## Práctica 1. PROGRAMACIÓN EN MAUDE

### Ejercicio 1

**Que devuelve `red in ... : consume(- ^ - ^ nil)`?**

El programa original devuelve `consume(- ^ - ^ nil)`. No se puede consumir una batería vacía, asi que devuelve la misma batería.

### Ejercicio 2

***EBattery* que no sea *ECell* y *Battery* que no sea *EBattery* ni *ECell***

El ejemplo de término de tipo *EBattery* pero no *ECell* seria la bateria nil.

Cualquier batería distinta de nil valdría para el segundo enunciado.

### Ejercicio 3

**Para que sirve diferenciar *ECell* dentro de *Cell*?**

Para poder representar casillas vacias y crear un nuevo comportamiento dentro del tipo *Cell*

### Ejercicio 4

Valores que pueden tomar un EBattery y un Battery con 2 celdas.

- *EBattery* solo puede ser `-` y `nil`
- *Battery* tiene 9 valores posibles (todas las combinaciones de `o`, `+` y `-` en 2 espacios)

### Ejercicio 5

**Que tipo devuelve la ejecución de `red in ... : consume(- ^ -) .`?**

Devuelve *EBattery*. Significa que sobre una batería vacía no se puede hacer `consume` porque no tiene ningún efecto.

### Ejercicio 6

**Qué se obtiene al ejecutar `red in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) .`?**

Devuelve `consume-left-right(- ^ o ^ o)`. Intenta ejecutar una ecuación, pero en este caso casi todas las definiciones son reglas `rl` y por tanto el resultado no es determinista.

**Ejercicio 7**

**Qué se obtiene al ejecutar `rew in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) .?`**

Devuelve `Battery: - ^ + ^ +`. `rew` ejecuta una regla haciendo una sobreescritura.

**Ejercicio 8**

**Qué se obtiene al ejecutar `search in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) =>! Bt:Battery .?`**

```
Solution 1 (state 1)
states: 3  rewrites: 2 in 1ms cpu (0ms real) (2000 rewrites/second)
Bt --> - ^ + ^ o

Solution 2 (state 2)
states: 3  rewrites: 2 in 1ms cpu (0ms real) (2000 rewrites/second)
Bt --> - ^ o ^ +

No more solutions.
states: 3  rewrites: 2 in 1ms cpu (0ms real) (2000 rewrites/second)
```

`search` intenta hacer matching con las reglas definidas. En este caso, `=>!` busca estados finales que no puedan ser reescritos más veces que den lugar a una `Battery`.

**Ejercicio 9**

**Qué se obtiene al ejecutar `search in BATTERY-LEFT-RIGHT : consume-left-right(- ^ o ^ o) =>* Bt:Battery .` ?**

```
Solution 1 (state 0)
states: 1  rewrites: 0 in 0ms cpu (0ms real) (~ rewrites/second)
Bt --> consume-left-right(- ^ o ^ o)

Solution 2 (state 1)
states: 2  rewrites: 1 in 0ms cpu (0ms real) (~ rewrites/second)
Bt --> - ^ + ^ o

Solution 3 (state 2)
states: 3  rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
Bt --> - ^ o ^ +

No more solutions.
states: 3  rewrites: 2 in 0ms cpu (0ms real) (~ rewrites/second)
```

En este caso, =>* ejecuta reglas hasta encontrar resultados que hayan pasado por 0, 1 o más pasos hasta llegar a una `Battery`.