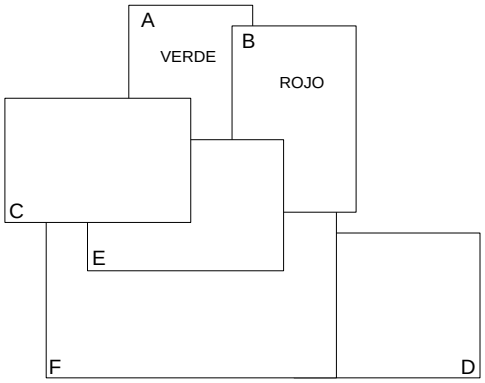
**Examen conv. 1 curso 96-97** **25 Enero 1997** **1 hora 45 minutos**

**Apellidos** **y** **Nombre:**
**.....**

## Problema 1: Búsquedas no informadas *[25 puntos] [15 minutos]*




**El problema de coloración de mapas.** El coloreado de mapas consiste en poner colores a los distintos paises evitando que dos paises adjacentes tengan el mismo color. Supongamos que podemos utilizar tres colores (**verde**, **rojo**, **azul**), y que se ha elegido verde para el pais A y rojo para el pais B. La única opción que queda para E es colorearlo de azul. Este es un típico problema de satisfacción de restricciones, sin embargo vamos a resolverlo aprovechando las posibilidades de resolución de problemas que nos proporcionan las búsquedas genéricas. El objetivo del presente problema es utilizar una búsqueda primero en profundidad para terminar de colorear (los paises A y B ya tienen color) el mapa de la figura anterior. Para hacer más sencilla la búsqueda vamos a incluir una restricción adicional**:** se deben ir coloreando los paises siguiendo el orden alfabético de sus nombres (habrá que buscar un color para C antes de buscar uno para D).

[**1.-** 5 puntos] Describe brevemente los operadores que se van a utilizar en la búsqueda.

[**2.-** 20 puntos] Dibuja el espacio de estados resultante de realizar la búsqueda hasta encontrar la primera solución, siguiendo extrictamente el orden de operadores que has descrito anteriormente (si no te cabe aquí en horizontal, dibújalo atrás en vertical). Escribe en el dibujo los colores de los paises que han resultado de tu búsqueda.

## Problema 2: Búsquedas no informadas/Common Lisp/Prácticas *[15 puntos] [10 minutos]*

El siguiente código corresponde al programa de búsqueda que has utilizado en la práctica 2.

```
(defconstant estado-inicial ...)
(defvar el-nodo-inicial nil "Nodo que plantea la situacion inicial")
(defvar nodos-a-expandir nil "Lista de estados a considerar")
(defvar nodos-expandidos nil "Lista de estados que ya han sido visitados una
vez")
(defun crea-nodo (estado padre &optional (simbolo (gensym "NODO-")))...)

 (defun mc ()
  (setq nodos-expandidos nil                          ;(1)
        el-nodo-inicial (crea-nodo estado-inicial nil 'nodo-inicial)
        nodos-a-expandir (list el-nodo-inicial)       ;(2)
        )
  (do* ((el-nodo (pop nodos-a-expandir)
                 (if nodos-a-expandir
                   (pop nodos-a-expandir)             ;(4)
                   (return (mendaje-de-fin))))      ;(3)
        )
       ((objetivop el-nodo)                           ;(5)
        (escribe-solucion el-nodo)
        )
    (push el-nodo nodos-expandidos)
    (reorganizar-nodos-a-expandir                     ;(7)
      (expandir-nodo el-nodo))))                      ;(6)

 (defun mensaje-de-fin ()
  (format t "~%~%Ya no es posible seguir con el proceso de busqueda."))

(defun escribe-solucion (solucion)
  (format t "~%~%Resuelto ! El camino de la solucion es: ")
  (escribe-un-camino solucion)
  'hecho)

(defun escribe-un-camino (nodo)
  ... )
```

Haciendo las actualizaciones oportunas, este código puede ser utilizado para realizar la búsqueda del problema de coloración de mapas. Con una llamada a `(mc)` conseguiríamos escribir por pantalla el camino para llegar a una solución (la primera que se encuentre). Indica que modificaciones (mínimas) se deben hacer para que se encuentren e impriman los caminos de todas las soluciones posibles (puedes indicar cambios en el propio código y si te falta sitio utiliza la parte de atrás de la hoja).

[**1.-** 10 puntos] Para una búsqueda primero en anchura.

[**2.-** 5 puntos] Para una búsqueda primero en profundidad.

## Problema 3: Búsquedas informadas *[25 puntos] [20 minutos]*

Para producir una pieza en un sistema de fabricación es necesario realizar una serie de operaciones en secuencia. Para realizar una operación se necesita una máquina y se consume cierto tiempo. Cuando son varias las piezas a realizar y hay un número reducido de máquinas, es necesario que algunas de las piezas esperen mientras que se están utilizando las máquinas disponibles para realizar operaciones en otras. Un típico problema de control de producción consiste en elegir adecuadamente el orden de realización de las operaciones consiguiendo de esta forma minimizar el tiempo total (tiempo en el que sale la última pieza - tiempo en el que entra la primera).

Supongamos que se dispone de un reducido sistema de fabricación con dos máquinas (M1 y M2) en el que se desea fabricar dos tipos de piezas (P1 y P2).
Las piezas de tipo P1 tienen la siquiente secuencia de operaciones**:**

| operacion | máquina | tiempo en seg. |
|---|---|---|
| op11 | M1 | 30 |
| op12 | M2 | 20 |

Las piezas de tipo P2 tienen la siquiente secuencia de operaciones**:**

| operacion | máquina | tiempo en seg. |
|---|---|---|
| op21 | M1 | 10 |
| op22 | M2 | 30 |
| op23 | M1 | 20 |

Se desea encontrar la secuencia de operaciones necesaria para fabricar **dos** piezas de tipo **P1** y **una** pieza de tipo **P2** en un tiempo mínimo. Para ello se va a utilizar una estrategia de búsqueda A. Dado un estado el tiempo transcurrido para llegar a ese estado es el intervalo de tiempo desde que comenzó la primera operación hasta el final de la última operación planeada (80 seg.). Para el tiempo que falta se utilizará la siguiente heurística**:** sumar todos los tiempos de las operaciones que faltan por realizar en la máquina M1 (20 seg.) al tiempo de finalización de la última operación planeada en M1 (70 seg.), hacer lo mismo para M2 (20 y 80 seg.), el máximo de estos dos tiempos es la estimación del tiempo total (100 seg.). El tiempo que falta será dicha estimación menos el tiempo transcurrido para llegar a ese estado (100 - 80 = 20 seg.).

A falta todavía de estas dos operaciones
Se pide**:**

[**1.-** 5 puntos] Justifica si con esa heurística se puede conseguir una solución óptima.

[**2.-** 20 puntos] Escribe la secuencia de operaciones a realizar en cada máquina y el tiempo total encontrados con esa búsqueda. Dibuja también el árbol de búsqueda que has generado para encontrar la solución con información clara de estapos y valores.

## Problema 4: Búsqueda en juegos *[20 puntos] [15 minutos]*

[**1.-** 8 puntos] Completa el árbol de juego dibujado arriba, rellenando los valores de todos los nodos que quedan sin valor, incluyendo el nodo raiz, utilizando búsqueda min-max.

[**2.-** 7 puntos] Señala con un círculo los nodos que serian podados en la poda alfa-beta.

[**3.-** 7 puntos] Reordena los nodos (de izquierda a derecha) de forma que den como resultado el máximo número de nodos podados. Reordena los hijos preservando las relaciones hijo-padre.

## Problema 5: Reglas/Clips/Prácticas *[15 puntos] [10 minutos]*

```
; El estado de la garrafa es
;
(deftemplate estado
              (slot garrafa
                  (type INTEGER)
                  (default 0)))
;---------------------------------------------------------------
; Regla para definirt el estado inicial

(defrule Estado-Inicial
        ?inicial <- (initial-fact)
=>
        (printout t "El estado inicial es la garrafa vacia" crlf)
        (assert (estado (garrafa 0))))
;---------------------------------------------------------------
; Regla para agnadir un litro a la garrafa
;
(defrule Agnade-Un-Litro
          ?estado <- (estado
                        (garrafa ?cantidad))
=>
          (printout t "Agnadiendo 1 litro a " ?cantidad " quedando "
             (+ 1 ?cantidad) " litros" crlf)
          (modify ?estado (garrafa (+ ?cantidad 1))))

;---------------------------------------------------------------
; Regla para agnadir dos litros a la garrafa
;
(defrule Agnade-Dos-Litros
          ?estado <- (estado
                        (garrafa ?cantidad))
=>
          (printout t "Agnadiendo 2 litros a " ?cantidad " quedando "
             (+ 2 ?cantidad) " litros" crlf)
   (modify ?estado (garrafa (+ ?cantidad 2))))

;---------------------------------------------------------------
; Regla para detectar tarea realizada (Es decir la garrafa tiene
; 3 litros).
(defrule Hecho
   ?estado <- (estado
                        (garrafa 3))
=>
          (printout t "Hecho - La garrafa tiene tres litros." crlf))
```

Escribe los resultados de la siguiente secuencia de llamadas**:**

```
CLIPS> (reset)
.....................................................................
CLIPS> (run 1)
.....................................................................
CLIPS> (run 1)
.....................................................................
CLIPS> (run 1)
.....................................................................
CLIPS> (run 1)
.....................................................................
CLIPS> (run 1)
.....................................................................
CLIPS> (run 1)
.....................................................................
```

## Problema 6: Frames/Common Lisp/Prácticas *[25 puntos] [15 minutos]*

En la práctica 4 se utilizaba un esquema de representación llamado IERL para almacenar la información de una base de conocimiento de una inmobiliaria. Se desea ahora ampliar la base de conocimiento para que todas las viviendas puedan almacenar la información de su precio con un atributo adecuado. Adicionalmente, la inmobiliaria dispone de unos márgenes de precio normales de las viviendas y desea que si se introduce algún precio que sobrepase estos márgenes se escriba también un mensaje por pantalla indicando el nombre de la calle, el número y el rango de precios recomendado para esa vivienda. Así se recomienda que el precio de los chalets esté comprendido entre 5 y 50 millones y el precio de los apartamentos entre 1 y 30 millones. Puedes asumir que IERL dispone ya de todos los mecanismos necesarios para utilizar demons.

Se pide que amplies la base de conocimiento con los atributos y demons adecuados para contemplar todo lo relativo al precio. Recuerda que es importante la elegancia del código y que resulte lo más reducido posible. A continuación se lista el código para crear la base de conocimiento y alguna función usada por los demons que te puede servir de ayuda para crear tu propio código.

```
;;; CALCULA-RUIDO es una funcion IF-NEEDED que calcula el nivel de ruido de un apartamento.
;;; Si existe un valor = lo toma, si no lo calcula dependiendo de la altura del piso.
;;;
 (defun calcula-ruido (form slot)
  (let* ((valor (find-aspect-from-supers form 'ruido-calle '=))
         piso)
    (cond (valor valor)
          (t (setq piso (get-value form 'piso))
             (cond ((> piso 15) 'muy-bajo) ((> piso 8) 'bajo) ((> piso 4) 'medio)
                   ((> piso 1) 'alto) (t 'muy-alto))))))

(defun gbd ()
  (setq  *tabla-de-forms* nil)
  (let ((la-form (create-form :is-a nil :name 'object))
        )
(form :name 'cosa
          :is-a (find-form 'object))

    (form :name 'vivienda
          :is-a (find-form 'cosa))

    (form :name 'apartamento
          :is-a (find-form 'vivienda)
          :slots (list (list 'ruido-calle 'if-needed #'calcula-ruido)))

    (form :name 'chalet
          :is-a (find-form 'vivienda)
          :slots (list (list 'ruido-calle 'if-needed #'calcula-ruido-en-casa) ) )

    (form :name 'apt-breton-22
          :is-a (find-form 'apartamento)
          :slots '((calle-nombre = Breton)
                   (calle-numero = 22)
                   (color-pared = blanco)
                   (material-suelo = madera)))

    (form :name 'apt-22-1
          :is-a (find-form 'apt-breton-22)
          :slots '((numero-habitaciones = 3)
                   (piso = 2)))

    (form :name 'chalet-montesol-5
          :is-a (find-form 'chalet)
          :slots '((calle-nombre = urbanizacion-montesol)
                   (calle-numero = 5)
                   (numero-habitaciones = 5)
                   (pisos = 2)
                   (distancia-carretera = 200)))))
```

**NOTA: Trata de hacer primero los problemas que te resulten más fáciles y cortos.**