Nombres: VERA

Aclaraciones: El parcial NO es a libro abierto. Cualquier decisión de interpretación que se tome debe ser aclarada y justificada. Para aprobar se requieren al menos 60 puntos. **Entregar cada ejercicio en hoja separada.**

Importante: Para la resolución del parcial **NO** es necesario ni está permitido el uso de **acum**. En caso de ser necesario, pueden asumir como definida la función *sum* ($\sum$) sobre listas. Las funciones *min* o *max* sobre listas,en caso de requerirlas, deben ser definidas.

```
tipo Ciudad=String;
tipo Precio=ℤ;
tipo Cliente=String;
tipo Vuelo=(Ciudad, Ciudad, Precio);
tipo TipoServicio = Hotel, Excursion;


tipo Servicio {
        observador tipo (s :Servicio) : TipoServicio;
        observador ciudad (s :Servicio) : Ciudad;
        observador precio (s :Servicio) : Precio;
        observador fecha (s :Servicio) : ℤ;
        invariante noEsGratis : precio(s) > 0;
}


tipo Paquete {
        observador ida (p: Paquete) : Vuelo;
        observador vuelta (p: Paquete) : Vuelo;
        observador servicios (p: Paquete) : [Servicio];
        observador precio (p: Paquete) : Precio;
        invariante voyYVengoBien : sgd(ida(p)) == prm(vuelta(p)) ∧
          prm(ida(p)) == sgd(vuelta(p)) ∧
          (∀s ← servicios(p))ciudad(s) == sgd(ida(p));
        invariante alMenosUnaNoche : |fechasHoteles(p)| > 0;
        invariante duermoBien : mismos(fechasHoteles(p),
          [i | i ← [primerNoche(p)..ultimaNoche(p)]]);
        invariante noEsUnaEstafa : precio(p) < trd(ida(p)) +
          trd(vuelta(p)) + ∑[precio(s) | s ← servicios(p)];
        invariante elPaqueteSePaga : precio(p) > 0;
}

tipo Agencia {
        observador vuelos (a: Agencia) : [Vuelo];
        observador servicios (a: Agencia) : [Servicio];
        observador paquetes (a: Agencia) : [Paquete];
        observador clientes (a: Agencia) : [Cliente];
        observador viajes (a: Agencia, c: Cliente) : [Paquete];
            requiere c ∈ clientes(a);
        invariante unoDeCada : sinRepetidos(vuelos(a)) ∧
          sinRepetidos(servicios(a));
        invariante paquetesValidos : ...;
        invariante viajesValidos : ...;
}

aux sinRepetidos (xs: T) : Bool = (∀i, j ← [0..|xs|))xs_i ≠ x_sj;
aux fechasHoteles (p: Paquete) : [ℤ] = [fecha(s) | s ← servicios(p), tipo(s) == Hotel];
aux primerNoche (p: Paquete) : ℤ = min(fechasHoteles(p));
aux ultimaNoche (p: Paquete) : ℤ = max(fechasHoteles(p));
```

Ejercicio 1. [20 puntos]

a) Completar el `invariante viajesValidos` , que indica que los viajes comprados por los clientes corresponden a paquetes ofrecidos por la agencia.

b) Completar el `invariante paquetesValidos` , que indica que los paquetes tienen vuelos y servicios que pertenecen a la agencia.

Ejercicio 2. [15 puntos] Especificar el aux $soloViajanPorTrabajo(a : Agencia) = [Cliente]$. Devuelve todos los clientes que sólo compraron paquetes que no contienen excursiones y en los cuales la duración del viaje (medida en la cantidad de noches de hotel) es estrictamente menor a 5 días.

Ejercicio 3. [35 puntos] Especificar el `problema cancelarExcursion (a: Agencia, s: Servicio)` . Modifica la agencia eliminando la excursión pasada como parámetro de todos los lugares en los que se estaba utilizando.

Ejercicio 4. [30 puntos]

a) Realizar la transformación de estados para el siguiente código:

```
bool arreglar(int[] a, int n){
  bool f = false;
  if(a[0]>a[1]){
    f = true;
    swap(a[0],a[1]);
  }
  return f;
}
```

Nota: Usar la siguiente especificación de swap

```
problema swap (a,b:ℤ) {
        modifica a, b;
        asegura a == pre(b) ∧ b == pre(a);
}
```

b) Para cada una de las siguientes especificaciones decidir si el programa es correcto. En caso de que no lo sea, agregar requieres o aseguras necesarios. Justificar en ambos casos.

```
problema arreglar (a:[ℤ], n : ℤ) = res : Bool {
        modifica a ;
        asegura mismos(a, pre(a)) ;
        asegura (∀i ← [0..|a| − 1)a_i < a_{i+1} ;
        asegura res == (∀i ← [1..|a|)pre(a)_0 > pre(a)_i ;
}

problema arreglar (a:[ℤ], n : ℤ) = res : Bool {
        requiere |a| == n ;
        requiere |a| == 10 ;
        modifica a ;
        asegura a == [min(pre(a)_0, pre(a)_1)] + +[max(pre(a)_0, pre(a)_1)] + +[ pre(a)_i | i ← [2..9] ] ;
        asegura res == pre(a)_0 > pre(a)_1 ;
}

problema arreglar (a:[ℤ], n : ℤ) = res : Bool {
        asegura True ;
}
```

Ejercicio 1:

a) invariante viajesValidos: (∀ c ← clientes(a), p ← viajes(a,c), [redacted]) perteneceP(p, paquetes(a))

aux perteneceP(p: Paquete, pa: [Paquete]): Bool = (∃ j ← pa) igualP(p, j)

aux igualP(p, j: Paquete): Bool = rdo(p) == rdo(j) ∧ dueño(p) == dueño(j) ∧ mismos(servicios(p), servicios(j)) ∧ precio(p) == precio(j)

✓

b) invariante paquetesValidos: (∀ p ← paquetes(a)) rdo(p) ∈ orden(a) ∧ dueño(p) ∈ clientes(a) ∧ perteneceS(servicios(p), servicios(a))

aux perteneceS(s: [Servicio], l: [Servicio]): Bool = (∀ j ← s) j ∈ l

✓

✓

NÚMERO DE ORDEN : 45

HOJA Nº

FECHA

Ejercicio 2:

aux todosVuelosPorTrabajo (a: Agencia) : [Cliente] =
[c | c ← clientes (a), noCompreExcursiones (c, a) ∧
duracionVuelosMS (c, a)]

aux noCompreExcursiones (c : Cliente, a : Agencia) : Bool =
(∀ p ← viajes (a, c), s ← servicios (p)) tipo (s)
≠ Excursion

aux duracionVuelosMS (c : Cliente, a : Agencia) : Bool =
(∀ p ← viajes (a, c)) | [1 | s ← servicios (p),
tipo (s) == Hotel] | < 5

Exercício 3:

problema cancelar Excursion (a: Agencia, s: Servicio){

(atrás) modifica a;

asegura: s ∉ servicios(a);

asegura: (∀ j ← servicios(pre(a)), j ≠ s) j ∈ servicios(a);

asegura: (∀ j ← servicios(a)) j ∈ servicios(pre(a));

asegura: mismos(viajes(a), viajes(pre(a)));

asegura: mismos(clientes(a), clientes(pre(a)));

asegura: (∀ p ← paquetes(a)), s ∉ servicios(p);

asegura: (∀ p ← paquetes(pre(a)), s ∉ servicios(p)

3 perteneceP(p, paquetes(a));

Em ej. 1a

asegura: (∀ p ← paquetes(a)) perteneceP(p, paquetes(pre(a)));

asegura (∀ c ← clientes(a)) mismosP(viajes(a,c), viajes sin s (pre(a) a, c, s));

aux viajes sin s (a: Agencia, c: Cliente, s: Servicio): [Paquete] = [p | p ← viajes(a,c), s ∉ servicios(p)];

aux mismosP(x: [Paquete], y: [Paquete]): Bool = |x| == |y| ∧ (∀ j ← x) cuentoP(j,x) == cuentoP(j,y);

aux cuentoP(p: Paquete, l: [Paquete]): ℤ =

|[1 | j ← l, igual(p,j)]|

↳ Em ej. 1.a

⊛ refine: ¬(∃p ← paquetes(a)) (|fechasHoteles(p)| == 1
∧ 5 ∈ servicios(p) ∧ tipo(s) == Hotel) ∨ (s ∈ servicios(p)
∧ tipo(s) == Hotel ∧ primeraNoche(p) < fecha(s) <
ultimaNoche(p)); NO! ⊛₁

refine: (∀ p ← paquetes(a)) precio(p) < tid(vdo(p))
+ tid(vuelto(p)) + Σ[precio(j) | j ← servicios(p),) ≠ s];

Ejercicio 4:

a) bool arreglar (int [] a, int n){
    bool f = false;
    // Estado E1;
    // Vale P_IF: f == False ; a == pre(a);
    // Modifico a
    if (a[0] > a[1]){
        f = true;
        swap(a[0], a[1]);
    }
    // Estado E2;
    // Vale Φ_IF: (pre(a)[0] > pre(a)[1] ∧ f == True ∧ a[0] == pre(a)[1] ∧ a[1] == pre(a)[0]) ∨ (pre(a)[0] ≤ pre(a)[1] ∧ f == False);
    return f;
}

No se usan comas en el lenguaje lógico.

Asumo las apertas

falta considerar el resto de los elementos de a

falta un estado acá.

• Demuestro el IF:

*RAMA TRUE:

if (a[0] > a[1]){
// Estado IFT1; ∧@E1
// Vale: f == False ∧ pre(a)[0] > pre(a)[1];
f = True;
// Estado IFT2;
// Vale f == True ∧ a@IFT1[0] > a@IFT1[1];
swap(a[0], a[1]);

B

B, la guarda no cumple.

NOTA

// Estado IFT3;
// Vale f==True ∧ a@IFT2[0] > a@IFT2[1] ∧ a[0] == a@IFT2[1] ∧ a[1] == a@IFT2[0];
// Implico: f==True ∧ me(a)[0] > me(a)[1] ∧ a[0] == me(a)[1] ∧ a[1] == me(a)[0];
// Implico: (f==True ∧ me(a)[0] > me(a)[1] ∧ a[0] == me(a)[1] ∧ a[1] == me(a)[0]) ∨ (me(a)[0] ≤ me(a)[1], f==False) : ϕ IF.;

* RAMA FALSE: {

}

// Estado IFF1;
// Vale f==False ∧ me(a)[0] ≤ me(a)[1];
// Implico (f==False ∧ me(a)[0] ≤ me(a)[1]) ∨ (f==True ∧ me(a)[0] > me(a)[1] ∧ a[0] == me(a)[1] ∧ a[1] == me(a)[0]) : ϕ IF;