

# Introducción a la Programación

## Segundo Entregable de laboratorio

### Ejercicio 1

Implementar la función `quienGana(j1: str, j2: str) ->str` , cuya especificación es la siguiente:

```
problema quienGana (in j1: seq<Char>, in j2: seq<Char>) : seq<Char> {  
    requiere: {juegaBien(j1) ∧ juegaBien(j2)}  
    asegura: {gana(j1, j2) → res = "Jugador1"}  
    asegura: {gana(j2, j1) → res = "Jugador2"}  
    asegura: {(¬gana(j1, j2) ∧ ¬gana(j2, j1)) → res = "Empate"}  
}  
  
pred juegaBien (j: seq<Char>) {  
    j = "Piedra" ∨ j = "Papel" ∨ j = "Tijera"  
}  
  
pred gana (j1, j2: seq<Char>) {  
    piedraGanaAtijera(j1, j2) ∨ tijeraGanaAPapel(j1, j2) ∨ papelGanaAPiedra(j1, j2)  
}  
  
pred piedraGanaAtijera (j1, j2: seq<Char>) {  
    j1 = "Piedra" ∧ j2 = "Tijera"  
}  
  
pred tijeraGanaAPapel (j1, j2: seq<Char>) {  
    j1 = "Tijera" ∧ j2 = "Papel"  
}  
  
pred papelGanaAPiedra (j1, j2: seq<Char>) {  
    j1 = "Papel" ∧ j2 = "Piedra"  
}
```

Aclaración: Respetar los nombres de “Piedra“, “Papel“ y “Tijera“ (empieza con mayúsculas)

# Introducción a la Programación

## Segundo Entregable de laboratorio

### Ejercicio 2

Implementar la función `fibonacciNoRecursivo(n: int) ->int` , cuya especificación es la siguiente, **sin utilizar recursión**:

```
problema fibonacciNoRecursivo (in n:  $\mathbb{Z}$ ) :  $\mathbb{Z}$  {  
    requiere:  $\{n \geq 0\}$   
    asegura:  $\{(\exists l : seq\langle \mathbb{Z} \rangle)(|l| = n + 1 \wedge esSecuenciaFibonacci(l) \wedge l[|l| - 1] = result)\}$   
}  
  
pred esSecuenciaFibonacci (l:  $seq\langle \mathbb{Z} \rangle$ ) {  
    ( $|l| > 0 \rightarrow l[0] = 0$ )  $\wedge$   
    ( $|l| > 1 \rightarrow l[1] = 1$ )  $\wedge$   
    ( $\forall i : \mathbb{Z})(2 \leq i < |l| \rightarrow l[i] = l[i - 1] + l[i - 2])$   
}
```

# Introducción a la Programación

## Segundo Entregable de laboratorio

### Ejercicio 3

Implementar la función `mesetaMasLarga(...) ->int` , cuya especificación es la siguiente:

```
problema mesetaMasLarga (in l: seq(Z)) : Z {  
  requiere: {True}  
  asegura: {hayMesetaDeLong(l,result) ∧ ¬hayMesetaDeLong(l,result + 1)}  
}  
  
pred hayMesetaDeLong (l: seq(Z), n: Z) {  
  (∃i : Z)(∃j : Z)(0 ≤ i ∧ i ≤ j < |l| ∧ j - i + 1 = n ∧ todosIguales(l,i,j))  
}  
  
pred todosIguales (l: seq(Z), i: Z, j: Z) {  
  (∀k : Z)(i ≤ k ≤ j → l[k] = l[i])  
}
```

# Introducción a la Programación

## Segundo Entregable de laboratorio

### Ejercicio 4

Implementar la función `def filasParecidas(...) -> bool`, cuya especificación es la siguiente:

```
problema filasParecidas (in m: seq<seq<Z>>) : Z {  
    requiere: {esMatriz(m)}  
    asegura: {res = True  $\leftrightarrow$  ( $\exists n : \mathbb{Z}$ )(filasParecidasAanterior(m,n))}  
}  
  
pred esMatriz (m: seq<seq<Z>>) {  
    |m| > 0  $\wedge$  |m[0]| > 0  $\wedge$  ( $\forall i : \mathbb{Z}$ )(0  $\leq i < |m| \rightarrow |m[i]| = |m[0]|$ )  
}  
  
pred filasParecidasAanterior (m: seq<seq<Z>>, n: Z) {  
    ( $\forall i : \mathbb{Z}$ )(1  $\leq i < |m| \rightarrow filaAnteriorMasN(m,i,n)$ )  
}  
  
pred filaAnteriorMasN (m: seq<seq<Z>>, i, n: Z) {  
    ( $\forall j : \mathbb{Z}$ )(0  $\leq j < |m[0]| \rightarrow m[i][j] = m[i-1][j] + n$ )  
}
```

# Introducción a la Programación

## Segundo Entregable de laboratorio

### Ejercicio 5

Contamos con un listado de vuelos, donde cada vuelo está representado por dos datos: la ciudad de la cual parte el vuelo (origen), y la ciudad a la cual llega (destino). Implementar la función **sePuedeLlegar**, que indica si hay una forma de llegar desde una ciudad a otra utilizando las conexiones disponibles en el listado de vuelos, y en caso afirmativo devuelve la cantidad de vuelos que deben tomarse.

La especificación detallada de la función que debe implementarse es la siguiente:

```
problema sePuedeLlegar (in origen: String, in destino: String, in vuelos: seq⟨String × String⟩) : ℤ {
  requiere: {origen ≠ destino}
  requiere: {soloParteUnVueloDeCadaCiudad(vuelos) ∧ soloLlegaUnVueloAcadaCiudad(vuelos) ∧ sinRepetidos(vuelos)}
  asegura: {hayRuta(vuelos, origen, destino) → largoDeRuta(vuelos, origen, destino, res)}
  asegura: {¬hayRuta(vuelos, origen, destino) → res = -1}
}

pred soloParteUnVueloDeCadaCiudad (vuelos: seq⟨String × String⟩) {
  (∀i, j : ℤ)((0 ≤ i < |vuelos| ∧ 0 ≤ j < |vuelos| ∧ i ≠ j) → vuelos[i]0 ≠ vuelos[j]0)
}

pred soloLlegaUnVueloAcadaCiudad (vuelos: seq⟨String × String⟩) {
  (∀i, j : ℤ)((0 ≤ i < |vuelos| ∧ 0 ≤ j < |vuelos| ∧ i ≠ j) → vuelos[i]1 ≠ vuelos[j]1)
}

pred hayRuta (vuelos: seq⟨String × String⟩, origen, destino: String) {
  (∃ruta : seq⟨String × String⟩)(vuelosValidos(ruta, vuelos) ∧ |ruta| ≥ 1 ∧ ruta[0]0 = origen ∧ ruta[|ruta| - 1]1 =
    destino ∧ caminoDeVuelos(ruta))
}

pred vuelosValidos (ruta, vuelos: seq⟨String × String⟩) {
  sinRepetidos(ruta) ∧ (∀v : String × String)(v ∈ ruta → v ∈ vuelos)
}

pred caminoDeVuelos (vuelos: seq⟨String × String⟩) {
  (∀i : ℤ)(1 ≤ i < |ruta| → ruta[i]0 = ruta[i - 1]1)
}

pred largoDeRuta (ruta: seq⟨String × String⟩, origen, destino: String, longCamino: ℤ) {
  (∃ruta : seq⟨String × String⟩)(vuelosValidos(ruta, vuelos) ∧ |ruta| ≥ 1 ∧ ruta[0]0 = origen ∧ ruta[|ruta| - 1]1 =
    destino ∧ caminoDeVuelos(ruta) ∧ |ruta| = longCamino)
}
```

**Nota:** Si desean realizar pruebas, se espera que el input tenga el siguiente formato:

origen

destino

ciudadOrigen [COMA] ciudadDestino [ESPACIO] ciudadOrigen [COMA] ciudadDestino...

Por ejemplo:

rosario jujuy misiones,jujuy salta,chubut rosario,misiones
--

Notar que cada vuelo se separa del siguiente mediante un espacio, y las ciudades origen y destino de un vuelo se separan por una coma

En este caso, el origen sería *Rosario*, el destino *Jujuy*, y existen 3 vuelos (*Misiones* → *Jujuy*, *Salta* → *Chubut* y *Rosario* → *Misiones*)