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 Nombre=String;
tipo Precio=ℤ;
tipo Mes=ℤ;
tipo Seccion = Bazar, Electro, Carniceria, Verduleria, Jugueteria, Limpieza, Lacteos, Vinos;
tipo Origen = Nacional, Importado;

tipo Producto {
        observador nombre (p: Producto) : Nombre;
        observador origen (p: Producto) : Origen;
}

tipo Chango {
        observador productos (c: Chango) : [Productos];
        observador cantidadXProducto (c: Chango, p: Producto) : ℤ;
            requiere p ∈ productos(c);
        invariante productosDistintos :
          sinRepetidos(nombres(productos(c)));
        invariante siEstaTieneCantidad :
          (∀p ← productos(c))cantidadXProducto(c, p) > 0;
}

tipo Gondola {
        observador seccion (g: Gondola) : Seccion;
        observador productos (g: Gondola) : [Productos];
        observador stock (g: Gondola, p: Producto) : ℤ;
            requiere p ∈ productos(g);
        invariante hayStock : (∀p ← productos(g))0 ≤ stock(g, p);
        invariante productosDistintos :
          sinRepetidos(nombres(productos(g)));
}

tipo Supermercado {
        observador gondolas (s: Supermercado) : [Gondola];
        observador changos (s: Supermercado) : [Chango];
        observador precios (s: Supermercado, p: Producto , m: Mes)
          : Precio;
            requiere 1 ≤ m ≤ 12;
            requiere (∃g ∈ gondolas(s))p ∈ productos(g);
        invariante gondolasDistintas :
          sinRepetidos(secciones(gondolas(s)));
        invariante noHaySeccionSinGondola :
          |sacarRepetidos(secciones(gondolas(s)))| = 8;
        invariante todosLosProductosTodosLosMeses : ...;
        invariante changosCopados : ...;
}

aux secciones (gs: [Gondola]) : [Seccion] = [seccion(g) | g ← gs];
aux sinRepetidos (xs: [T]) : Bool = (∀i, j ← [0..|xs|), i ≠ j)xs_i;
aux sacarRepetidos (xs: [T]) : [T] = [x_i | i ← [0..|xs| − 1)x_i ∈ xs(i..|xs| − 1]];
aux ordenada (xs: [T]) : Bool = (∀i ← [0..|xs| − 1))xs_i ≤ xs_{i+1};
```

Ejercicio 1. [20 puntos]

a) Completar el *invariante todosLosProductosTodosLosMeses* del tipo Supermercado, que garantiza que todos los productos incluidos en la lista de los precios se encuentran en alguna de las góndolas, que todos los precios incluidos en la lista de los precios son mayores que cero (estrictamente) y que están presentes en todos los meses.

b) Completar el *invariante changosCopados* del tipo Supermercado, que indica que en todos los changos del supermercado se cumple el hecho de que si poseen productos, estos productos se encuentran en alguna gondola del supermercado (con cantidad mayor o igual a cero).

Ejercicio 2. [20 puntos] Especificar el `problema gondolasChinas` (s: Supermercado, m: Mes) = `result` : [Seccion]
que devuelve aquellas secciones cuyas gondolas poseen más artículos importados que nacionales y además no poseen un artículo nacional más barato que alguno de los artículos importados en el mes *m*.

Ejercicio 3. [20 puntos] Especificar el `problema expulsarAntiPatrias` (s: Supermercado)
que modifica el supermercado de forma tal que saca a todos aquellos clientes (*changos*) que sólo compran más de una unidad de cierto producto sólo si es importado. En decir, que no tienen en su chango más de una unidad del mismo producto nacional y si tienen más de un mismo producto, éste es importado.

Ejercicio 4. [20 puntos] Especificar el `problema supersSinMimos` (ss: [Supermercado]) = `result` : [Supermercado] , que devuelve aquellos supermercados de la lista *ss* en los cuales los precios de todos los productos tuvieron un aumento constante, estrictamente creciente, mes a mes (desde el 1 hasta el 12 inclusive).

Ejercicio 5. [20 puntos] Dado el siguiente programa, decidir si es correcto para las siguientes especificaciones. En caso afirmativo, demostrar utilizando transformación de estados. En caso contrario, justificar por qué no es correcto.

```
int sumaCaprichosa(int a, int b){
  int x = 0;
  int y = 0;
  int c = 0;
  int result = 0;
  x=a+1;
  y=b+1;
  if(a > b){
   c = x - y;
  }else{
   c = y - x;
  }
  result = x + y + c;
  return result;
}
```

```
problema sumaCaprichosa (a,b: ℤ) = res : ℤ {
    requiere a == b;
    asegura res == a + b;
}
```

```
problema sumaCaprichosa (a,b: ℤ) = res : ℤ {
    requiere a > 0;
    requiere b > 0;
    asegura a > b ⇒ res == 2a + 2;
    asegura a ≤ b ⇒ res == 2b + 2;
}
```

Primer Parcial 44

Algorítmos I

Fecha 14/09/2016

a)

1) invariante todosLosProductosTodosLosMeses:
(∀ p ← productosG(s), m ← [1..12])
precios(s, p, m) > 0

aux productosG(s: supermercado): [Producto] =
[p | g ← gondolas(s), p ← productos(g)]

b) invariante changosCopados:
(∀ c ← changos(s), p ← productos(c))
p ∈ productosG(s)

(Cantidad mayor o igual a cero ya está garantizado por los invariantes)

Primer Parcial

44

Algorítmos I

Fecha 14/09/2016

2) problema gondolasChinas (s: Supermercado, m: Mes) =
result: [Seccion] {
  requiere 1 ≤ m ≤ 12;

  asegura ((∀ sec ← result) masImportados(sec, s)
    ∧ noHayNMasBarato(sec, m, s));

  asegura ((∀ sec ← secciones(gondolas(s)))
    (masImportados(sec, s) ∧ noHayNMasBarato(sec, m, s))
    ⟹ sec ∈ result);

  asegura sinRepetidos(result);

}

aux masImportados(sec: Seccion, s: Supermercado): Bool =
  cantImp(gondSec(sec, s)) > cantNac(gondSec(sec, s))

aux noHayNMasBarato(sec: Seccion, m: Mes, s: Supermercado): Bool =
  ¬ (∃ pn ← prodsNac(gondSec(sec, s)))
    (∀ pi ← prodsImp(gondSec(sec, s)))
      precios(s, pn, m) < precios(s, pi, m)

aux gondSec(sec: Seccion, s: Supermercado): Gondola =
  cab([g | g ← gondolas(s), seccion(g) == sec])

aux prodsNac(g: Gondola): [Producto] =
  [p | p ← productos(g), origen(p) == Nacional]

aux prodsImp(g: Gondola): [Producto] =
  [p | p ← productos(g), origen(p) == Importado]

aux cantImp(g: Gondola): ℤ = |prodsNac(g)|

aux cantNac(g: Gondola): ℤ = |prodsImp(g)|

```
3) problem expulsarAntiPatrias (s: Supermercado) {

    modifica s;

    asegura infoSinoCambio:
                mismasGondolas(pre(s), s)
            ∧ mismosPrecios(pre(s), s);

    asegura changosOK:
                |patriotas(pre(s))| <= |changos(s)|
            ∧ (∀ c ← patriotas(pre(s)))
                cuentaPatIguales(c, pre(s)) == cuentaPatIguales(c, s);

}


aux mismasGondolas(s1, s2 : Supermercado): Bool =
    (∀ g1 ← gondolas(s1)) (∃ g2 ← gondolas(s2))
        (seccion(g1) == seccion(g2)
        ∧ mismos(productos(g1), productos(g2))
        ∧ (∀ p ← productos(g1)) stock(g1, p) == stock(g2, p))

aux mismosPrecios(s1, s2: Supermercado): Bool =
    (∀ p ← productosG(s1), m ← [1..12]) precios(s1, p, m) == precios(s2, p, m)
                aux ejercicio 1.

aux patriotas(s: Supermercado): [Chango] =
    [c | c ← changos(s), patriota(c)]

aux patriota(c: Chango): Bool =
    (∀ p ← productos(c), cantidadXProducto(c, p) > 1)
        origen(p) == Importado

aux cuentaPatIguales(c1: Chango, s: Supermercado): ℤ =
    |[1 | c2 ← changos(s), mismoChango(c1, c2)]|

aux mismoChango(c1: Chango, c2: Chango): Bool =
    mismos(productos(c1), productos(c2))
    ∧ (∀ p ← productos(c1))
        cantidadXProducto(c1, p) == cantidadXProducto(c2, p)
```

Primer Parcial 44

Algorítmos I

Fecha 14/09/2016

4) problema supersSinMismos(ss:[Supermercado]) =
result: [Supermercado] {

asegura {SupsConAumentoConstante(ss)| == |result|;
(pido que en caso de haber sups. iguales, solo se repiten los repetidos de ss)

asegura (∀ s ← supsConAumentoConstante (ss))
cantSupIguales(s, ss) == cantSupIguales(s, result)

• OK (Aunque es indirecto..)

}

aux supsConAumentoConstante(ss:[Supermercado]): [Supermercado] =
[s | s ← ss, supConAumentoConstante(s)]

aux supConAumentoConstante(s: Supermercado): Bool =
(∀ p ← productos(s)) aumentoConstante(p, s)
aux de ej 1.

aux aumentoConstante(p: Producto, s: Supermercado): Bool =
ordEstCreciente([precios(s, p, m) | m ← [1..12]])

aux ordEstCreciente(l: [ℤ]): Bool =
(∀ i ← [0..|l|-1)) l_i < l_{i+1}

aux cantSupIguales(s1: Supermercado, ss: [Supermercado])
: ℤ =
|[1 | s2 ← ss, mismosSup(s1, s2)]|

aux mismosSup(s1, s2: Supermercado): Bool =
mismasGondolas(s1, s2)
aux ej 3
∧ mismosPrecios(s1, s2)
aux ej 3
∧ mismosChangos(s1, s2)

aux mismosChangos(s1, s2: Supermercado): Bool =
(∀ c ← changos(s1)) cantChangos(c, s1) == cantChangos(c, s2)
∧ |changos(s1)| == |changos(s2)|
aux cantChangos(c1: Chango, s: Supermercado): ℤ =
|[1 | c2 ← changos(s), mismoChango(c1, c2)]|
aux ej 3.

Primer Parcial 44

Algorítmos I

Fecha 14/09/2016

5) Caso 1 no cumple y caso 2 si cumple con la especif.
Realizo transformación de estados y justifico.

```
int sumaCaprichosa(int a, int b) {
    // estado e0
    // vale a==pre(a) ∧ b==pre(b)   (Esto lo asumo directamente en todo el ej. ya que a y b no cambian su valor durante el prog.. No uso clausura local)

    int x=0;
    // estado e1
    // vale x==0
    int y=0;
    // estado e2
    // vale y==0 ∧ x==x@e1
    // implica x==0                 (por e1)
    int c=0;
    // estado e3
    // vale c==0 ∧ x==x@e2 ∧ y==y@e2
    // implica x==0 ∧ y==0          (por e2)
    int result=0;
    // estado e4
    // vale result==0 ∧ c==c@e3 ∧ x==x@e3 ∧ y==y@e3
    // implica c==0 ∧ x==0 ∧ y==0 (por e3)
    if (a>b) {
        // estado t1
        // vale (a>b) ∧ result==result@e4 ∧ c==c@e4
        //      ∧ x==x@e4 ∧ y==y@e4
        // implica result==0 ∧ c==0 ∧ x!=0 ∧ y!=0 (por e4)
        c = x-y;
        // estado t2
        // vale (a>b) ∧ c==x@t1-y@t1
    }
    x = a+1;
    // estado e5
    // vale x==a+1 ∧ c==c@e4 ∧ y==y@e4
    //      ∧ result==result@e4
    // implica c==0 ∧ y==0 ∧ result==0 (por e4)
    y = b+1;
    // estado e6
    // vale y==b+1 ∧ x==x@e5 ∧ c==c@e5 ∧ result==result@e5
    // implica x==a+1 ∧ c==0 ∧ result==0.   (por e5)
    if(a>b) {
        // estado t1
        // vale (a>b) ∧ result==result@e6 ∧ c==c@e6
        //      ∧ x==x@e6 ∧ y==y@e6
        // implica result==0 ∧ c==0 ∧ x==a+1 ∧ y==b+1 (por e6)
        c = x-y;
        // estado t2
        // vale (a>b) ∧ c==x@t1-y@t1 ∧ result==result@t1 ∧ x==x@t1
        //      ∧ y==y@t1
```

5)

// implica c == a+1-b-1 ∧ x == a+1 ∧ y == b+1 (por t1)
// implica result == 0 (por t1)
// implica c == a-b
// implica Qif: result == 0 ∧ x == a+1 ∧ y == b+1
∧ ((a > b ∧ c == a-b) ∨ (a ≤ b ∧ c == b-a)) (por impl. anteriores y porque p ⇒ p∨q es tautología)

} } else {
// estado e1
// vale (a ≤ b) ∧ result == result@e6 ∧ c == c@e6
∧ x == x@e6 ∧ y == y@e6
// implica result == 0 ∧ c == 0 ∧ x == a+1 ∧ y == b+1 (por e6)
c == y - x;
// estado e2
// vale (a ≤ b) ∧ c == y@e1 - x@e1 ∧ result == result@e1
∧ x == x@e1 ∧ y == y@e1
// implica x == a+1 ∧ y == b+1 ∧ result == 0 (por e1)
// implica c == b+1-b-1
// implica c == b-a.
// implica Qif: result == 0 ∧ x == a+1 ∧ y == b+1
∧ ((a > b ∧ c == a-b) ∨ (a ≤ b ∧ c == b-a)) (por implicas anteriores y porque q ⇒ p∨q es tauto.)
}
// estado e7
// vale Qif: result == result@e6 ∧ x == x@e6 ∧ y == y@e6
∧ ((a > b ∧ c == a-b) ∨ (a ≤ b ∧ c == b-a))
// implica result == 0 ∧ x == a+1 ∧ y == b+1 (por e6).
result = x + y + c;
// estado e8
// vale result == x@e7 + y@e7 + c@e7 ∧ x == x@e7 ∧ y == y@e7
∧ c == c@e7
// implica x == a+1 ∧ y == b+1 (por e7)
return result;
// vale res == result@e8 ∧ result == result@e8
∧ x == x@e8 ∧ y == y@e8 ∧ c == c@e8
// implica res == a+1+b+1+c@e7 ∧ result == result@e7
∧ x == a+1 ∧ y == b+1 ∧ c == c@e7. (por e8)
// implica res == a+b+2+c@e7.
// (Separo en caso 1:)
// implica a ≤ b (por req a == b)
// implica result == a+b+2+b-a
// " result == a+a+2+a-a (por req)
// " result == 2a+2 ∧ res == 2a+2 (Como conclusión, no hay forma de implicar el asegura res == a+b, porque a+b equiv a 2a por req y 2a ≠ 2a+2)

// (Separo en caso 2!)
// implica (a > b ∧ res == a+b+2+a-b)
v (a ≤ b ∧ res == a+b+2+b-a)
// implica (a > b ∧ res == 2a+2)
// implica (a ≤ b ∧ res == 2b+2)

(por c en e7)

(Estos implicas son equivalentes a los asegurs del problema caso2)

// A:= a>b ∧ B:= res== 2a+2 ∧ C:= res== 2b+2.
// desde de equivalencia

/*

A	B	C	(A => B)	∧	(¬A => C)	(A ∧ B)	v	(¬A ∧ C)
V	V	V	V	V	V	V	V	F
V	V	F	V	V	V	V	V	F
V	F	V	F	F	V	F	F	F
V	F	F	F	F	V	F	F	F
F	V	V	V	V	V	F	V	V
F	V	F	V	F	F	F	F	F
F	F	V	V	V	V	F	V	V
F	F	F	V	F	F	F	F	F

=

*/

}