Facultad de Ciencias - UNAM
Lógica Computacional 2026-2
Práctica 1: Introducción a Haskell

Favio Ezequiel Miranda Perea Patricio Ordoñez Blanco Eduardo Vargas Pérez

10 de Febrero de 2025
Fecha de entrega: 19 de Febrero de 2025 hasta las 23:59

1. Introducción

Haskell es un lenguaje de programación estandarizado multi-propósito, ***puramente funcional****, con* ***evaluación perezosa*** *y memorizada, y* ***con tipado estático****. Su nombre se debe al lógico estadounidense Haskell Curry, debido a su aportación al* ***cálculo lambda****, el cual tiene gran influencia en el lenguaje.*[1]

A lo largo de este curso de Lógica Computacional se utilizará Haskell como herramienta para la implementación de los conceptos y algoritmos que se verán en la parte teórica. Los conceptos dados en la definición anterior son de suma importancia para poder comprender de mejor manera el funcionamiento de Haskell.

Puramente Funcional (Purely functional): Todas las funciones en Haskell son funciones en el sentido matemático. No hay sentencias ni instrucciones, sólo expresiones que no pueden mutar variables (locales o globales) ni acceder a estados como el tiempo o los números aleatorios.

Evalución Perezosa (Lazy Evaluation): Es una estrategia de evaluación que retrasa el cálculo de una expresión hasta que su valor sea necesario, y que también evita repetir la evaluación en caso de ser necesaria en posteriores ocasiones.

Tipado estático (Statically typed): Cada expresión en Haskell tiene un tipo que se determina en tiempo de compilación. Todos los tipos compuestos por la aplicación de la función tienen que coincidir. Si no lo hacen, el programa será rechazado por el compilador. Los tipos se convierten no solo en una forma de garantía, sino en un lenguaje para expresar la construcción de programas.

2. Desarrollo de la práctica

2.1. Tipos de datos Algebraicos

1. En el archivo base de esta práctica se incluye un tipo de dato llamado `Shape` que representa a las figuras geométricas. Este tipo de dato puede representar los *círculos, cuadrados, rectángulos, triángulos (equiláteros) y trapecios (isósceles).* Realiza lo siguiente:

   - `area :: Shape -> Float`
     Implementa la función *area* que calcula el área de una figura. **[0.5 puntos]**
   - `perimeter :: Shape -> Float`
     Implementa la función *perimeter* que calcula el perímetro de una figura dada.
     [0.5 puntos]

2. Crea un tipo de dato llamado `Point` que represente un punto en un plano cartesiano. Para esto se debe utilizar un **sinónimo** y el tipo de dato `Float`. Posteriormente, define la función `distance :: Point -> Point -> Float` para calcular las distancia entre dos puntos y la función `from0 :: Point -> Float` para calcular la distancia de un punto al origen. **[1 punto]**

3. En la isla Funcional, la cual cuenta con un área de terreno infinita, habita la tribu Haskelliums. Esta tribu tiene algunas costumbres parecidas a las nuestras, pero otras que son muy distintas. Por ejemplo:

   - Tienen un nombre, un primer apellido y un segundo apellido. Cuando tienen hijos, los padres eligen el nombre, el primer apellido del hijo es el primer apellido del primer padre y el segundo apellido es el primer apellido del segundo padre, además de que este hijo al nacer vivirá en casa de sus padres.
   - Ellos viven en casas con formas de figuras geométricas. Todas las casas son de un solo piso con paredes de dos metros y medio de alto.
   - En el centro de la isla, está la plaza que es donde trabajan todos los Haskelliums. Si ellos viven cerca de la plaza (menos de 300 u) se van en bicicleta, pero si viven lejos utilizan moto. La bicicleta se mueve con una velocidad a 30u/t, mientras que la moto a una velocidad de 70u/t.

   Conociendo toda esta información acerca de los Haskelliums, crea un tipo de dato llamado `Haskellium` que represente a un Haskellium, almacenando su ***nombre, primer apellido, segundo apellido, ubicación de su casa y las características de su casa***. Estos se deben almacenar bajo el nombre de `name,lastName1, lastName2, location y houseShape`. Para esto se debe usar los tipos de dato creados en los ejercicios anteriores (las figuras y los puntos). **[1 punto por la implementación]**

Posteriormente, implementa las siguientes funciones:

- `son :: Haskellium -> Haskellium -> String -> Haskellium`
  Dados dos Haskelliums y un String de nombre, regresa un Haskellium que sería hijo de los dos Haskelliums con el nombre dado. **[0.5 puntos]**
- `houseCost :: Haskellium -> Float`
  Dado un Haskellium, calcula las unidades necesarias para construir su casa, contemplando las paredes y el techo. Para esto se desestimará el grosor de los muros, ya que sólo nos interesa el área de las paredes. Notar que se dijo la altura de las paredes en la descripción de los Haskelliums. **[1 punto]**
- `timeToWork :: Haskellium -> Float`
  Dado un Haskellium, se calcula el tiempo en unidades t, que le cuesta llegar a su trabajo. Contemplando si este va en bicicleta o en moto. **[1 punto]**

NOTA: Para este ejercicio es importante utilizar las funciones definidas en ejercicios pasados de ser el caso.

2.2. Listas y Funciones

Ahora que se ha comprendido más a fondo el funcionamiento de los tipos de datos algebraicos, se realizarán ejercicios para poner a prueba el razonamiento lógico y el entendimiento del paradigma funcional.

1. `palindromo :: [a] -> Bool`
   Dada una lista, esta función tiene que devolver `True` si la lista es un palíndromo, `False` en otro caso. **[0.5 Puntos]**
   Ejemplos

```
ghci > palindromo "ana" ▷ True
ghci > palindromo [1,2,3,4] ▷ False
```

2. `myFoldr :: (a -> b -> b) -> b -> [a] -> b`
   Investiga cómo funciona la función `foldr` y realiza tu propia implementación. **[1 punto]**
3. `conjuntoPotencia :: [a] -> [[a]]`
   Dada una lista, esta función debe calcular el conjunto potencia de la lista ingresada.
   [1.5 Puntos]
   Ejemplo

```
ghci > conjuntoPotencia [1,2] ▷ [[1,2],[1],[2],[]]
```

   Hint: Usa recursión y listas por comprensión.

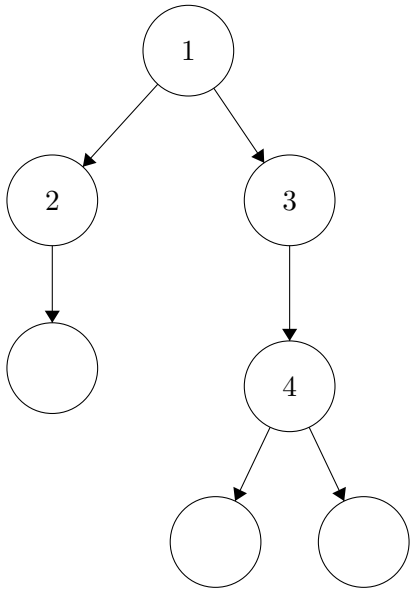
2.3. Árboles

Considera una estructura en la que un árbol puede ser cualquiera de los siguientes:

- Un nodo vacío.
- Un nodo con un elemento y un hijo (un subárbol).

- Una rama con un elemento y dos hijos (dos subárboles).

En donde se puede tener algo como el siguiente ejemplo:



Lo cual podría verse en código como *Branch* 1 $\underbrace{\textit{(Node 2 Void)}}_{\text{Subárbol izquierdo}}$ $\underbrace{\textit{(Node 3 (Branch 4 Void Void))}}_{\text{Subárbol derecho}}$.

Realiza lo siguiente:

1. Siguiendo la descripción anterior, implementa un tipo de dato algebraico *OneTwoTree a*. **[1 punto]**
2. `suma :: OneTwoTree Int -> Int`
   Implementa recursivamente la función *suma* que dado un árbol con enteros, regrese la suma de todos los elementos del árbol. **[0.5 puntos]**

¡Buena suerte a todos! ☺☺☺

Referencias

[1] Bryan O'Sullivan, John Goerzen, and Donald Bruce Stewart. *Real World Haskell: Code You Can Believe In*. O'Reilly Media, Inc., November 2008.