


Condiciones de aprobación:

Para aprobar es necesario <u>simultáneamente</u> : <ul style="list-style-type: none"> • obtener 8 puntos de 14 • obtener al menos la mitad de los puntos en cada paradigma. 	Y recordá: en todas tus respuestas sé puntual, no pierdas el foco de lo que se pregunta. Respuestas en exceso generales son tan malas como respuestas incompletas.	
---	--	---

Punto 1 (4 puntos - 2 c/ predicado)

Se sabe que una persona contrajo una enfermedad si tiene todos los síntomas de dicha enfermedad (Independientemente que además tenga otros síntomas). Disponemos de la siguiente base de conocimiento:

```

tiene(ana, fiebre).           sintoma(zika, sarpullido).
tiene(ana, sarpullido).       sintoma(zika, fiebre).
tiene(ana, dolorMuscular).    sintoma(gripeA, fiebre).
tiene(luis, fiebre).          sintoma(gripeA, dolorDeCabeza).
tiene(luis, congestionNasal). sintoma(gripeA, tos).
persona(ana).                 sintoma(gripeA, congestionNasal).
persona(luis).
persona(juan).

contrajo(Persona, Enfermedad):- persona(Persona),
    not((tiene(Persona, Sintoma), not(sintoma(Enfermedad, Sintoma)))).
estaSano(Persona):- persona(Persona),
    findall(Enfermedad, contrajo(Persona, Enfermedad), Lista),
    length(Lista, 0).
  
```

Para cada uno de los predicados definidos, responder y justificar conceptualmente: ¿Funcionan adecuadamente, tanto para consultas existenciales como de verificación?

- En caso negativo, corregirlos.
- En caso afirmativo, ¿están aprovechando las herramientas principales del paradigma?
 - En caso negativo, mejorar las soluciones.
 - En caso afirmativo, mostrar un ejemplo de consulta y respuesta.

Punto 2 (5 puntos - a:2 b:2 c:1)

Dada la siguiente función en haskell:

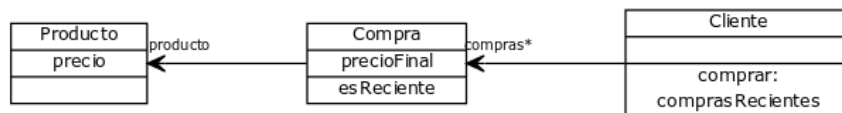
```

f h m p lista
| elem h lista = head (filter (m h) lista)
| otherwise = fst p
  
```

- Si la lista que recibe f tuviera infinitos elementos, ¿sería posible que esta función termine? Justificar por qué y mostrar al menos un ejemplo de invocación de la función que demuestre por qué sí o por qué no sería posible.
- La función f es de orden superior. ¿Cuál es la ventaja que aporta a la solución?
- Reescribir la función de dos formas: utilizando expresiones lambda y mediante composición de funciones.

Punto 3 (5 puntos - a:1,5 b:1 c:2,5)

Un equipo de desarrolladores está incorporando nuevos requerimientos para el sistema de ventas online para el que trabajan. Actualmente un cliente del sistema puede comprar productos al precio normal del mismo, para los cual se tiene algo así:



```
#Cliente
```

```
>> comprar: unProducto
```

```
compras add: (Compra de: unProducto precioFinal: unProducto precio)
```

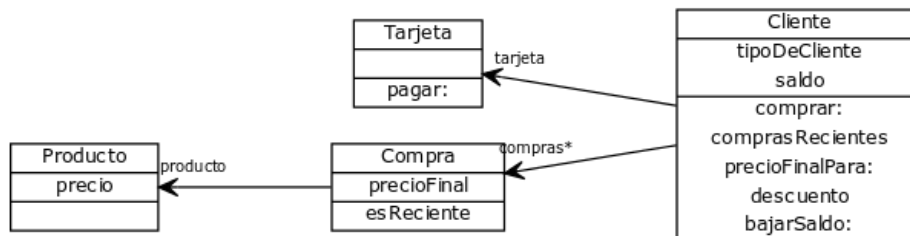
```
>> comprasRecientes
```

```
^ compras select: [:compra | compra esReciente]
```

Se quiere incorporar que haya distintos tipos de clientes (básico, gold o platinum) teniendo en cuenta que debería poder cambiar el tipo del cliente en cualquier momento. Para un cliente básico el precio final sería el precio del producto, a un cliente gold si realizó más de 3 compras recientemente, el precio final es el 90% del precio del producto (de lo contrario no se le realiza ningún descuento), y para un platinum siempre se descuenta el 25%.

Además hace falta validar que el cliente efectivamente pueda comprar el producto, ya que nos enteramos que los clientes básicos tienen un saldo en el sistema que se va decrementando a medida que compran (y si no le alcanza, no debería poder comprar el producto), mientras que a los gold y platinum se les permite la compra siempre, porque en vez de saldo tienen una tarjeta de crédito asociada.

Con esta información, se propuso la siguiente solución:



```
#Cliente
```

```
>> precioFinalPara: unProducto
```

```
^ unProducto precio - (self descuento * unProducto precio)
```

```
>> descuento
```

```
tipoDeCliente = 'platinum' ifTrue: [^ 0.25].
```

```
tipoDeCliente = 'gold' & (self comprasRecientes size > 3)
```

```
ifTrue: [^ 0.1] ifFalse: [^ 0]
```

```
>> comprar: unProducto
```

```
|precioFinal|
```

```
precioFinal := self precioFinalPara: unProducto.
```

```
tipoDeCliente = 'basico' ifTrue: [ self bajarSaldo: precioFinal ]
```

```
ifFalse: [self tarjeta pagar: precioFinal ].
```

```
compras add: (Compra de: unProducto precioFinal: precioFinal)
```

```
>> bajarSaldo: monto
```

```
self saldo < monto ifTrue: [ ^ 'No le alcanza el saldo' ].
```

```
self saldo: self saldo - monto.
```

- ¿Qué problemas trae aparejados modelar al tipo de cliente con un String? ¿De qué otra forma podría modelarse la existencia de distintos tipos de clientes de modo que siga siendo posible cumplir con lo especificado y se solucionen los problemas mencionados?
- ¿Es correcta la forma en la cual se evita comprar un producto si no hay saldo suficiente? Justificar.
- En base a las respuestas para a y b, corregir la solución (incluyendo código y diagrama de clases).

¹ **de:precioFinal:** es un método que entiende la clase Compra que crea una compra inicializada correctamente.