cuestionario con una serie de acciones que se explican en esta sección.

Acerca de las transformaciones. Todos las funciones de esta etapa se deben implementar en el archivo `Etapa3.hs` y tienen el tipo

```
t1, t2, ... tn :: Cuestionario QA -> Cuestionario QA
```

La composición de las mismas es una nueva transformación

```
transformaciones :: Cuestionario QA -> Cuestionario QA
transformaciones = t1 . t2 . ... . tn
```

5.1 Las transformaciones requeridas.

Transformación `mo2short`. Tipo `Short` Recuerde la definición

```
data Respuesta = ESSAY | MO [ Opcion ] | FV Bool | SHORT [ Opcion ]
```

Hasta ahora se han usado los primeros tres constructores en la implementación. La transformación `mo2short` toma como argumento un `Cuestionario QA` obtenido en la Etapa 2, que sabemos que no usa el constructor `SHORT`, y devuelve un nuevo `Cuestionario QA`. Cada aparición de una respuesta

```
MO [ OK op1, OK op2, OK op3, ..., OK opn ]
```

donde todas las opciones son correctas, debe ser reemplazada por

```
SHORT [ OK op1, OK op2, OK op3, ..., OK opn ]
```

El constructor `SHORT` se usa para aquellas respuestas `MO` espúreas, en el sentido de que todas sus respuestas son correctas. Esta transformación modifica el `Cuestionario` de forma que no queden respuestas `MO` espúreas.

Transformación `sortMO`. Ordenar opciones Una pregunta múltiple opción tiene opciones correctas e incorrectas que aparecen en cualquier orden. La transformación `sortMO` toma como argumento un `Cuestionario QA`, y devuelve un nuevo `Cuestionario QA` donde las opciones aparecen ordenadas. Por ejemplo, cada respuesta del `Cuestionario QA` de entrada de la siguiente forma

```
MO [ NOK op1, OK op2, NOK op3, OK op4 ]
```

será reemplazada por

```
MO [ OK op2, OK opn, NOK op1, NOK op3 ]
```

Todas las opciones correctas deben anteceder a las incorrectas, preservando el orden relativo entre las correctas y las incorrectas.

Transformación `trim`. Eliminar espacios. Los `Fragmento`s tienen una tira de caracteres, en ocasiones con blancos en los extremos.

La transformación `trim` toma como argumento un `Cuestionario QA`, y devuelve un nuevo `Cuestionario QA` en que cada uno de los fragmentos fue reemplazado por otro idéntico, salvo que no tiene espacios al comienzo ni al final. Además, deben eliminarse todos los fragmentos que solamente contengan espacios.

Por ejemplo, si en el cuestionario de entrada aparece

```
OK [TXT  " La  suma ", MATH "  2+ 2", TXT " es  ", MATH "   4   ", TXT "   "]
```

en el cuestionario de salida debe aparecer

```
OK [TXT  "La  suma", MATH "2+ 2", TXT "es", MATH "4"]
```

Transformación `nodupMO`. Eliminación de opciones duplicadas.

> Para que esta transformación sea más sencilla se supondrá que el cuestionario de entrada ya ha sido transformado con `trim`.

En una pregunta MO puede aparecer repetida la misma opción. Decimos que dos opciones son la misma cuando:

1. sus constructores `OK` o `NOK` coinciden;
2. los fragmentos `TXT` coinciden salvo por la cantidad de espacios (más que cero) entre palabras;

Por ejemplo, estas opciones son las mismas:

```
OK [TXT "La  suma"  , MATH "2+ 2"]
OK [TXT "La    suma", MATH "2+ 2"]
```

Pero las siguientes son distintas:

```
OK [TXT "La s uma", MATH "2+ 2"]
OK [TXT "La  suma", MATH "2+ 2"]
OK [TXT "La  suma", MATH "2 + 2"]
```

La transformación `nodupMO` toma como argumento un `Cuestionario QA` (que haya sido procesado por `trim`), y devuelve un nuevo `Cuestionario QA` que no tiene opciones duplicadas en las preguntas `MO`. En caso de haber duplicaciones, se deja la primera ocurrencia de la misma.

Por ejemplo, si en el cuestionario de entrada aparece

```
MO [OK [TXT "uno", MATH "1"], OK [TXT "dos", MATH "1"], OK [TXT "uno", MATH "1"]]
```

en el cuestionario de salida debe aparecer

```
MO [OK [TXT "uno", MATH "1"], OK [TXT "dos", MATH "1"]]
```

Transformación `filtroFV`. Respuestas FV variadas. Cada cuestionario tiene un conjunto de respuestas `FV`; si hay más de una de estas respuestas, pero todas son verdaderas o todas falsas, nos encontramos frente a un cuestionario *pobre en `FV`*.

La transformación `filtroFV` toma como argumento un `Cuestionario QA`. Si el mismo es pobre en `FV`, devuelve un nuevo `Cuestionario QA` donde se han eliminado todas las respuestas `FV` de la entrada. Si por el contrario, el argumento no es pobre, se devuelve el mismo `Cuestionario QA` como salida.

Observe que un cuestionario con una única respuesta `FV` no es pobre. De igual forma, un cuestionario sin respuestas `FV` tampoco es pobre.

Se pide. Implemente las funciones

```
mo2short  :: Cuestionario QA -> Cuestionario QA
sortMO    :: Cuestionario QA -> Cuestionario QA
trim      :: Cuestionario QA -> Cuestionario QA
nodupMO   :: Cuestionario QA -> Cuestionario QA
filtroFV  :: Cuestionario QA -> Cuestionario QA
```

6 Escritura

La última etapa es la de reescribir el cuestionario en formato HTML o LaTeX. El archivo `Salida.hs` muestra cómo hacerlo. En esta etapa usted ya no tiene nada que programar.

Se proporcionan dos archivos `Main2HTML.hs` y `Main2LaTeX.hs` que integran todas las etapas anteriores para implementar el analizador de cuestionarios con salida a HTML y LaTeX respectivamente.

7 Se Pide

En suma, se pide completar las partes `undefined` de los archivos `Etapa1.hs`, `Etapa2.hs` y `Etapa3.hs`, de manera que el analizador funcione correctamente.