UBA – Facultad de Ciencias Exactas y Naturales – Departamento de Computación– Algoritmos y Estructura de Datos I
Primer parcial – 24/04/2015

1		2	3	4		TOTAL
a	b			a	b	

Nro. de orden:
LU:
Apellidos:
Nombres:

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 ≠ xs_j;
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..|n| − 1)a_i < a_{i+1} ;
        asegura res == (∀i ← [1..|n|)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 ;
}
```