EJERCICIOS TADS PED

1. Sea un vector de números naturales. Utilizando exclusivamente las operaciones *asignar* y *crear*, define la sintaxis y la semántica de la operación *eliminar* que borra las posiciones pares del vector marcándolas con "-1". Para calcular el resto de una división, se puede utilizar la operación MOD.

1)

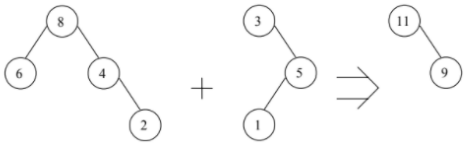
eliminar: vector → vector

Var v:vector; i,x:natural;

eliminar(crear()) = crear()
si (i MOD 2) == 0
  entonces eliminar(asignar(v,i,x)) = asignar(eliminar(v),i,-1)
  sino eliminar(asignar(v,i,x)) = asignar(eliminar(v),i,x)

---

1. Utilizando exclusivamente las operaciones constructoras generadoras de los árboles binarios, definir la sintaxis y la semántica de la operación *sumar* que crea un árbol binario resultado de sumar las etiquetas de los nodos simétricos de 2 árboles binarios dados cuyas etiquetas son números naturales.
Nota: Si un nodo de un árbol no tiene nodo simétrico en el otro, en el árbol binario resultado no aparecerá dicho nodo.



1)

sumar: arbin, arbin → arbin

var i,l,d,r:arbin; x,y:natural;

sumar(crear_arbin(),crear_arbin()) = crear_arbin()
sumar(crear_arbin(),d) = crear_arbin()
sumar(i,crear_arbin()) = crear_arbin()
sumar(enraizar(i,x,d),enraizar(l,y,r)) = enraizar(sumar(i,l),x+y ,sumar(d,r))

1. Utilizando las operaciones definidas en clase para la definición del tipo vector definir la sintaxis y la semántica de la operación *inversos* que indica si dos vectores de naturales dados son inversos entre sí, es decir, si un vector es el inverso del otro. Nota: se asume que el vector está creado con 100 componentes y el rango de las mismas es de 1…100.

1.

Sintaxis:

inversos: vector, vector --> bool
invAux: vector, vector --> bool

Semántica:

Var v,w: vector; i,j,x,y: natural;

inversos(v,w) = invAux(v,w) && invAux(w,v)
invAux(crear_vector(),w) = VERDADERO
si x == recu(w,100-i+1) entonces invAux(asig(v,i,x),w) = invAux(v,w)
sino invAux(asig(v,i,x),w) = FALSO

1. Define la sintaxis y semántica de la operación **posición** que actúa sobre un vector y devuelve la posición menor sobre la que se ha asignado el valor que recibe como parámetro. Si no se ha asignado el valor en el vector se debe devolver 0. Ejemplo:

posición(asig(crear(), 3, b), a) = 0
posición(asig(asig(crear(), 3, b), 1, b), b) = 1
posición(asig(asig(asig(crear(), 3, b), c, 2), 1, b), b) = 1
posición(asig(asig(asig(crear(), 1, b), c, 2), 3, b), b) = 1

1.

posicion(vector, item) → entero
posicionAux(vector, item, entero) → entero
Var v:vector; i: entero; x: item;
posicion(crear(), x) = 0
si (x<>y) entonces
posicion(asig(v, i, x), y) = posicion(v, y)
si no posicion(asig(v, i, x), y) = posicionAux(v, y, i)
posicionAux(crear(), x, i) = i
si (x<>y) entonces
posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, j)
si no si (i < j) entonces
posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, i)
si no posicionAux(asig(v, i, x), y, j) = posicionAux(v, y, j)

1. Realizar la especificación algebraica de una función que determine si un árbol binario cumple las condiciones de balanceo de un AVL. Para ello, se supone que están definidas las siguientes operaciones:

- Resta de naturales: *resta(a,b)*, *a* debe ser mayor o igual que *b*.
- Comparación de naturales: <=, >=, ==, >, <
- Operaciones booleanas: AND, OR.
- Y todas las operaciones del tipo de datos árbol binario.

1.

VAR i,d: arbin; x: item

ES_AVL(arbin) → bool

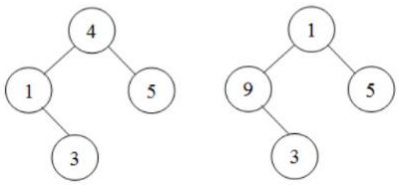
ES_AVL(crear()) = V
ES_AVL(enraizar(crear_arbin(), x, crear_arbin())=V

ES_AVL(enraizar(i, x, d)) =

si altura(i) >= altura(d)
entones
    si resta(altura(i), altura(d)) >= suc(suc(cero))
    entones
        F
    sino
        ES_AVL(i) y ES_AVL(d)
    fsi
sino
    si resta(altura(d), altura(i))>=suc(suc(cero))
    entones
        F
    sino
        ES_AVL(i) y ES_AVL(d)

    fsi
fsi

1. Utilizando exclusivamente operaciones constructoras generadoras definir la sintaxis y la semántica de la operación *mismas_hojas* que actúa sobre dos árboles binarios de números naturales e indica si los dos árboles tienen la misma estructura y hojas iguales. (Por ejemplo, los árboles de la figura tienen la misma estructura y hojas iguales).



1.

Sintaxis:

mismas_hojas(arbin, arbin) → bool

Semántica:

Var i, d, iz, der: arbin; x, y: item;

[enraizar(i,x,d) → E(i,x,d)]

mismas_hojas(crear_arbin(), crear_arbin()) = TRUE
mismas_hojas(E(crear_arbin(), x, crear_arbin()), E(crear_arbin(), y, crear_arbin())) =
si (x==y) entonces TRUE

mismas_hojas(E(i, x, d), crear_arbin()) = FALSE
mismas_hojas(crear_arbin(), E(i, x, d)) = FALSE

mismas_hojas(E(i, x, d), E(iz, y, der)) = mismas_hojas(i, iz) & mismas_hojas(d, der)

1. a) Sean las siguientes variables: G: Grafo; x, y, a, b:Vértice; p, q: Ítem; Completa en esta misma hoja la semántica de la operación BorrarOcurrencias (BO) que recibe un multigrafo y devuelve un multigrafo en el que se han eliminado todas las ocurrencias del arco <a, b> pasado como parámetro.

BO(CrearGrafo() , a, b) ↔ CrearGrafo()
BO(. . . . . , a, b) ↔ . . . . . .
. . . . . .
. . . . . .

1.

a) BO(C r e a r G r a f o () , a, b) ↔ CrearGrafo()
BO(InsertarArista(G, x, y, p), a, b) ↔ si (x == a) y (y == b)
entonces BO(G, a, b)
sino InsertarArista(BO(G, a, b), x, y, p)

a) Utilizando exclusivamente operaciones constructoras generadoras, completa la semántica de la operación *borrar* de listas estudiada en clase.

Sintaxis: borrar(lista, posicion) → lista
VAR L_1, L_2: lista; x: item; p: posicion;
borrar(crear( ), p) = crear( )
si p == primera(inscabeza(L_1, x)) **entonces**
borrar( ) =
si no borrar( ) =

b) Utilizando exclusivamente operaciones constructoras generadoras, completa la semántica de la operación *multiplicación* de números naturales estudiada en clase.

Sintaxis: mult(natural, natural) → natural
VAR x, y: natural;
mult(cero, x) =
mult( ) =
mult( ) =

1.

a)

borrar(crear( ), p) = crear( )
si p == primera(inscabeza(L_1, x)) **entonces**
borrar(inscabeza(L_1, x), p) = L_1
si no borrar(inscabeza(L_1, x), p) = inscabeza(borrar(L_1, p), x)

b)

mult(cero, x) = cero
mult(x, cero) = cero
mult(suc(y), x) = suma(mult(y, x), x)

1. Dada la sintaxis y la semántica de la operación Examen:
Examen: lista → lista

Var L: lista; X,Y,a,b,c: Item;
Examen(crear()) = crear()
Examen(IC(crear(), X)) = IC(crear(), X)
Examen(IC(IC(L, Y), X)) = IC(IC(Examen(L), X), Y)

a) Explicar el funcionamiento de la operación Examen.
b) Aplicar la operación Examen a la lista: IC(IC(IC(crear(), c), b), a). Mostrar todos los pasos.
c) Sea una entidad bancaria que dispone de dos tipos de datos: (i) una lista (lista_de_clientes_morosos) que contiene los identificadores (números enteros) de los clientes morosos; (ii) un vector (vector_de_saldos) que almacena en cada posición (identificador del cliente) el saldo de todas las cuentas bancarias del cliente en la mencionada entidad bancaria.
Utilizando exclusivamente las operaciones generadoras constructoras (vistas en clase) del tipo lista y vector, definir la sintaxis y semántica de la operación "moroso" que actualiza el vector_de_saldos, asignando a cada cliente de la entidad bancaria que es moroso un saldo de "0".

1. a) La operación Examen actúa sobre una lista y devuelve una lista. Los elementos que estén en posiciones pares (x) de la lista pasan a ocupar una posición impar anterior (x-1). Los que estén en posiciones impares (y) pasan a ocupar la posición par siguiente (y+1).
Si la lista contiene un número impar de elementos, el último no cambia su posición.

b) IC(IC(IC(crear(), c), a), b)

c) Según la definición de la operación asignar del tipo vector vista en clase:
si (i < > j) **entonces**
asig(asig(v, i, x), j, y) = asig(asig(v, j, y), i, x)
si no asig(asig(v, i, x), j, y) = asig(v, i, y) **fsi**

Con lo que:
moroso: lista, vector → vector Var L: lista; V: vector; id: entero;

moroso (crearLista(), V) = V
moroso(L, crearVector()) = crearVector()
moroso(IC(L, id), V) = moroso (L, asig(V, id, 0)

1. **a)** Utilizando exclusivamente las operaciones constructoras generadoras del **tipo pila**, definir la sintaxis y la semántica de la operación QuitaPares que actúa sobre una pila y devuelve una pila en la que se han eliminado las posiciones pares de la misma.

Nota: se asume que la posición 1 (impar) de la pila está en la cima de la misma.

b) Explicar las dos representaciones enlazadas del **tipo cola** vistas en clase definiendo los elementos que aparecen en la misma. Para cada una de ellas, explicar razonadamente (justificando la respuesta) la complejidad temporal (en su mejor y peor caso) de las operaciones encolar y desencolar.

1. a) QuitaPares: pila → pila

Var p: pila; x,y: item;
QuitaPares (crear_pila())= crear_pila()
QuitaPares (apilar(crear_pila(), x))= apilar(crear_pila(), x)
QuitaPares (apilar(apilar(p, x), y))= apilar(QuitaPares(p), y)

b) Para la representación enlazada de las colas se utilizan punteros a **nodo**. El nodo contiene el **dato** a almacenar y un **puntero al siguiente nodo**. Se definen dos punteros adicionales: **tope** y **fondo**. **tope** apunta al primer elemento que hay que desencolar y **fondo** apunta al último elemento de la cola.

Utilizando esta estructura la complejidad temporal de las operaciones **encolar y desencolar** es la misma (ya que tenemos punteros a la primera y a la última posición de la cola): $\Omega(1) = O(1) = \theta(1)$

Esta representación se optimiza con las **colas circulares enlazadas**, en las que sólo se necesita un puntero (**fondo**) al último elemento de la cola; el siguiente elemento de fondo apunta al primer elemento de la cola.

La complejidad temporal de las operaciones **encolar y desencolar** es la misma (ya que tenemos un puntero que apunta a la última posición y al primero –siguiente de fondo–): $\Omega(1) = O(1) = \theta(1)$

1.

a) Define la sintaxis y la semántica de la operación **inverso** que actúa sobre un vector de números naturales y devuelve el vector inverso del vector de entrada.

Nota: utilizar exclusivamente las operaciones constructoras generadoras del vector. Se asume que el tamaño del vector es una constante determinada, tam, y que están definidas y se pueden usar todas las operaciones de números naturales (suc, suma, resta, multiplicación y división).

b) Define la sintaxis y la semántica de la operación **insertar** vista en clase que inserta un elemento en una lista con acceso por posición.

1.

a)

inverso(vector) → vector
VAR v: vector; i, x: natural;
inverso(crear_vector()) = crear_vector()
inverso(asig (v,i,x)) = asig(inverso(v), tam+suc(cero)-i, x)

b)

insertar(lista, posicion, item) → lista
VAR L_1: lista; x, y: item; p: posicion;
insertar(crear_lista( ), p, x) = crear_lista( )
si p == primera(inscabeza(L_1, x)) **entonces**
    insertar(inscabeza(L_1, x), p, y) = inscabeza(inscabeza(L_1, y), x)
si no insertar(inscabeza(L_1, x), p, y) = inscabeza(insertar(L_1, p, y), x)

14 → 11 → 12 → 13

c) Utilizando exclusivamente las operaciones constructoras generadoras de grafo, definir la sintaxis y la semántica de la operación *CalculaPesos* que actúa sobre un grafo dirigido ponderado donde: los vértices son números Naturales y los Item son números Naturales que representan distancias kilométricas para cada par de vértices. *CalculaPesos* devuelve la suma de las distancias kilométricas de todos los arcos del grafo entre vértices pares.

Nota: se pueden utilizar todas las operaciones definidas para números naturales.

c)

CalculaPesos: grafo → natural
Var G: grafo; x,y: vértice; p: natural;

CalculaPesos (crear_grafo())=0
CalculaPesos (InsertarArista(G,x,y,p))=
si ((x MOD 2 ==0) && (y MOD 2 ==0))
entonces p + CalculaPesos (G)
sino CalculaPesos (G)

c) Utilizando exclusivamente las operaciones constructoras generadoras del tipo grafo, definid la semántica de la operación examen que se aplica sobre un grafo dirigido ponderado cuyos arcos están etiquetados con números naturales (es decir, el peso de los arcos son números naturales) y devuelve el número de arcos cuyo peso es igual a uno especificado. La sintaxis de la operación 'examen' es la siguiente:

examen: grafo, natural_peso → natural

c) Var G: grafo; x,y: vértice; p,q: natural;
examen(crear_grafo(),q)=0
examen(InsertarArista(G,x,y,p),q)=
si (q ==p) entonces 1 + examen(G,q)
si no examen(G,q)

3. a) Utilizando exclusivamente las operaciones constructoras generadoras del tipo árbol, define la sintaxis y la semántica de la operación *EsHoja*, que actúa sobre un árbol y nos indica si éste es una hoja o no. **(0,5 puntos)**

sintaxis:
EsHoja: arbin -> bool

semántica:

var x:item; i,d: arbin

EsHoja(crea_arbin()) = FALSO
EsHoja(enraizar(crea_arbin(), x, crea_arbin())) = CIERTO

EsHoja(enraizar(i, x , d)) = FALSO

1. Utilizando exclusivamente las operaciones constructoras generadoras del tipo lista, definir la sintaxis y la semántica de la operación *examen* que actúa sobre una lista ordenada ascendente de números naturales que permite elementos repetidos y devuelve una lista en la que se han borrado todas las ocurrencias de un ítem especificado. **La operación debe ser lo más eficiente posible, reduciendo al máximo el número de llamadas recursivas**.
Nota: se asume que está definido el TAD de los números naturales con todas las operaciones aritméticas.

Sintaxis:
examen(lista, natural) → lista **(0,1 puntos)**

Semántica:
Var L: lista; ítem, x: natural;

[InsertarCabeza(L,x) → IC(L,x)]
examen(crearLista(), item)=crearLista() **(0,1 puntos)**
examen(IC(L,x), item) =
si (x>ítem) **entonces** IC(L,x) **(0,7 puntos)**
sino si (x<ítem) **entonces** IC(examen(L,ítem), x) **(0,3 puntos)**
sino examen(L,item) **(0,3 puntos)**

1. a) Utilizando exclusivamente las operaciones constructoras generadoras de vector, definir la sintaxis y la semántica de la operación *subvector* que actúa sobre un vector de números naturales y recibe dos números naturales (que representan posiciones del vector). La operación devuelve el vector original conteniendo sólo los elementos entre las posiciones dadas (ambas posiciones incluidas).
Nota: Se asume que las posiciones son correctas y están en el rango del vector: desde 1 hasta la longitud del vector. Se pueden utilizar todas las operaciones definidas para números naturales y booleanos.
Ej: *subvector* ([1,5,2,20,3], 2, 4) → [5,2,20]

a)

```
subvector: vector, natural, natural → vector
Var v: vector; i,x,p1,p2: natural
subvector(crear(), p1, p2) = crear()
subvector(asig(v,i,x), p1, p2) =
        si [(i>p1) y (i<p2)] o [i==p1] o [i==p2]
 entonces asig (subvector(v, p1, p2), i, x)
                sino subvector(v, p1, p2)
```

a) Utilizando exclusivamente las operaciones constructoras generadoras de lista, definir la sintaxis y la semántica de la operación *sublista* que actúa sobre una lista de números naturales y devuelve la lista original en la que se han eliminado los elementos que ocupan las posiciones pares y también los elementos iguales a un número natural especificado que ocupan las posiciones impares.

Nota: se pueden utilizar todas las operaciones definidas para números naturales.

a)

```
sublista: lista , ítem → lista
Var l1:lista; x,y, Z:item;
sublista(crear_lista(),Z) = crear_lista()
sublista(IC(crear_lista(),x),Z) =
        si x==Z entonces crear()
          sino IC(crear_lista(),x)
sublista(IC(IC(l1,x),y),Z) =
        si y==Z entonces sublista(l1,Z)
          sino IC(sublista(l1,Z),y)
```

1. Utilizando exclusivamente las operaciones constructoras generadoras de los números Naturales define la sintaxis y la semántica de las operaciones resta y división (calcula el cociente de dos números Naturales). NOTA: Se asume que para las dos operaciones el primer argumento es mayor que el segundo y que el cociente de la división es entero y exacto. Para la operación división se puede utilizar la operación resta.

resta, division: natural, natural --> natrual

Var x, y: natural

resta(x,cero)= x
resta(suc(x), suc(y))= resta(x,y)

division(x,cero)= error_natural
divison(x,x)= suc(cero)
division(x,y)= suc(division(resta(x,y),y))