Facultad de Ciencias, UNAM
Práctica 3

Alejandro Hernández Mora Pablo Camacho González Luis Manuel Martínez Dámaso
Edith Araceli Reyes López Adrián Felipe Vélez Rivera

Fecha de entrega: 30 de Enero

Introducción.

En esta práctica vamos a trabajar con la creación de tipos en haskell.

Creación de tipos en Haskell.

Haskell es un lenguaje fuertemente tipado. Esto significa que, una vez que definimos de que tipo es un parámetro, este ya no puede cambiar de tipo. Antes de crear un nuevo tipo de dato en haskell, enunciamos algunas carácteristicas que puede tener un tipo:

- El tipo de dato puede ser representado con una cadena de carácteres. De esta forma podemos ver gráficamente el resultado en la terminal.
- Los elementos del nuevo tipo pueden ser comparados. Es decir, saber cuando un elemento a es igual o distinto al elemento b.
- Los elementos del tipo nuevo tienen un orden.

Vamos a contruir un tipo de dato que represente a una fórmula lógica proposicional. En las fórmulas de la lógica proposicional tenemos a las variables proposicionales como el caso base de una fórmula, que es el elemento más simple de la lógica proposicional. La palabra reservada **data** sirve para crear un nuevo tipo, seguida del nombre que le daremos al nuevo tipo. En nuestro caso, el nuevo tipo se va a llamar **Var**. Después de eso, especificamos cuáles van a ser los elementos que pertenecen a el nuevo tipo. Para representar a éstas, usaremos las letras del alfabeto. En haskell se hace de la siguiente forma:

```
data Var = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z
```

Para poder hacer que los tipos de datos sean comparables, ordenables y que puedan tener una representación en terminal usamos la palabra reservada **deriving**, seguida de unos paréntesis con las palabras reservadas para dichas características. **Ord** para la ordenación, **Eq** para la comparación y **Show** para la representación de salida. Por lo tanto la definición queda de la siguiente forma:

```
data Var = A|B|C|D|E|F|G|H|I|J|K|L|M|N|O|P|Q|R|S|T|U|V|W|X|Y|Z deriving (Show, Eq, Ord)
```

Constructores en Haskell

En haskell podemos construir un elemento nuevo de muchas formas. A nosotros nos conviene representar las fórmulas de la manera más parecida posible a como las escribimos en la teoría. Tenemos los siguientes operadores:

- Negación ($\neg$), que se representará con el símbolo `Neg`.
- Conjunción ($\wedge$), que se representará con el símbolo `:|:`
- Disyunción ($\vee$), que se representará con el símbolo `:&:`
- Condicional ($\rightarrow$), que se representará con el símbolo `:=>:`
- Bicondicional ($\leftrightarrow$), que se representará con el símbolo `:<=>:`

Para poder representar los operadores como lo descrito anteriormente, hacemos lo siguiente:

```
data Formula = Prop Var
             | Neg Formula
             | Formula :&: Formula
             | Formula :|: Formula
             | Formula :=>: Formula
             | Formula :<=>: Formula deriving (Show, Eq, Ord)
```

Con lo anterior, creamos en el tipo **Formula**, que está definido recursivamente, dónde el elemento más simple es **Prop Var**.

Además de la sintáxis, podemos especificar la precedencia de los contructures binarios. Es decir, que en caso de no poner explícitamente paréntesis para indicar la semántica de la fórmula, podemos indicar a Haskell la representación que queremos que tenga.

Para esta práctica:

- Evaluamos por la izquierda.
- Los operadores de conjunción y disyunción, tienen la máxima jerarquía.
- El operador de bicondicional tiene mayor jerarquía que el operador condicional.

Entonces, el tipo Formula quedaría de la siguiente forma:

```
data Formula = Prop Var
             | Neg Formula
             | Formula :&: Formula
             | Formula :|: Formula
             | Formula :=>: Formula
             | Formula :<=>: Formula deriving (Show, Eq, Ord)
infixl 9 :&:
infixl 9 :|:
infixl 7 :=>:
infixl 8 :<=>:
```

Para definir la evaluación de un operador usamos la palabra reservada **infixl (evaluación izquierda), infixr (evaluación derecha)**. Está palabra recibe la jerarquía y el operador al que le vamos agregar la condición de evaluación.

Ejercicios.

Para esta práctica deberás programar algunas funciones en Haskell. Deberás usar las firmas que están en cada ejercicio.

1. **Negación**. Una función recibe una fórmula y devuelve su negación. Hay que aplicar las reglas de la negación. **(2 puntos)**

   ```
   negar :: Formula -> Formula
   ```

2. **Variables de la fórmula**. La función devuelve una lista con todas las variables de una fórmula. La lista no debe contener repetidos. **(2 puntos)**

   ```
   variables :: Formula -> [Var]
   ```

3. **Equivalencia**. La función recibe una fórmula y devuelve la fórmula equivalente sin condicionales, bicondicionales y la negación sólo está frente a variables proposicionales. **(2 puntos)**

   ```
   equivalencia :: Formula -> Formula
   ```

4. **Interpretación**. La función recibe una fórmula y una lista con parejas formadas por una variable y su valor booleano. La función debe devolver la interpretación de la fórmula usando la lista de parejas. Puedes asumir que todas las variables tienen su valor en la lista de parejas. **(2 punto)**

   ```
   interpretacion :: Formula -> [(Var,Bool)] -> Bool
   ```

5. **Tabla de Verdad**. La función calcula la tabla de verdad de una lista de variables. **(2 puntos)**

   ```
   tablaVerdad :: [Var] -> [(Var,Bool)]
   ```

6. **Tautología**. La función verifica si una fórmula es una tautología. **(1 punto)**

   ```
   tautologia :: Formula -> Bool
   ```

Entrega

Deberás entregar un archivo llamado **practica3.hs** respetando el nombre y la extensión del archivo, junto con el archivo **ReadMe.txt** conforme a los lineamientos de entrega de prácticas. La entrega es a través de la plataforma *classroom*, en la actividad asignada a la tarea. Se tiene como límite las **23:59:59** de la fecha de entrega, cada día de retraso se penalizará con puntos menos.

¡Que tengas éxito!