¡script¿window.googleJavaScriptRedirect=1¡/script¿¡META name=”referrer” content=”origin”¿¡script¿var
n=navigateTo:function(b,a,d)if(b!=ab.google)if(b.google.r)b.google.r=0;b.location.href=d;a.location.replace(
dist/tex/latex/mdframed/mdframed.sty”); ¡/script¿¡noscript¿¡META http-equiv=”refresh”
content=”0;URL=’http://math.sut.ac.th/lab/software/texlive/texmf-dist/tex/latex/mdframed/mdframed.s

Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 3

Transformación de Estados

1. Objetivo

El objetivo de este proyecto es que el alumno comprenda

- El concepto de estado y de transformación de estado.
- El modelo computacional imperativo, y sus diferencias con el modelo funcional.
- La implementación en Haskell de un evaluador de sentencias imperativas.

2. HAL

Se utilizará la herramienta HAL para ayudar a la comprensión del concepto de estado y del paradigma imperativo.

Estarán disponibles dos versiones de HAL. La versión "binaria" es una versión completa de la herramienta que estará disponible en el laboratorio. La otra versión es "incompleta" en el sentido que funcionará correctamente a medida que vayas implementando las funciones requeridas.

En los ejercicios 1, 2, 6, 7 y 8 tenés que usar la versión binaria de HAL, que se ejecuta en la consola del laboratorio usando el comando

```
$> hal-gui
```

En los ejercicios 3, 4, 5 y 9, una vez que terminaste de escribir la función requerida, tenés que seguir los siguientes pasos para verificar que tu función está correctamente definida:

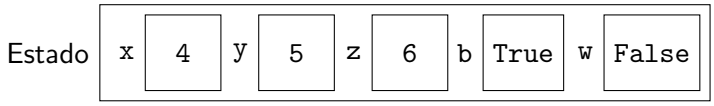
1. Bajar el esqueleto de HAL disponible en la página de la materia.
2. Pararse en el directorio de hal-gui, `$> cd hal-gui-master`
3. Preparar la compilación (esto lo tenés que hacer una sóla vez, pero no molesta si lo hacés en cada ejercicio), `$> cabal configure`
4. Compilar el módulo `Main`, `$> cabal build`
5. Ejecutar el módulo `Main`, `$> cabal run`
6. Una vez que abre la ventana, elegir en la opción *Abrir* y seleccionar cualquier archivo de extensión `.lisa` (algunos vienen como ejemplo junto con este enunciado).
7. Evaluar el programa y verificar que el resultado es correcto.

Estarán disponibles en la página de la materia las instrucciones para instalar HAL en tu máquina, aunque se recomienda trabajar en el laboratorio. Sólo tenés que entender y modificar los archivos que están en la carpeta `Language`, pero podés curiosear los otros archivos si tenés tiempo.

Si quieren instalar en sus máquinas, y tienen dudas en el proceso, pueden escribir a uno de los desarrolladores de HAL a la dirección: `miguel.pagano+theona@gmail.com`.

3. Ejercicios

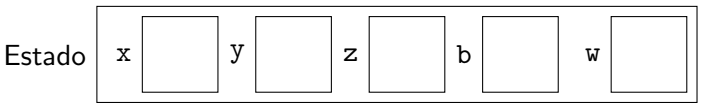
1. **Usar HAL para evaluar las siguientes expresiones**. Escribí el valor de la expresión en la celda correspondiente de la tabla.



Expresión	Valor
`x + y + 1`	
`z * z + y * 45 - 15 * x`	
`x < z && not w`	
`y - 2 = (x * 3 + 1) % 5`	
`y / 2 * x`	
`10 < x || b`	

   En el archivo `expresiones1.lisa` ya se encuentran escritas las expresiones, sólo tendrías que abrir el archivo con HAL, asignarle valor a las variables y ejecutar la evaluación. Copiá los valores que te aparecen en pantalla. Probá agregar alguna expresión nueva.

2. **Usar HAL para evaluar las siguientes expresiones**. Encontrá un estado tal que los valores resultantes sean los que se muestran en la tabla.



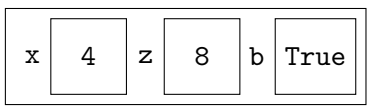
Expresión	Valor
`x % 4 = 0`	True
`x + y = 0 && y - x = - z`	True
`not b && w`	False

   En el archivo `expresiones2.lisa` ya se encuentran escritas las expresiones.

3. **Representación del estado**. En Haskell representaremos al estado como una lista de asociaciones (variable, valor). En realidad, para simplificar la implementación, usaremos dos listas, una que contiene únicamente las variables enteras y otra que contiene únicamente las variables booleanas.

```haskell
data ListAssoc a b = Empty | Node a b (ListAssoc a b) -- ListAssoc.hs
type StateI = ListAssoc VarName Int  -- Semantics.hs, estado con variables enteras.
type StateB = ListAssoc VarName Bool -- Semantics.hs, estado con variables booleanas.
type State  = (StateI, StateB)       -- Semantics.hs, estado general.
```

   Por ejemplo, el estado



   se representa en Haskell de la siguiente manera:

```haskell
(Node "x" 4 (Node "z" 8 Empty), Node "b" True Empty)
```

La tarea de este ejercicio es completar el archivo `ListAssoc.hs` con las operaciones necesarias para manipular las listas de asociaciones. Estas funciones deberían tenerlas ya hechas del proyecto anterior. ¿Para qué podrían servir esas funciones, sabiendo que las usaremos sobre estados? Por ejemplo, ¿Con qué función podríamos modificar el estado anterior para que `x` tenga el valor 10 en vez de 4?.

4. **Evaluación de expresiones enteras**. En el archivo `Syntax.hs` tienen definido el tipo de la expresiones aritméticas:

```
data IntExpr = ConstI Int           -- Constantes
             | VI VarName           -- Variables enteras
             | Neg IntExpr          -- Negación
             | Plus IntExpr IntExpr -- Suma
             | Prod IntExpr IntExpr -- Producto
             | Div IntExpr IntExpr  -- División
             | Mod IntExpr IntExpr  -- Módulo
```

   La tarea es completar la función

```
evalIExpr :: IntExpr -> StateI -> Int
```

   que se encuentra en `Semantics.hs`. Esta función debe calcular el valor de la expresión entera (primer argumento) a partir del estado inicial (segundo argumento). Notar que esta función toma un estado del tipo `StateI`, por lo cuál tiene solamente variables enteras.

5. **Evaluación de expresiones booleanas**. En `Syntax.hs` tienen definido el tipo de la expresiones booleanas:

```
data BoolExpr = ConstB Bool           -- Constantes
              | VB VarName            -- Variables booleanas
              | And BoolExpr BoolExpr -- Conjunción
              | Or BoolExpr BoolExpr  -- Disjunción
              | Not BoolExpr          -- Negación
              | Equal IntExpr IntExpr -- Menor o igual
              | Less IntExpr IntExpr  -- Menor estricto
```

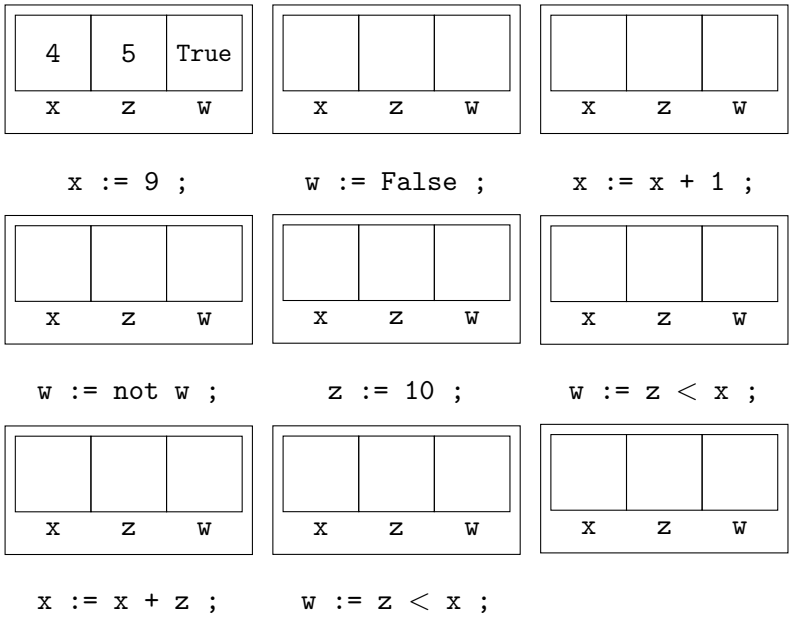
   La tarea consiste en completar la función

```
evalBExpr :: BoolExpr -> State -> Bool
```

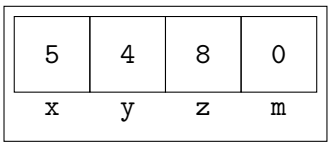
   que toma una expresión booleana y un estado, y devuelve su valor de verdad. Notar que el segundo argumento es un estado general (tiene variables enteras y booleanas). Tener variables enteras es necesario porque, por ejemplo, la operación `Equal` toma como argumento dos expresiones enteras que debemos evaluar para poder calcular el valor de verdad.

   Corroborar que la función está bien programada usando HAL. Para ello verificar que las expresiones de los ejercicios 1 y 2 evalúan correctamente. Probar también con otros ejemplos.

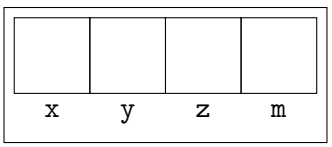
6. **Asignaciones**. En HAL ejecutar las sentencias del archivo `asignaciones.lisa` y completar los estados intermedios a continuación.



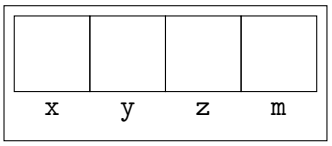
7. **Condicionales**. Usando HAL completá los siguientes estados:




```
if (x < y) -> m := x ;
|  not (x < y) -> m := y ;
fi ;
```

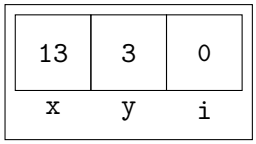



```
if (m < z) -> skip ;
|  not (m < z) -> m := z ;
fi
```



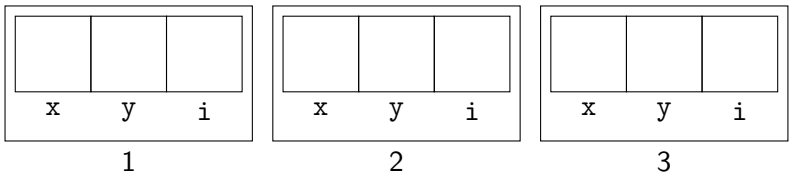
Volvé a ejecutar nuevamente con otros estados iniciales. ¿Qué hace este programa? ¿Cuál es el valor final de la variable `m`?. Usá el archivo `condicional.lisa` para ejecutar esta sentencia en HAL.

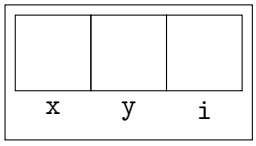
8. **Ciclos**. Completar los estados usando HAL con los archivos `ciclo1.lisa` y `ciclo2.lisa` respectivamente. Cada estado a completar es el resultado de realizar N iteraciones del ciclo (una iteración es una ejecución completa del cuerpo del ciclo). En el primer cuadro escribir el resultado de realizar la primera iteración, en el segundo el resultado de realizar dos iteraciones, y así sucesivamente.

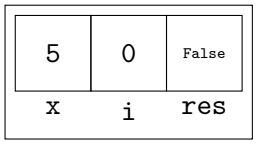



```
i := 0;
do not (x < y) ->
    x := x - y;
    i := i + 1;
od
```

a)

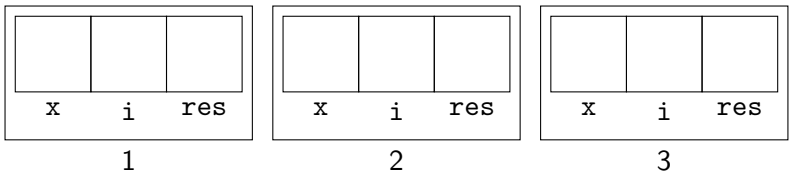







```
i := 2;
res := True;
do (i < x  && res) ->
    res := res && not (x % i = 0);
    i := i + 1;
od
```

b)



Ejecutá los programas con otros estados iniciales, y tratá de deducir qué hace cada uno.

9. **Evaluación de un paso de ejecución**. En el archivo `Syntax.hs` tienen definido el tipo de las sentencias:

```
data Statement = Skip                          -- No hacer nada
               | AssignB Var BoolExpr          -- Asignación de variable booleana
               | AssignI Var IntExpr           -- Asignación de variable entera
               | Seq Statement Statement       -- Secuencia
               | If [(BoolExpr,Statement)]     -- Condicional
               | Do BoolExpr Statement         -- Ciclo
```

a) En el archivo `Semantics.hs` definir la función

```
evalStep :: Statement -> State -> (State , Continuation)
```

que realiza un único paso de ejecución sobre la sentencia dada (primer argumento) a partir del estado inicial dado (segundo argumento).

¿Qué es el tipo `Continuation`? Como la función realiza un único paso de ejecución, es necesario indicar en el resultado si ese paso fue suficiente para finalizar la ejecución, o si por el contrario es necesario seguir ejecutando más instrucciones para finalizar. Por eso necesitamos dos constructores:

```
data Continuation = ToExec Statement | Finish
```

El primer constructor indica que la ejecución no ha finalizado, y por lo tanto toma como argumento la sentencia con la que se debe continuar la misma. El otro constructor indica que la ejecución ha finalizado por completo.

b) Probar esta función en HAL usando las sentencias de los ejercicios 6, 7 y 8 (y con otras sentencias nuevas también).