Algoritmos y Estructuras de Datos I - Laboratorio
Proyecto 3

Programas como transformadores de estados

1. Objetivo

El objetivo de este proyecto es introducir

- el concepto de estado y de programas como transformadores de estado;
- el modelo computacional imperativo, y sus diferencias con el modelo funcional;
- la implementación en Haskell de un evaluador de programas imperativos.

2. HAL

A lo largo de todo el proyecto, se utilizará la herramienta HAL para ayudar a la comprensión del concepto de estado y del paradigma imperativo.

Trabajaremos con dos versiones de HAL. La versión "binaria" es una versión completa de la herramienta que está disponible en el laboratorio. Tenés que usar esta versión en los ejercicios 1, 2, 5, 6 y 7, ejecutando en la consola del laboratorio el comando

```
$> hal-gui
```

La otra versión de HAL está "incompleta". Es necesario implementar las funciones de los ejercicios 3, 4, y 8, que forman parte de la herramienta, para que funcione correctamente. Para poder hacer estos ejercicios, debes instalar la versión "incompleta", implementar las funciones, y luego compilar y ejecutar. Las instrucciones para hacer esto, tanto en una máquina del laboratorio como en tu computadora personal están en la página de la materia.

3. Ejercicios

1. **Usá HAL para evaluar las siguientes expresiones:**

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`	

a partir del siguiente estado:

x	4	y	5	z	6	b	True	w	False

Las expresiones ya están escritas en el archivo `expresiones1.lisa`. Sólo debés abrir el archivo con HAL, asignarle valor a las variables y ejecutar la evaluación. Copiá los valores que te aparecen en pantalla. Probá además evaluar alguna expresión nueva.

2. **Usando HAL como ayuda**, encontrá valores para las variables que forman el estado:

x		y		z		b		w	

de manera que las siguientes expresiones tengan el valor indicado:

Expresión	Valor
`x % 4 = 0`	True
`x + y = 0 && y - x = (-1) * z`	True
`not b && w`	False

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

3. **Representación del estado**. En HAL representamos el estado de un programa imperativo como una tupla de listas de asociación (variable, valor). La primer lista contiene los valores de las variables enteras, mientras que la segunda, los valores de las variables *booleanas*.

```
-- ListaAsoc.hs
data ListaAsoc a b = Vacia | Nodo a b (ListaAsoc a b)

-- Semantics.hs
type StateI = ListaAsoc VarName Int    -- estado con variables enteras.
type StateB = ListaAsoc VarName Bool   -- estado con variables booleanas.
type State  = (StateI, StateB)         -- estado general.
```

Por ejemplo, el estado

x	4	z	8	b	True

se representa de la siguiente manera:

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

La tarea de este ejercicio es completar el archivo `ListAsoc.hs` con las operaciones necesarias para manipular las listas de asociaciones. Cada una de ellas está descripta en el comentario que las acompaña. Algunas de estas funciones ya fueron implementadas en el proyecto 2.

A partir de tu conocimiento sobre cómo funciona el paradigma imperativo, ¿en qué momento de la ejecución de un programa imperativo en HAL se utilizan las distintas funciones sobre listas de asociación? Dicho de otra manera, ¿en qué momentos de la ejecución de un programa se manipula el estado?

4. **Evaluación de expresiones enteras**. En el archivo `Syntax.hs` se define 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 de un estado (segundo argumento). Notar que esta función toma un estado del tipo `StateI`, por lo cuál tiene solamente variables enteras.

Corroborá que la función está bien programada usando HAL. Para ello verificá que las expresiones de los ejercicios 1 y 2 que contienen subexpresiones enteneras evalúen correctamente. Probá también con otros ejemplos.

5. **Asignaciones**. Ejecutá en HAL la siguiente secuencia de asignaciones (archivo `asignaciones.lisa`) y completá los estados intermedios y final:

4	5	True						
x	z	w	x	z	w	x	z	w

```
x := 9 ;          w := False ;          x := x + 1 ;
```

x	z	w	x	z	w	x	z	w

```
w := not w ;          z := 10 ;          w := z < x ;
```

x	z	w	x	z	w	x	z	w

```
x := x + z ;          w := z < x ;
```

6. **Condicionales**. Usando HAL, ejecutá el siguiente programa (archivo `condicional.lisa`) y completá los estados:

5	4	8	0
x	y	z	m

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

x	y	z	m

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

x	y	z	m

Volvé a ejecutar nuevamente con otros estados iniciales. ¿Qué hace este programa? ¿Cuál es el valor final de la variable `m`?.

7. **Ciclos**. Para cada item, completá los estados corriendo los programas en HAL (archivos `ciclo1.lisa` y `ciclo2.lisa` respectivamente). Cada estado a completar es el resultado de realizar 1, 2, 3 o 4 iteraciones del ciclo. Una iteración es la ejecución completa del cuerpo del ciclo.

a)

13	3	0
x	y	i

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

x	y	i

1

x	y	i

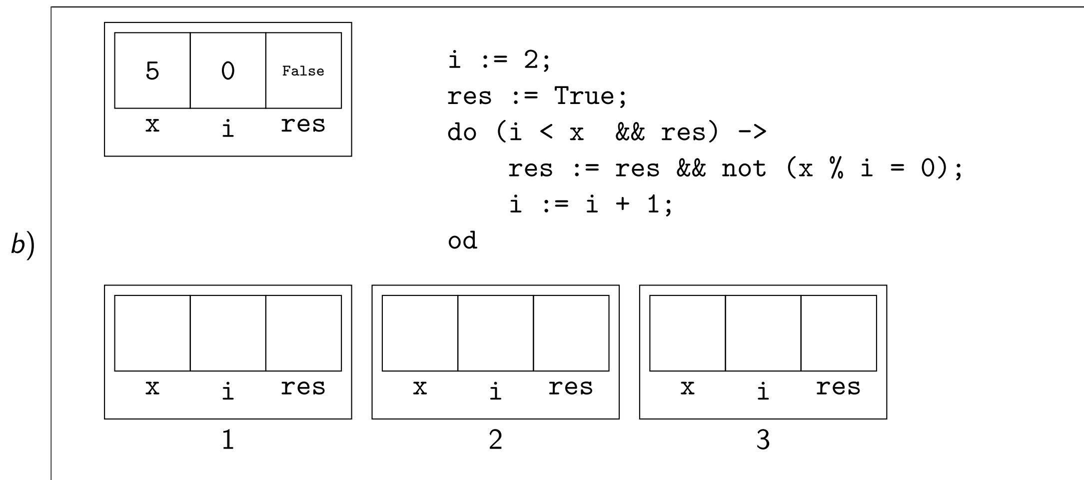
2

x	y	i

3

x	y	i

4



Ejecutá los programas con otros estados iniciales para deducir qué hace cada uno.

8. **Evaluación de programas**. En el archivo `Syntax.hs` se encuentra definido el tipo de las sentencias:

```haskell
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
```

La tarea es implementar la función

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

que se encuentra en el archivo `Semantics.hs`. Esta función realiza un único paso de ejecución sobre la sentencia dada (primer argumento) a partir del estado dado (segundo argumento).

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. Para ello utilizamos el tipo `Continuation`, y necesitamos dos constructores:

```haskell
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.

Probá esta función en HAL usando los programas de los ejercicios 5, 6 y 7 (y con otras sentencias nuevas también).