Primera Parte: (6 puntos)

Ejercicio 1 [2 puntos]
Un metaintérprete con pregunta es

```
prueba_p(verdad).                                    % cláusula 1
prueba_p((A & B)) :- prueba_p(A), prueba_p(B).       % cláusula 2
prueba_p(G) :- preguntable(G), respuesta(G,si).      % cláusula 3
prueba_p(G) :-                                       % cláusula 4
   preguntable(G),
   no_preguntado(G),
   pregunta(G,Respuesta),
   assert(respuesta(G,Respuesta)),
   Respuesta=si.
prueba_p(A) :- (A <- B), prueba_p(B).                % cláusula 5
```

1. Explicar cómo afecta al comportamiento del metaintérprete el suprimir la última condición (`Respuesta=si`) de la cuarta cláusula.
2. Explicar cómo afecta al comportamiento del metaintérprete el eliminar la cuarta condición (`assert(respuesta(G,Respuesta))`) de la cuarta cláusula y eliminar la tercera cláusula.

En los casos en que el comportamiento sea distinto, poner un ejemplo de base de conocimiento que demuestre la diferencia.

. . . . . .

Solución:

Al suprimir la última condición de la cuarta cláusula, considera la pregunta demostrada aunque la respuesta sea `no`.

Al eliminar la cuarta condición (`assert(respuesta(G,Respuesta))`) de la cuarta cláusula y eliminar la tercera cláusula, no se guardan las repuestas con lo que puede realizar más de una vez la misma pregunta.

Para mostrar las diferencias consideremos la siguiente base de conocimiento:

```
multiplo_de(6) <-
  divisible_por(2) &
  divisible_por(3).
multiplo_de(10) <-
  divisible_por(2) &
  divisible_por(5).
```

```
preguntable(divisible_por(_)).
```

Una sesión con el metaintérprete inicial es

```
?- prueba_p(multiplo_de(X)).
¿Es verdad divisible_por(2)? (si/no)
|: si.
¿Es verdad divisible_por(3)? (si/no)
|: no.
¿Es verdad divisible_por(5)? (si/no)
|: si.
X = 10
Yes
```

Con la primera modificación la sesión es

```
?- prueba_p(multiplo_de(X)).
¿Es verdad divisible_por(2)? (si/no)
|: si.
¿Es verdad divisible_por(3)? (si/no)
|: no.
X = 6
Yes
```

dando una respuesta incorrecta.

Con la segunda modificación la sesión es

```
?- prueba_p(multiplo_de(X)).
¿Es verdad divisible_por(2)? (si/no)
|: si.
¿Es verdad divisible_por(3)? (si/no)
|: no.
¿Es verdad divisible_por(2)? (si/no)
|: si.
¿Es verdad divisible_por(5)? (si/no)
|: si.
X = 10
Yes
```

preguntando dos veces si es divisible por 2.

Ejercicio 2 [2 puntos]

Se considera la siguiente base de conocimiento en CLIPS:

```
(defrule regla1
  ?h1 <- (dato1 $?i1 ?x $?f1)
  ?h2 <- (dato2 $?i2 ?x $?f2)
  =>
  (retract ?h1 ?h2)
  (assert (dato1 $?i1 $?f1)
          (dato2 $?i2 $?f2)))

(defrule regla2
  ?h1 <- (dato1 $? ?x $?)
  (not (dato2 $? ?x $?))
  ?h2 <- (dato2 $?)
  =>
  (retract ?h1 ?h2)
  (assert (respuesta NO)))

(defrule regla3
  ?h1 <- (dato1)
  ?h2 <- (dato2)
  =>
  (retract ?h1 ?h2)
  (assert (respuesta SI)))
```

Se pide:

1. Construir una tabla de seguimiento con el siguiente conjunto de hechos iniciales. ¿Qué hechos quedan en la base de conocimiento al terminar la ejecución?.

```
(deffacts ej1
  (dato1 1 2 3 1)
  (dato2 2 1 1 3))
```

2. Construir una tabla de seguimiento con el siguiente conjunto de hechos iniciales. ¿Qué hechos quedan en la base de conocimiento al terminar la ejecución?.

```
(deffacts ej1
  (dato1 1 2 3 1)
  (dato2 2 1 2 3))
```

3. Explicar brevemente el comportamiento del conjunto de reglas anterior.

4. Existe alguna situación en la que, a partir de cierto conjunto de hechos iniciales del tipo `(dato1 $?)` y `(dato2 $?)`, al terminar la ejecución no aparezca en el conjunto de hechos resultantes ni `(respuesta NO)` ni `(respuesta SI)`. Si es así, proporcionar un conjunto de hechos iniciales con los que ocurra esto y construir la correspondiente tabla de seguimiento. Proporcionar una regla `regla4` que arregle esta situación manteniendo el comportamiento de la base de conocimiento.

5. Escribir un predicado Prolog `comprueba(L1,L2)`, tal que si `L1` es la lista de elementos almacenada en el hecho `(dato1 $?l1)` y `L2` es la lista de elementos almacenada en el hecho `(dato2 $?l2)` entonces el predicado `comprueba(L1,L2)` tiene éxito si y sólo si al terminar la ejecución de la base de conocimiento CLIPS se obtiene el hecho `(respuesta SI)`. Si se obtiene el hecho `(respuesta NO)`, el predicado `comprueba(L1,L2)` falla.

. . . . . .

Solución:

Apartado 2.1

Hechos	E	S	Agenda	D	S
f0 (initial-fact)	0				
f1 (dato1 1 2 3 1)	0	1			
f2 (dato2 2 1 1 3)	0	1	regla1: f1 ($?i1 = ())		
			f2 ($?i2 = (2))	-	1
			regla1: f1 ($?i1 = ())		
			f2 ($?i2 = (2 1))	-	1
			regla1: f1 ($?i1 = (1))		
			f2 ($?i2 = ())	-	1
			regla1: f1 ($?i1 = (1 2))		
			f2 ($?i2 = (2 1 1))	-	1
			regla1: f1 ($?i1 = (1 2 3))		
			f2 ($?i2 = (2))	-	1
			regla1: f1 ($?i1 = (1 2 3))		
			f2 ($?i2 = (2 1))	1	
f3 (dato1 1 2 3)	1	2			
f4 (dato2 2 1 3)	1	2	regla1: f3 ($?i1 = ())		
			f4 ($?i2 = (2))	-	2
			regla1: f3 ($?i1 = (1))		
			f4 ($?i2 = ())	-	2
			regla1: f3 ($?i1 = (1 2))		
			f4 ($?i2 = (2 1))	2	
f5 (dato1 1 2)	2	3			
f6 (dato2 2 1)	2	3	regla1: f5 ($?i1 = ())		
			f6 ($?i2 = (2))	-	3
			regla1: f5 ($?i1 = (1))		
			f6 ($?i2 = ())	3	

f7 (dato1 1) f8 (dato2 1)	3 3	4 4	regla1: f7 ($?i1 = ()) f8 ($?i2 = ())	4	
f9 (dato1) f10 (dato2)	4 4	5 5	regla3: f9, f10	5	
f11 (respuesta SI)	5				

Los hechos que quedan en la base de conocimiento son `(respuesta SI)` y, posiblemente, `(initial-fact)`.

Apartado 2.2

Hechos	E	S	Agenda	D	S
f0 (initial-fact)	0				
f1 (dato1 1 2 3 1)	0	1			
f2 (dato2 2 1 2 3)	0	1	regla1: f1 ($?i1 = ())		
			f2 ($?i2 = (2))	-	1
			regla1: f1 ($?i1 = (1))		
			f2 ($?i2 = ())	-	1
			regla1: f1 ($?i1 = (1))		
			f2 ($?i2 = (2 1))	-	1
			regla1: f1 ($?i1 = (1 2))		
			f2 ($?i2 = (2 1 2))	-	1
			regla1: f1 ($?i1 = (1 2 3))		
			f2 ($?i2 = (2))	1	
f3 (dato1 1 2 3)	1	2			
f4 (dato2 2 2 3)	1	2	regla1: f3 ($?i1 = (1))		
			f4 ($?i2 = ())	-	2
			regla1: f3 ($?i1 = (1))		
			f4 ($?i2 = (2))	-	2
			regla1: f3 ($?i1 = (1 2))		
			f4 ($?i2 = (2 2))	-	2
			regla3: f3 ($?x = 1), ,f4	2	
f5 (respuesta NO)	2				

Los hechos que quedan en la base de conocimiento son `(respuesta NO)` y, posiblemente, `(initial-fact)`.

Apartado 2.3

Hay dos posibles explicaciones:

1. La base de conocimiento sirve para comprobar si un multiconjunto (`dato1`) está contenido en otro multiconjunto (`dato2`).

2. La base de conocimiento sirve para comprobar si dos secuencias son permutación la una de la otra (o multiconjuntos iguales).

Apartado 2.4

Una de tales situaciones se tiene con el siguiente conjunto de hechos iniciales: `(dato1)` y `(dato2 1)`. La tabla de seguimiento es la siguiente:

Hechos	E	S	Agenda	D	S
f0 (initial-fact)	0				
f1 (dato1)	0				
f2 (dato2 1)	0				

En función de la respuesta al apartado 2.3, una regla que arregla esta situación es:

1.
```
(defrule regla4
   ?h1 <- (dato2 $? ?x $?)
   (not (dato1 $? ?x $?))
   ?h2 <- (dato1 $?)
   =>
   (retract ?h1 ?h2)
   (assert (respuesta SI)))
```

2.
```
(defrule regla4
   ?h1 <- (dato2 $? ?x $?)
   (not (dato1 $? ?x $?))
   ?h2 <- (dato1 $?)
   =>
   (retract ?h1 ?h2)
   (assert (respuesta NO)))
```

Apartado 2.5

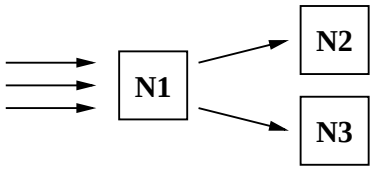
En función de la respuesta al apartado 2.3, un predicado Prolog `comprueba` viene dado por el siguiente conjunto de reglas:

1.
```
comprueba([],_).
comprueba([X|L1],L2) :-
   select(X,L2,L3),
   comprueba(L1,L3).
```

2.
```
comprueba([],[]).
comprueba([X|L1],L2) :-
   select(X,L2,L3),
   comprueba(L1,L3).
```

Ejercicio 3 [2 puntos]

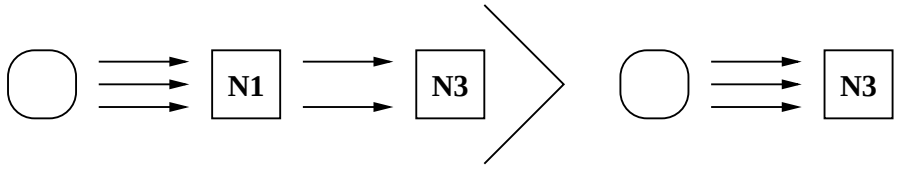
Consideremos una red de comunicaciones en la que existen una serie de nodos que reciben cierta información y la trasmiten a otros nodos. Un nodo puede recibir información de varios pero la transmite únicamente a otros dos, tal y como indica el gráfico:



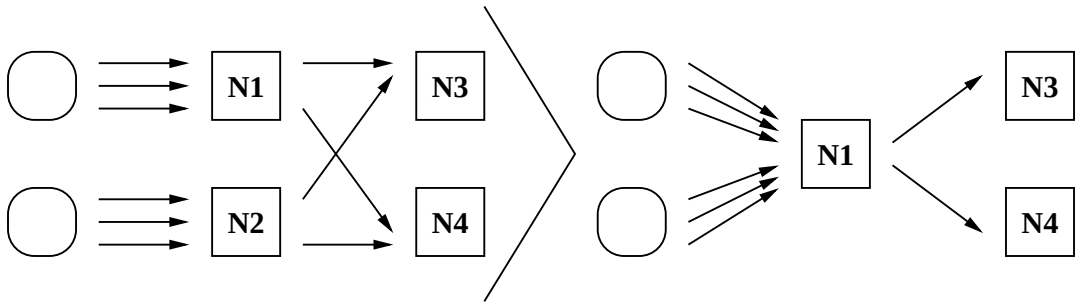
En este caso el nodo `N1` recibe información de otros nodo y la trasmite a los nodos `N2` y `N3`. Diremos que `N2` y `N3` son los nodos sucesores de `N1`.

Una red de comunicaciones puede simplificarse de acuerdo con dos reglas:

1. **Redundancia**: Un nodo de una red de comunicaciones es redundante si los dos nodos sucesores a los que trasmite la información son el mismo. En este caso se puede prescindir del nodo redundante y hacer que la información que recibe pase directamente a sus sucesores. Tal y como se indica en el gráfico:

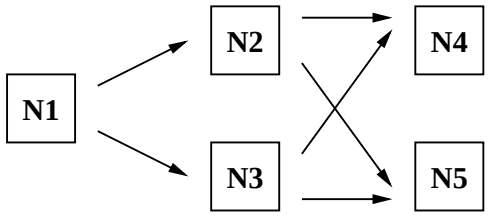


2. **Duplicidad**: Un nodo de una red de comunicaciones está duplicado si existe otro nodo distinto tal que los dos transmiten la información a los mismos sucesores. En este caso se puede prescindir de uno de los dos nodos y hacer que la información que recibe el eliminado sea transmitida al que mantenemos.



Se pide:

1. Construir una plantilla para almacenar la información acerca de un nodo y sus sucesores.
2. Construir un conjunto de reglas CLIPS que simplifiquen una red de comunicación de acuerdo con las reglas de redundancia y duplicidad.
3. Proporcionar un conjunto de hechos iniciales que describa la siguiente red de comunicaciones:



¿Qué conjunto de hechos queda en la base de conocimiento, cuando se utiliza el conjunto de reglas pedido en el apartado 2 sobre el conjunto de hechos pedido en este apartado?.

[**Nota:** *Para el desarrollo de este ejercicio no se permite el uso de condicionales (*`if ... then ... else`*), el uso de bucles (*`while ... do` *o* `loop-for-count`*), la definición de nuevas funciones (*`deffunction`*) ni el uso de prioridades (*`salience`*)*]

. . . . . .

Solución:

Apartado 3.1

```
(deftemplate nodo
  (slot id)
  (multislot hijos))
```

Apartado 3.2

```
(defrule nodo-redundante
  ?h <- (nodo (id ?v1) (hijos ?v2 ?v2))
  =>
  (retract ?h)
  (assert (cambia ?v1 ?v2)))

(defrule nodo-duplicado
  ?h1 <- (nodo (id ?v1) (hijos ?h1 ?h2))
  (or (nodo (id ?v2&~?v1) (hijos ?h1 ?h2))
      (nodo (id ?v2&~?v1) (hijos ?h2 ?h1)))
  =>
  (retract ?h1)
  (assert (cambia ?v1 ?v2)))

(defrule propaga-cambio
  (cambia ?v1 ?v2)
  ?h <- (nodo (hijos $?i ?v1 $?f))
  =>
  (modify ?h (hijos $?i ?v2 $?f)))
```

Apartado 3.3

```
(deffacts ejemplo
  (nodo (id n1) (hijos n2 n3))
  (nodo (id n2) (hijos n4 n5))
  (nodo (id n3) (hijos n4 n5)))
```

Al aplicar a este conjunto de hechos el conjunto de reglas mostrado en el apartado 2, quedan los siguientes hechos en la base de conocimiento: `(nodo (id n2) (hijos n4 n5))` y, posiblemente, `(initial-fact)`.
