PROGRAMACIÓN DECLARATIVA G. INFORMÁTICA CURSO 2018-19
PRÁCTICA INDIVIDUAL

Objetivo de la práctica: Escribir un programa en Haskell que implemente el método de deducción de los *tableaux* para la lógica proposicional.

Descripción del problema: Una fórmula de la lógica proposicional puede ser atómica:

- Símbolos de proposición $p,q,r\ldots$,
- Las constantes lógicas $\top$ (cierto) $\bot$ (falso).

O bien compuesta:

- Negación ($\neg$) de una fórmula.
- Conjunción ($\wedge$) de dos fórmulas.
- Disyunción ($\vee$) de dos fórmulas.

Por ejemplo, las siguientes son fórmulas de la lógica proposicional:

$f_1 \equiv (p \wedge \neg q) \vee \neg p$
$f_2 \equiv \neg\bot \vee (p \wedge \neg(q \vee \neg q))$
$f_3 \equiv \top \wedge q \wedge (\neg q \vee r)$

Para procesar en Haskell fórmulas de esta lógica partimos de la siguiente definición de tipo de datos para representar fórmulas:

```
data  FProp =  Cierto | Falso | P String |  No FProp | Y FProp FProp | O FProp FProp
```

Por dar un ejemplo, la fórmula f_1 correspondería a la siguiente expresión:

```
f1 = O (Y (P "p") (No (P "q"))) (No (P "p"))
```

¿Qué se debe implementar?

En primer lugar, una función `tableau_cerrado` que, dado un conjunto de fórmulas, devuelva cierto o falso según sea posible o no encontrar un *tableau* cerrado para dicho conjunto (para ello harán falta otras auxiliares, claro). A continuación, basándose en la función anterior, se deben programar las siguientes funciones:

- `tautologia`: reconoce si una fórmula es una tautología o no. Es decir, si su negación tiene un *tableau* cerrado y por tanto es insatisfactible.
- `consecuencia`: reconoce si una fórmula φ es consecuencia lógica de un conjunto de fórmulas Φ. Es decir, si es posible encontrar un *tableau* cerrado para $\Phi \cup \{\neg\varphi\}$.
- `equivalentes`: reconoce si dos fórmulas φ_1 y φ_2 son lógicamente equivalentes. Es decir, la fórmula $(\neg\varphi_1 \vee \varphi_2) \wedge (\neg\varphi_2 \vee \varphi_1)$ es una tautología.

Parte básica:

- Programar las funciones anteriores, atendiendo además a las indicaciones que siguen.
  - Hay que declarar los tipos de todas las funciones que se programen, incluidas las funciones auxiliares que pudieran necesitarse.
  - Se pueden usar todas las funciones de `Prelude`, es decir, las que se cargan con el sistema, pero no se puede importar ningún otro módulo, lo que quiere decir que, por ejemplo, si usan operaciones con listas que no están en `Prelude` hay que programarlas.
  - Para poder realizar ejemplos y evaluar la práctica, deben incluirse al menos cinco fórmulas proposicionales concretas (`f1,f2,f3,f4,f5`), definidas mediante funciones de aridad 0 (al estilo de la `f1` de más arriba). Las tres primeras, `f1,f2,f3` deben corresponder a las fórmulas f_1, f_2, f_3 de arriba.
- Declarar `FProp` como instancia de las siguientes clases de tipos:
  - Como instancia de la clase `Eq`, haciendo que la igualdad entre fórmulas coincida con la igualdad estructural (es decir, componente a componente), salvo por el hecho de que el orden en conjunciones o disyunciones no importe. Por ejemplo, la fórmula $\neg(p \wedge (p \vee r))$ sería igual a la fórmula $\neg((r \vee p) \wedge p)$.
  - Como instancia de la clase `Ord`, de modo que una fórmula φ sea menor que otra φ' si φ' es consecuencia lógica de φ.

Parte opcional:
Programar una pequeña interacción con el usuario, de modo que se pida al usuario que introduzca las fórmulas, se le pregunte en un sencillo menú qué quiere hacer con ellas y muestre el resultado de lo pedido. El formato y procedimiento concreto para esta interacción se deja a criterio del programador.

Nota: Para leer las fórmulas introducidas en la interacción **no** se pretende que se haga un analizador léxico-sintáctico de fórmulas proposicionales, algo relativamente costoso y que desvía la atención del objetivo de la práctica. En su lugar, se puede utilizar el analizador que viene *de fábrica* con la función

```
read::Read a => String -> a
```

que puede convertir un `String` en el valor representado por él en un tipo `a` que esté en la clase `Read` (y un error de *parsing* si el `String` no representa correctamente ningún valor en ese tipo). Es decir, que `read` se comporta como inversa de `show::Show a => a -> String` para un tipo que esté en las clases `Show` y `Read`. Es importante saber que se puede meter un tipo en la clase `Read` usando `deriving Read` al definir el tipo (del mismo modo que se puede meter en `Show` mediante `deriving Show`). Por ejemplo, si se ha incluido `deriving Read` en la definición del tipo `FProp`, entonces `read "Y Cierto Falso"::FProp` produce la fórmula (o sea, el valor de `FProp`) `Y Cierto Falso`.

¿Qué, cómo y cuándo se debe entregar?

- La entrega se realizará a través del Campus Virtual y consistirá en un solo fichero, en el que las explicaciones irán como comentarios Haskell.
- **Fecha límite para la entrega**: **16 de enero**

¿Cuánto influye la calificación de la práctica en la calificación final?

- La nota de la práctica supone el **10 % de la nota final (1 punto)**.
- Para obtener 0,7 puntos basta dar una solución razonable y bien explicada a la parte básica. La eficiencia no es nuestra mayor preocupación. Para obtener los 0,3 puntos restantes hay que programar la interacción con el usuario.
- El trabajo es individual. La copia de otros compañeros o de cualquier otra fuente, así como facilitar la copia a otros, será severamente castigado en la calificación **global** de la asignatura. Ante las dudas, consultad con el profesor.