Fundamentos de Computación
Obligatorio

Implementación de un lenguaje imperativo

Este trabajo tiene un puntaje de 12 puntos y puede ser realizado en grupos de hasta dos estudiantes.
Cada estudiante debe subir **POR GESTIÓN** un único documento con las funciones pedidas en un archivo de haskel (.hs) que compile, antes del día 10/12/20 a las 21 hs.

El objetivo de este trabajo es programar con listas y árboles en Haskell.
Los ejercicios a realizar están indicados con los números 1), 2) ... en el texto.
Se tendrá en cuenta el estilo de programación y el nivel de abstracción utilizado en la definición de las funciones.
Recomendamos utilizar las funciones del Preludio de Haskell vistas en clase para listas, así como definir funciones auxiliares cuando sea necesario, para hacer más legible el código.

1. Descripción general

Los programas imperativos describen la computación como una serie de instrucciones que se ejecutan según un control de flujo explícito, y modifican la memoria a través de las variables.
Las variables contienen datos, pueden ser modificadas, y representan el estado del programa. La instrucción que permite modificar el valor de una variable en la memoria es la asignación, y existen construcciones que permiten combinar instrucciones para definir diferentes flujos de control (secuencial, condicional, repetición).
El objetivo de este trabajo es definir un lenguaje imperativo sencillo embebido en Haskell para escribir y ejecutar programas con expresiones enteras.

2. El lenguaje

a) Variables
Representamos a las variables como Strings:

```
type Var = String
```

b) Expresiones
Las expresiones del lenguaje se construyen a partir de las variables y constantes numéricas las cuales se representan usando los contructores **V** e **I** respectivamente.
Para las expresiones aritméticas se tienen los operadores de suma, resta y multiplicación, que se representan mediante los constructores **(:+), (:-)** y **(:*)** respectivamente.

Además, los valores booleanos se representan utilizando el **0** para representar el False y el resto de los números enteros para representar el True. A estos valores se agregan los conectivos &&, || y not, representados mediante los constructores **(:&&), (:||)** y **Not** respectivamente y la igualdad entre expresiones, representada mediante el constructor **(:==)**.
Definimos entonces el tipo de las expresiones del siguiente modo:

```
data Exp where {  V      :: Var → Exp;
                  I      :: Int →  Exp;
                  (:+)   :: Exp → Exp → Exp;
                  (:-)   :: Exp → Exp → Exp;
                  (:*)   :: Exp → Exp → Exp
                  (:&&)  ::Exp → Exp → Exp
                  (:||)  :: Exp → Exp → Exp
                  Not    :: Exp → Exp
                  (:==)  :: Exp → Exp → Exp }
```

Para facilitar la lectura y escritura expresiones se define en Haskell que **:*** tiene mayor precedencia que **:+** y **:-** y que **:&&** tiene mayor precedencia que **:||**. Además **:==** tiene la menor precedencia de todos los operadores.
De este modo, por ejemplo:
- la expresión **I 3 :+ I 5 :* I 2 :== I 13** representará la expresión “(3 + (5 * 2)) == 13”
- la expresión **Not (I 1 && I 0 :== I 1)** representará a “not ((True && False) == True)”.

c) Memoria

La memoria se representa como una lista de variables con su valor asociado:

```
type Mem = [(Var,Int)]
```

d) Evaluación de expresiones

Las operaciones que debe proveer una memoria para poder calcular el valor de las expresiones son las siguientes:

- **Lookup**: devuelve el valor de una variable en la memoria. Si la variable no fue inicializada, devuelve un mensaje de error.
- **Update:** modifica el valor de una variable en la memoria con un valor dado. Si la variable no está en la memoria, se la debe agregar a la misma con el valor dado.
- **Eval:** calcula el valor de una expresión utilizando el valor de sus variables en la memoria.

Se pide definir estas tres funciones como sigue:

1) **(@@) :: Var → Mem → Int**, que implementa la operación de lookup.
2) **upd :: (Var, Int) → Mem → Mem**, que implementa la operación de update.
3) **eval :: Exp → Mem → Int**, que implementa la operación eval.

Ejemplos:
"y" @@ [("x",1),("y",2),("z",3)] = 2
"z" @@ [("x",1),("y",2),("z",3)] = 3
"zz" @@ [("x",1),("y",2),("z",3)] = error: La variable no se encuentra en la memoria

upd ("z",7) [("x",1),("y",2),("z",3)] = [("x",1),("y",2),("z",7)]
upd ("z",7) [("x",1),("y",2)] = [("x",1),("y",2),("z",7)]
upd ("z",7) [] = [("z",7)]

eval (V "x" :+ V "y") [("x",1),("y",2),("z",3)] = 3
eval (V "x" :* I 1) [("x",1),("y",2),("z",3)] = 1
eval (Not(V "x" :&& V "z")) [("x",1),("y",2),("z",3)] = 0
eval (I 0 :|| V "z") [("x",1),("y",2),("z",3)] = 1 (o cualquier otro número distinto de 0)
eval (V "x" :== I 1) [("x",1),("y",2),("z",3)] = 1 (o cualquier otro número distinto de 0)

3. Los Programas

a) El tipo Prog

Introducimos el tipo de los programas **Prog.** Los mismos se construyen a partir de las siguientes primitivas:

- **Asignación múltiple**: permite asignar valores de expresiones a variables en forma múltiple y simultánea.
  A partir de una lista de parejas de la forma (variable,expresión), *primero* se evalúan todas las expresiones de la lista y *después* se asignan esos valores a las variables correspondientes.
  Para representar la asignación se utilizará el constructor
  Asig :: [(Var, Exp)] → Prog

  Ejemplo: **Asig [("x", I 1) , ("z", I 1 :&& I 0)]** es un programa que asigna el valor 1 a la variable x, y el valor 0 a la variable z.

- **Composición secuencial**: permite ejecutar un programa después de otro, al igual que el ";" de muchos lenguajes de programación.
  Para representar la secuencia se utilizará el operador
  (:>) :: Prog → Prog → Prog

  Ejemplo: **Asig [("z", I 1)] :> Asig [("x", V "z" :+ I 1)]** es un programa que primero asigna el valor 1 a la variable z, y después le asigna a x el valor de la variable z más 1 (o sea, 2).

- **Condicional**: permite ejecutar uno de varios programas, dependiendo de condiciones booleanas.
  A partir de una lista de parejas de la forma (condición,programa), evalúa las condiciones de izquierda a derecha hasta encontrar la primera que sea distinta de **0**, en cuyo caso prosigue a ejecutar el programa que le corresponde. En el caso de que todas las condiciones evalúen a **0**, no hace nada.
  Para representar el condicional se utilizará el constructor

  Cond :: [(Exp, Prog)] → Prog

  Ejemplo: **Asig [("x", I 0) , ("z", I 1)] :>**
  Cond [(V "x", Asig [("y" , I 1)]) ,
  (V "z", Asig [("y" , I 2)])]
  es un programa que primero asigna 0 a la variable x, y 1 a la variable z, y luego asigna 2 a la variable y (ya que el condicional entra a la segunda opción por ser z distinto de 0).

- **Ciclo**: ejecuta un programa mientras que el valor de una expresión sea distinto de cero (Verdadero).
  Para representar el ciclo se utilizará el constructor

  While:: Exp → Prog → Prog

  Ejemplo: **Asig [("z", I 10)] :>**
  While (V "z") (Asig [("z" , V "z" :- I 2)])
  es un programa que primero asigna 10 a la variable z, y luego entra a un ciclo donde decrementa el valor de z en 2 hasta llegar a 0.

Observar que, como las variables contienen valores enteros, para las primitivas **Cond** y **While**, se utiliza el **0** para representar el False y el resto de los números enteros para representar el True.

Definimos entonces el siguiente tipo:

```
data Prog where { Asig :: [(Var, Exp)] → Prog;
                  (:>) :: Prog → Prog → Prog;
                  Cond :: [(Exp, Prog)] → Prog ;
                  While :: Exp → Prog → Prog }
```

Considere los siguientes ejemplos de programas codificados como expresiones de tipo Prog. Asegúrese de comprender la intención de los mismos y pruebe su comportamiento. ¿Qué valores tienen las variables x e y luego de su ejecución con una memoria inicialmente vacía?

```
p1 :: Prog
p1 = Asig [("x" , I 1), ("y", I 1)]
     :> Cond [(V "y" :- V "x" , Asig [("z", I 10)]),
              (I 1, Asig [("z", I 0)])]

p2 :: Prog
p2 = Asig [("x",  I 27), ("y", I 5)]
     :> While (V "x") (Asig [("y", V "y" :+ I 2)] :> Asig [("x", V "x" :- V "y")])
```

b) Ejecución de programas

Como explicamos anteriormente, la ejecución de un programa modifica el valor de las variables en la memoria. Por ello tiene sentido definir la ejecución como una función que recibe una memoria y devuelve otra. Para ello pedimos:

4) Definir la función **run :: Prog** → **Mem** → **Mem**, tal que **run p m** es la memoria resultante luego de ejecutar el programa p en la memoria m.
Para ello se deberá definir el efecto de ejecutar cada constructor sobre una memoria dada.

c) Programar en Prog

Utilizando el lenguaje de programación definido mediante el tipo **Prog**, definir los siguientes programas.
Sugerencia: escribir los programas primero en pseudo-código utilizando las primitivas de Prog, y después traducirlos a la sintaxis de **Prog**.

5) Definir el programa **swap :: Prog** que intercambia los valores de las variables **x** e **y** en una memoria. Pruébelo en la memoria resultante luego de ejecutar p1.

6) Definir la función **fact :: Int** → **Prog** tal que dado un entero **n** mayor o igual a **0,** devuelve un programa que calcula el factorial de éste y lo guarda en la variable **"fact".** Si **n** es negativo, el programa puede comportarse como desee.

7) Definir la función **par :: Int** → **Prog** tal que dado un entero **n** mayor o igual a **0,** devuelve un programa que guarda en la variable **"par"** un **1** o un **0,** dependiendo si el **n** es par o no. Si **n** es negativo, el programa puede comportarse como desee.

8) Definir, la función **mini :: Int -> Int -> Prog** que recibe un entero **n**, un entero **m**, ambos mayores o iguales a **0,** y devuelve un programa que guarda en la variable **"min"** el mínimo

de **n** y **m.** En el caso que alguno de los argumentos sea negativo, el programa puede comportarse como desee.

9) Definir la función **fib :: Int → Prog** tal que dado un entero **n,** devuelve un programa que calcula el enésimo número de la serie de fibonacci empezando desde (1, 1, 2, 3, 5, 8, 13…) y lo guarda en la variable **"fib"**. Si **n** es negativo, el programa puede comportarse como desee.

5. Entregables

- El trabajo podrá realizarse en grupos de hasta dos estudiantes.
- Se pueden utilizar las funciones del Preludio de Haskell. Cualquier otra función auxiliar que se necesite utilizar aparte de las de Preludio debe ser definida y se debe explicar qué́ hace (en forma de comentario en el código).
- La entrega deberá realizarse por **Gestión** antes del 10 de diciembre a las 21hs.
- Cada estudiante deberá subir un único archivo Haskell (.hs) con el código fuente de la solución. **En caso de haber hecho el trabajo en un grupo de dos estudiantes, el archivo debe incluir ambos nombres y números de estudiantes como comentarios al principio del mismo**.
- En Aulas se encuentra el archivo **Prog.hs** con las funciones que deben implementarse y funciones para probar los programas pedidos en los ejercicios 6 a 9. Para facilitar la corrección, solicitamos utilizarlo como template.
- **IMPORTANTE:** No se corregirán archivos que no compilen, por lo que recomendamos comentar el código que no compile y dejar como **undefined** las funciones no implementadas.
- Se correrá software para la detección de plagios, y en caso de verificarse porcentajes significativos de similitud entre programas de distintos grupos, se tomarán las acciones correspondientes.
- La defensa del obligatorio se hará conjuntamente con la del parcial.