**MEMORIA PRÁCTICA 2**

**Autores: Victoria Pelayo e Ignacio Rabuñal**

**Grupo: 2301 pareja 8**

**Ejercicio1:**

En este ejercicio nos piden implementar dos funciones para evaluar el valor de la heurística, según el tiempo o el precio, de un estado.

**Pseudocódigo de las funciones:**

**F-H-TIME (state sensors)**

input:

state: nombre de la ciudad (estado) actual

sensors: lista de los estados y sus correspondientes heurísticas

los elementos de la lista son (state (h-time h-price))

Output:

Res: el valor de la heurística del tiempo

Proceso:

i = 0

para i < longitud de sensors:

si state = valor-estado(sensors[i]):

devolver valor-heurisitica-time (sensors[i])

**F-H-PRICE (state, sensors)**

input:

state: nombre de la ciudad (estado) actual

sensors: lista de los estados y sus correspondientes heurísticas

los elementos de la lista son (state (h-time h-price))

Output:

Res: el valor de la heurística del precio

Proceso:

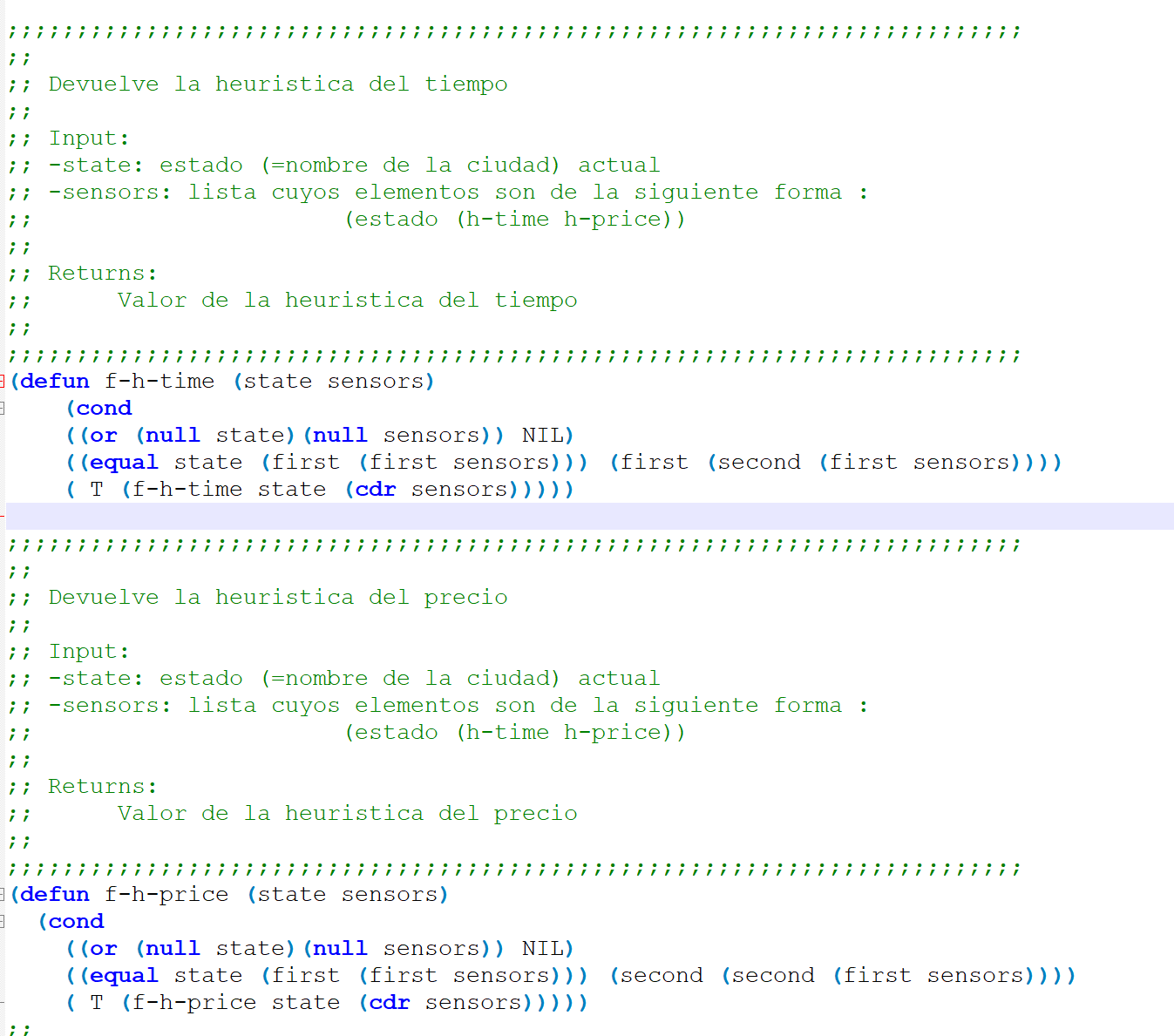
i = 0

para i < longitud de sensors:

si state = valor-estado(sensors[i]):

devolver valor-heurisitica-price (sensors[i])

Capturas del Código implementado:



**Ejemplos de ejecución ejercicio 1:**

**CG-USER**(21): (f-h-time 'Nantes \*estimate\*)

75.0

**CG-USER**(22): (f-h-time 'Marseille \*estimate\*)

145.0

**CG-USER**(35): (f-h-price 'Lyon \*estimate\*)

0.0

**CG-USER**(36): (f-h-price 'Madrid \*estimate\*)

NIL

**CG-USER**(37): (f-h-time 'Madrid \*estimate\*)

NIL

**Ejercicio2:**

En este ejercicio tenemos que implementar 4 funciones(navigate-canal-time, navigate-canal-price, navigate-train-time, navigate-train-price) que devolverán una lista de acciones que pueden realizar dado un estado actual. Por ejemplo, navigate-canal-time nos devolverá la lista de acciones posibles, partiendo de un estado, utilizando canales y tomará como heurística el tiempo.

Nos piden crear una función auxiliar navigate más genérica que recibe una función (“cfun”, una de las mencionadas anteriormente) y se encargará de devolver la lista de acciones posibles en función de la “cfun” pasada. Es importante la existencia de esta función, pues, evita repetir código.

Para este ejercicio también hemos creado otras tres funciones auxiliares: navigate-aux, que es la función recursiva de navigate; ciudad-permitida, que devuleve si una ciudad está permitida en el camino; get-time, que devuelve el coste del tiempo de un elemento determinado; get-price, que devuelve el coste del precio de un elemento determinado.

**Pseudocódigos de las funciones:**

**Ciudad-permitida(state, forbidden)**

Input: state (estado actual), forbidden (lista de estados prohibidos)

Output: T si está permitida y F si no es así

Proceso:

Si estado = null:

Devolver False

Si prohibidas = null:

Devolver False

Si estado = forbidden[0]:

Devolver True

Si no:

Devolver ciudad-permitida (state, resto de forbidden)

\*resto de forbidden = lista de forbidden a partir del elemento forbidden[1]

**Get-time (lst)**

Input: lst (lista de elementos de la forma (origin final (time price)) )

Output: Res(tiempo del primer elemento de la tabla)

Proceso:

Si lst = null:

Devolver null

Si no:

Devolver lst[0][2][0] (primer elemento del tercer elemento del primer elemento de la lista)

**Get-price (lst)**

Input: lst (lista de elementos de la forma (origin final (time price)) )

Output: Res(precio del primer elemento de la tabla)

Proceso:

Si lst = null:

Devolver null

Si no:

Devolver lst[1][2][0] (segundo elemento del tercer elemento del primer elemento de la lista)

**Navigate(state lst-edges cfun name(opcional) forbidden)**

Input: state(estado actual), lst-edges(lista de los nodos del grafo), cfun(nombre de la función para saber que coste usar), nombre (es opcional, nombre que le pondremos a la acción), forbidden (lista de los estados no permitidos)

Output: Res lista de acciones originadas a partir del estado dado

Proceso:

Res = navigate-aux(state, lst-edges, name, forbidden, () )

**Navigate-aux(state lst-edges cfun name(opcional) forbidden lst)**

Input: state(estado actual), lst-edges(lista de nodos del grafo), cfun(función para saber que coste obtener), name(combre de la acción), forbidden(lista de estados no permitidos), lst(lista de acciones)

Output: lst(lista de acciones realizadas a partir del estado)

Proceso:

Si (estado = null OR lst-edges = null):

Devolver null

Si (estado = lst-edges[0][0] AND True = ciudad-permitida(lst-edges[0][1], forbidden)):

Acción = nueva acción{

Name = name

Origin = state

Final = lst-edges[0][1]

Cost = cfun(lst-edges)

}

Añadir-a-lst(accion)

Añadir-a-lst(navigate-aux(state lst-edges cfun name forbidden lst))

Devolver lst

Si no:

Añadir-a-lst(navigate-aux(state lst-edges cfun name forbidden lst))

Devolver lst

En esta función de manera recursiva en la lista de grafos comprobamos si el grafo corresponde su estado actual al nuestro y si a donde lleva es una ciudad permitida, si esto es así añadimos una acción que creamos con estas características a nuestra lista de acciones y el resultado de aplicar esta función al resto de nodos de la lista. Si el primero no coincide devolvemos la lista tras añadirle el resultado de aplicar esta función al resto de la lista de nodos.

**Navigate-canal-time(state canals)**

Input: state(estado actual), canals(lista de canales)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del tiempo)

Proceso:

Deolver navigate (state, canals, función get-time, “navigate-canal-time”, null)

**Navigate-canal-price(state canals)**

Input: state(estado actual), canals(lista de canales)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del precio)

Proceso:

Deolver navigate (state, canals, función get-price, “navigate-canal-price”, null)

**Navigate-trains-price(state canals)**

Input: state(estado actual), trains(lista de caminos por trenes)

Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del precio)

Proceso:

Deolver navigate (state, canals, función get-price, “navigate-train-price”, \*forbidden\*)

\*forbidden\* = lista de ciudades prohibidas por tren

**Navigate-trains-timee(state canals)**

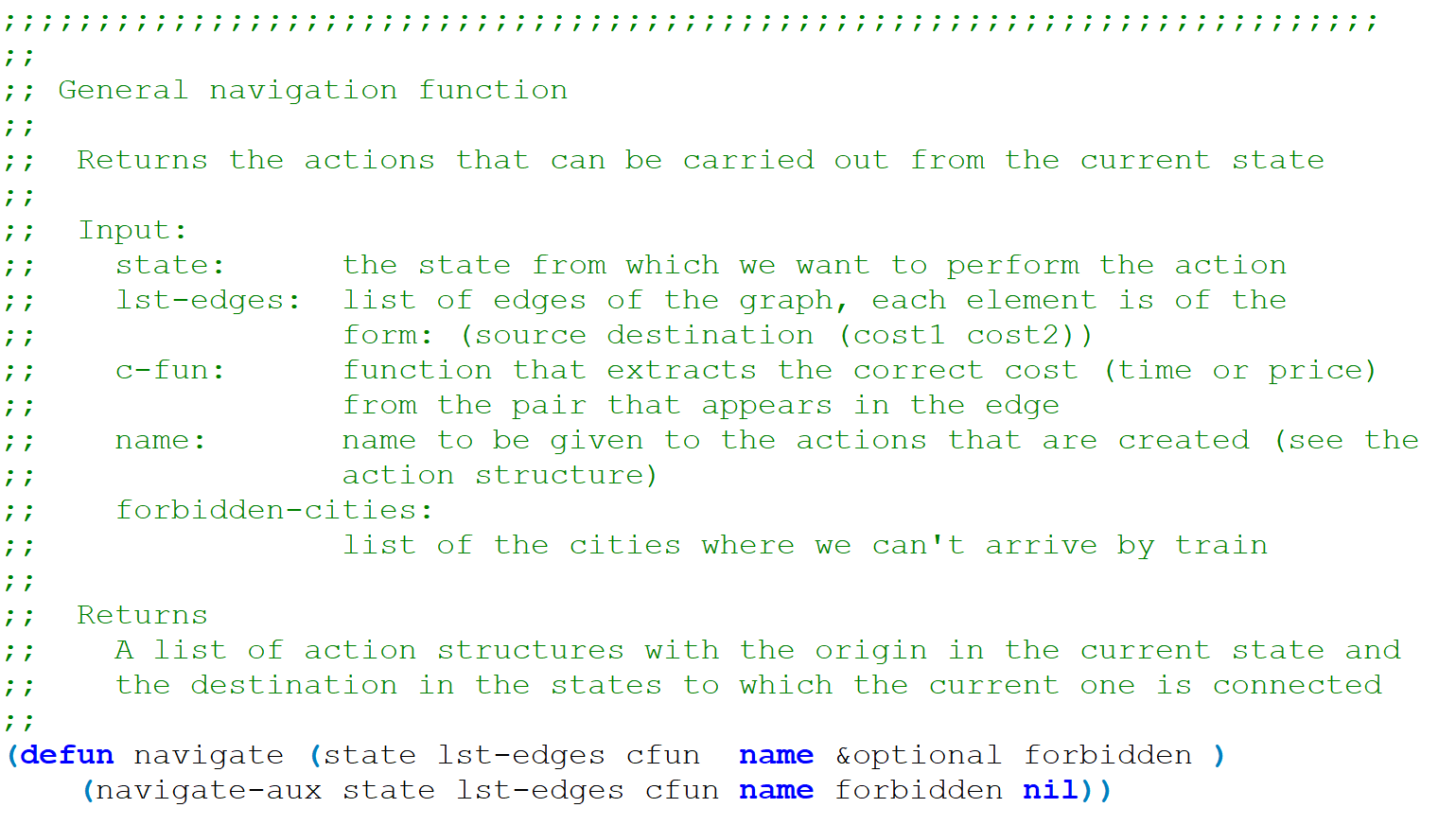
Input: state(estado actual), trains(lista de caminos por trenes)

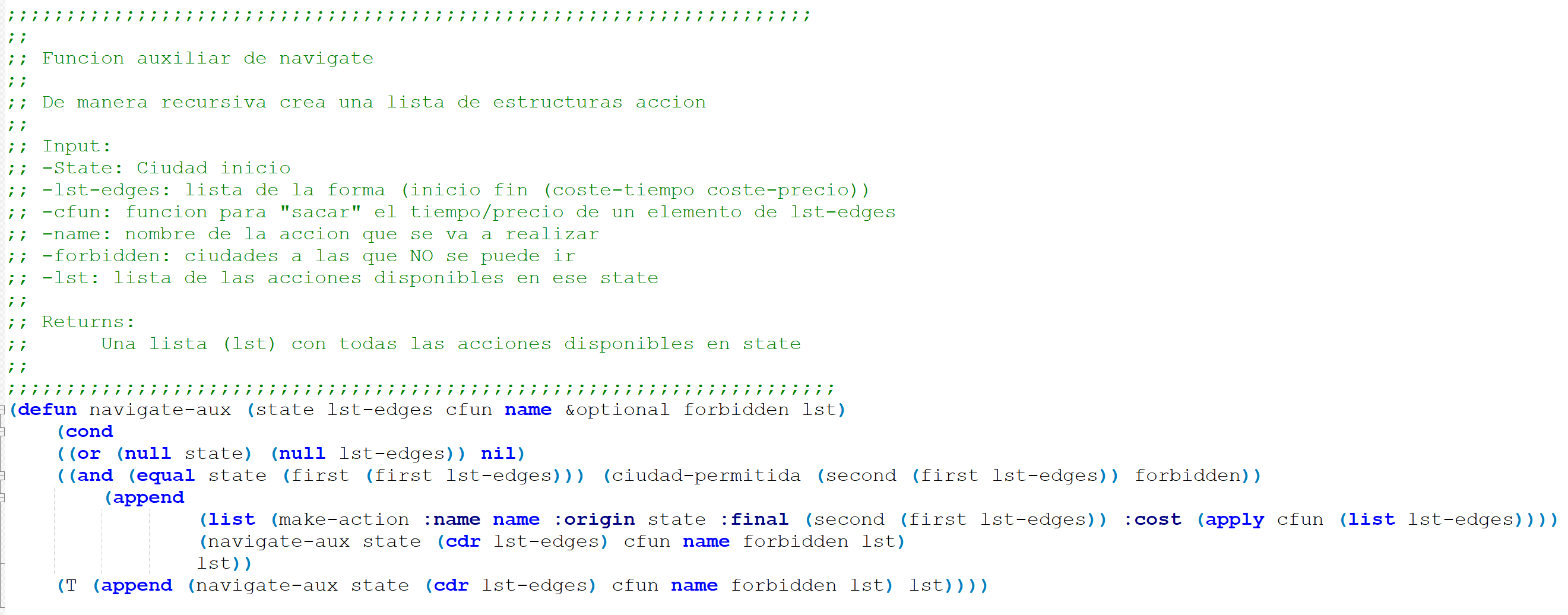
Output: Res(lista de acciones al usar canales a partir del estado actual indicando el coste del tiempo)

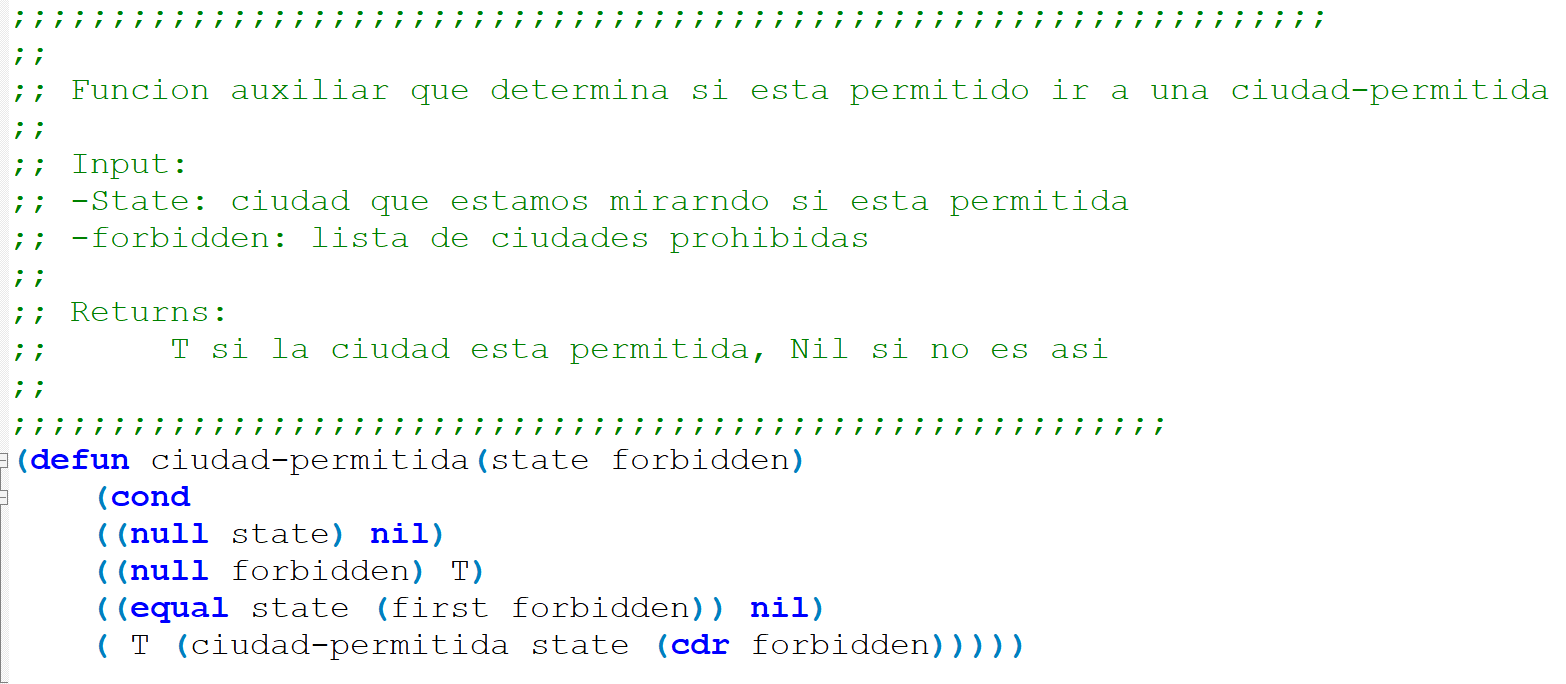
Proceso:

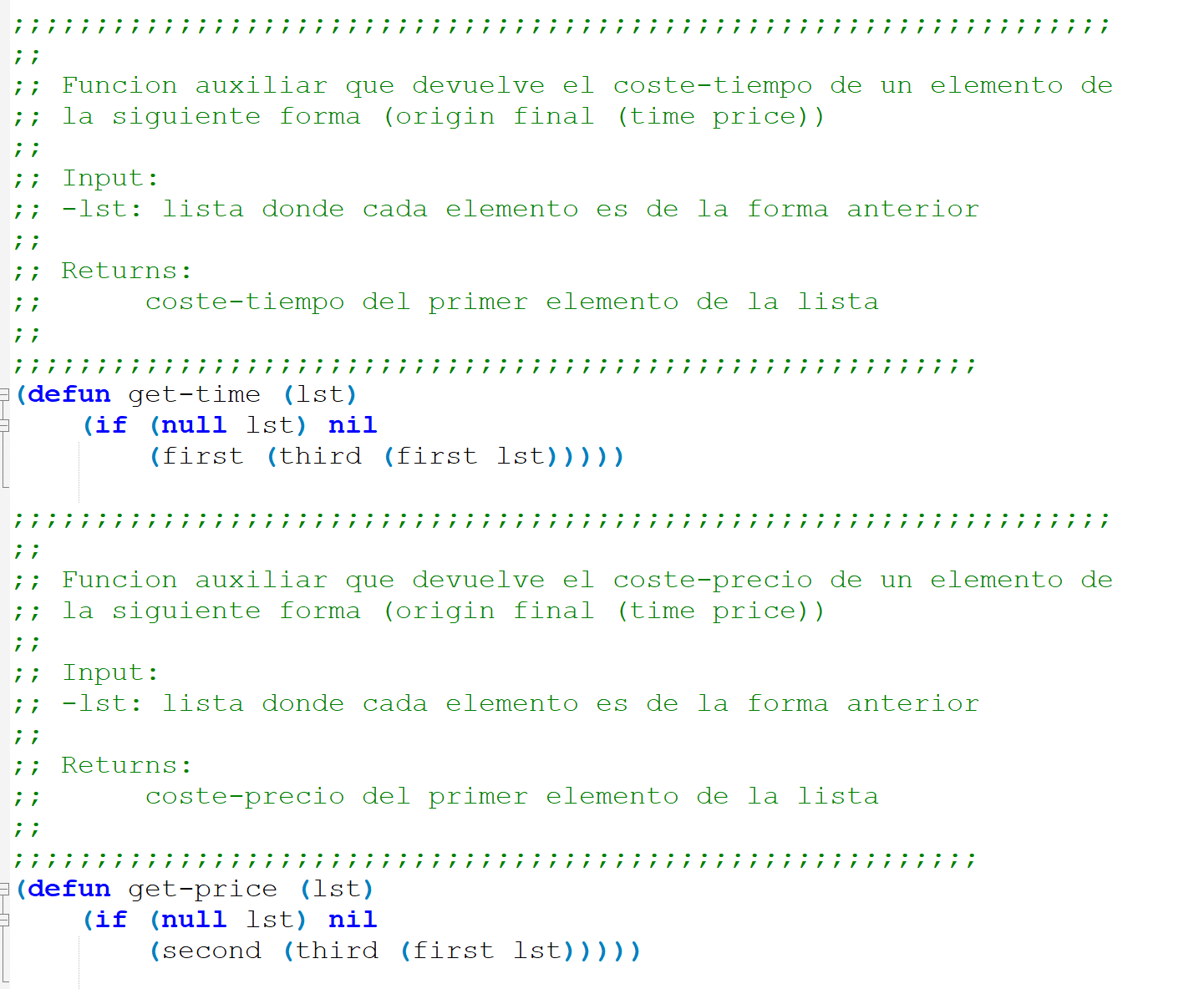
Deolver navigate (state, canals, función get-price, “navigate-train-price”, \*forbidden\*)

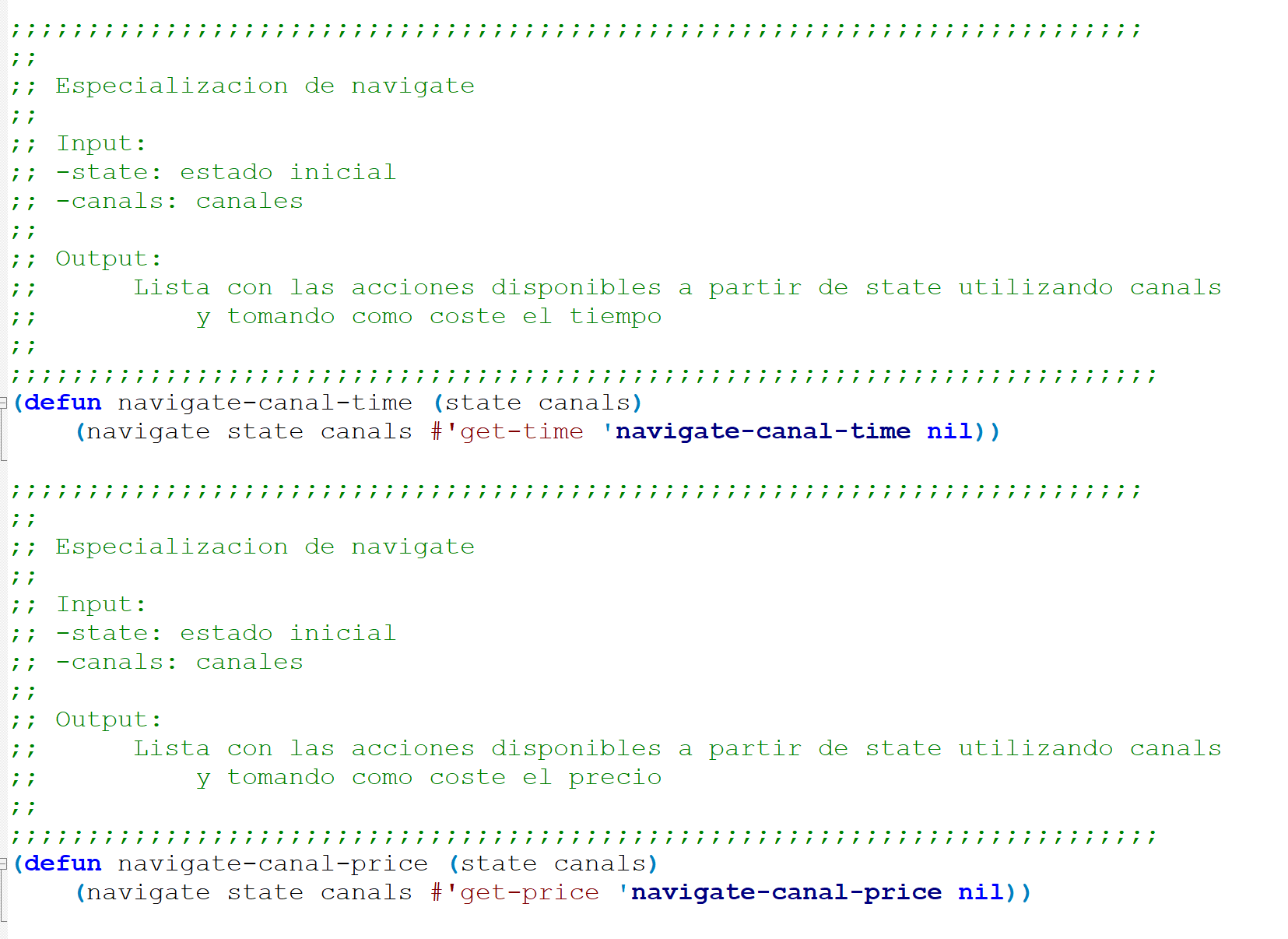
**Código del ejercicio 2:**

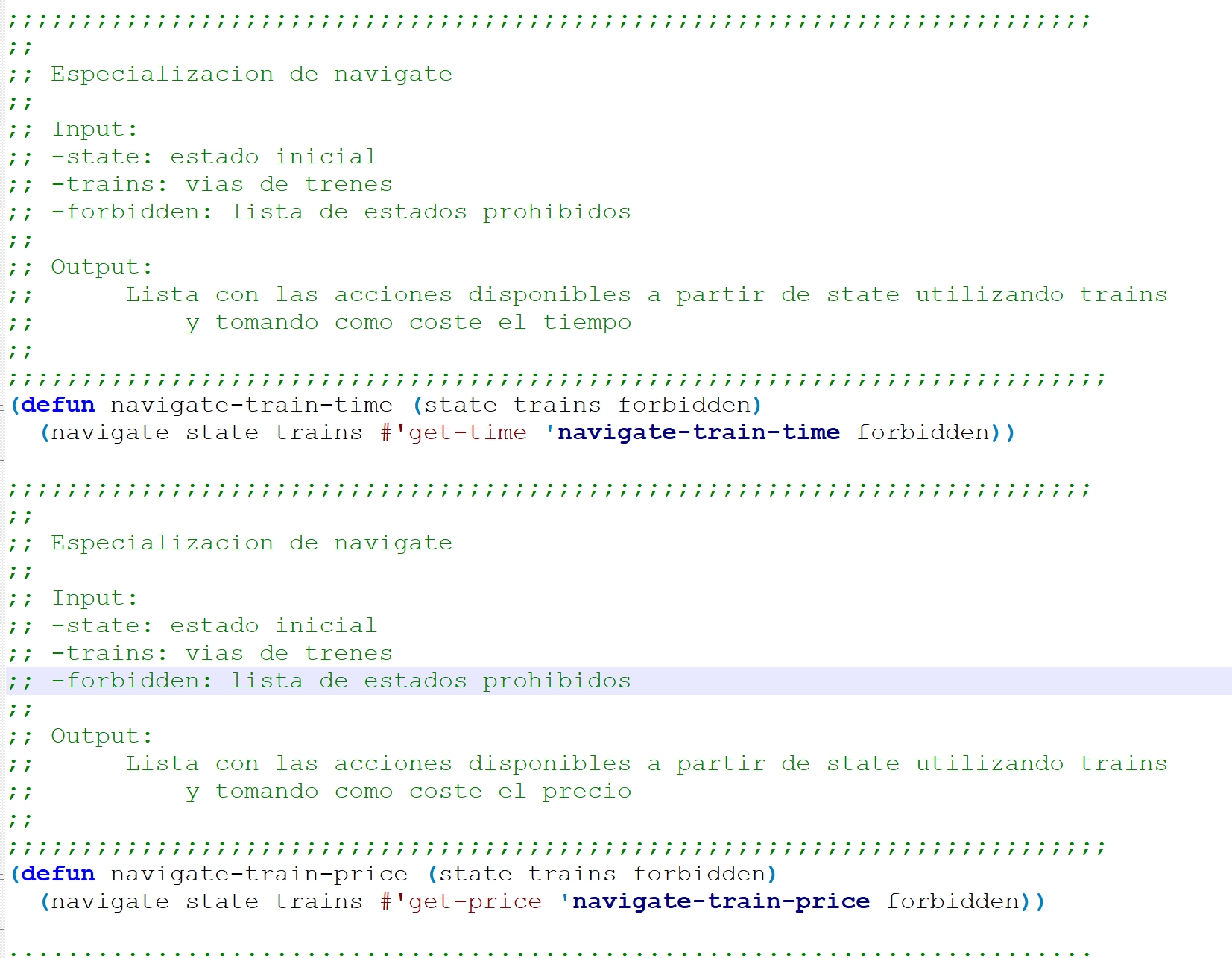












**Ejemplos de ejecución ejercicio 2:**

**CG-USER**(139): (navigate-canal-time 'Avignon \*canals\*)

(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN AVIGNON :FINAL MARSEILLE :COST 35.0))

**CG-USER**(140): (navigate-train-price 'Avignon \*trains\* '())

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0)

#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL MARSEILLE :COST 25.0))

**CG-USER**(141): (navigate-train-price 'Avignon \*trains\* '(Marseille))

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN AVIGNON :FINAL LYON :COST 40.0))

**CG-USER**(142): (navigate-canal-time 'Orleans \*canals\*)

NIL

**CG-USER**(143): (navigate-canal-time 'Lyon \*canals\*)

(#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL NANCY :COST 150.0) #S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL ROENNE :COST 40.0)

#S(ACTION :NAME NAVIGATE-CANAL-TIME :ORIGIN LYON :FINAL AVIGNON :COST 50.0))

**CG-USER**(144): (navigate-train-time 'Lyon \*trains\* \*forbidden\*)

(#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LYON :FINAL TOULOUSE :COST 60.0) #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LYON :FINAL ROENNE :COST 18.0))

**CG-USER**(145): (navigate-train-price 'Lyon \*trains\* \*forbidden\*)

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LYON :FINAL TOULOUSE :COST 95.0) #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LYON :FINAL ROENNE :COST 25.0))

**Ejercicio 3**

En este ejercicio nos piden implementar una función que compruebe si un nodo ha alcanzado el objetivo. Para ello debe corresponder a una de las ciudades objetivo y haber pasado por todas las ciudades obligatorias.

Para este ejercicio hemos implementado, a parte de la función pedida en el enunciado, la función presente que determina si un elemento está presente en una lista; otra que devuelve el camino (de estados) dado un nodo; otra que sirve para evaluar una lista de valores de verdad; y otra que determina si un camino es válido teniendo en cuenta las ciudades obligatorias, es decir, si ha pasado por todas ellas.

**Pseudocódigo de la las funciones:**

**Presente (x lst)**

Input: x(elemento que buscamos), lst (lista donde buscamos el elemento)

Output: T si el elemento está presente, Nil si no es así

Proceso:

Si x = null OR lst = null:

Devolver nil

Si no:

Equal(x, lst[0]) OR presente (x, rest(lst))

**Evaluar(lst)**

Input: lst(lista cuyos eementos son valores de verdad)

Output: T si todos son T, Nil si no

Proceso:

Si lst = True:

Devolver True

Si resto(lst) = null:

Devolver lst[0]

Si no:

List[0] AND Evaluar(lst)

**Get-camino (node lst)**

Input: node(nodo cuyo camino queremos saber), lst(lista de estados anteriores, camino)

Output: lst (lista de estados del camino)

Proceso:

When node node >< node-nevers

Parent = node-get-parent(node)

Añadir-a-lst (get-state(node))

Añadir-a-lst (get-camino (parent, lst))

**Camino-valido (camino mandatory)**

Input: camino(lista de estados), mandatory(lista de estados, las ciudades obligatorias)

Output: T si ha camino pasa por todas, Nil si no es así

Proceso:

Para i <= longitud(mandatory):

Lista

Añadir-a-lista(presente (mandatory[i]), camino)

Devolver evaluar(lista)

**f-goal-test (node destination mandatory)**

Input: node(nodo correspondiente al estado actual), destination (ciudades destino), mandatory(ciudades obligatorias)

Output: T si el estado es una ciudad destino y ha pasado por todas las obligatorias

Proceso:

Estado = node-estado(node)

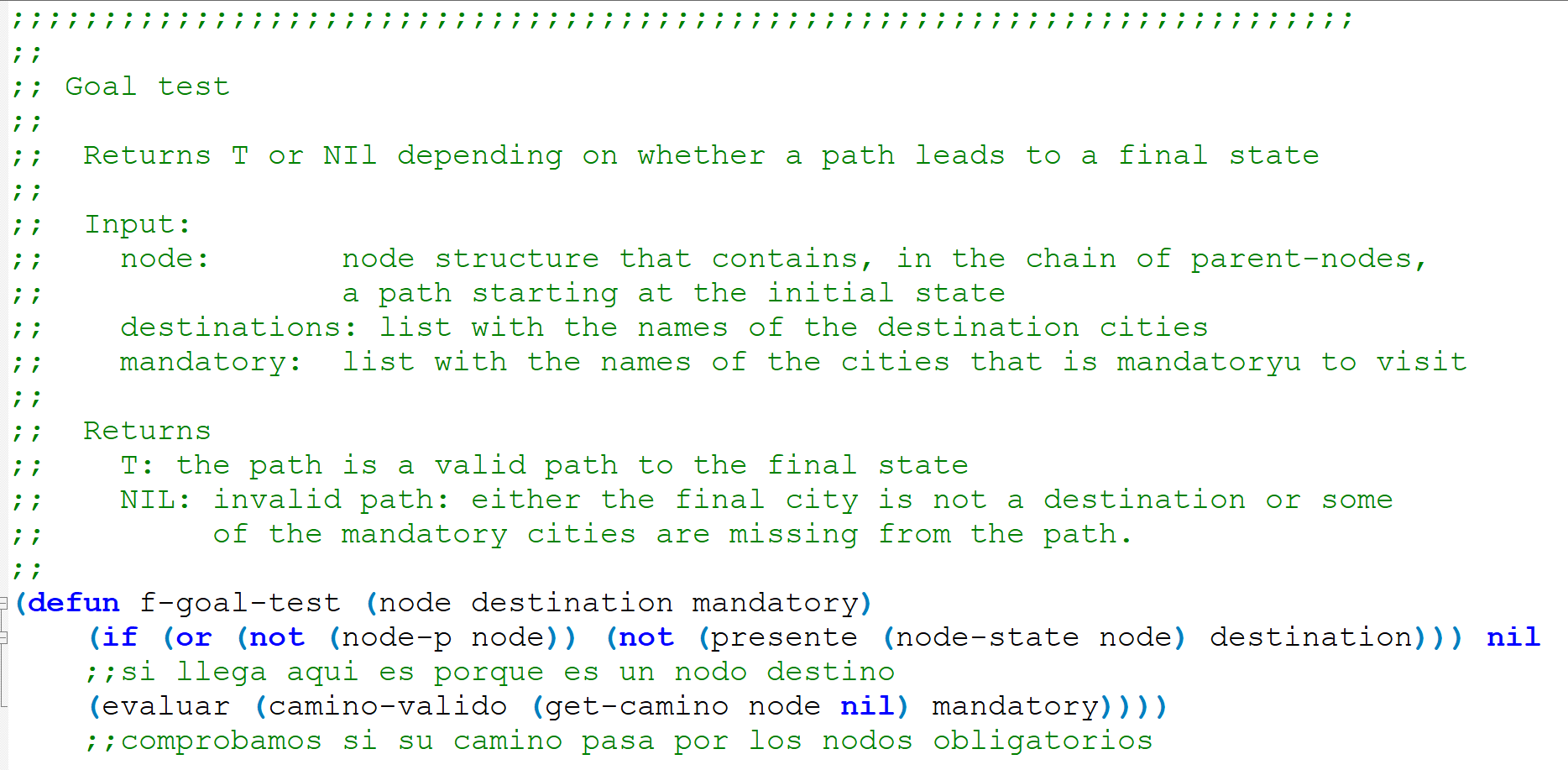
Si (nil = presente(estado, destination):

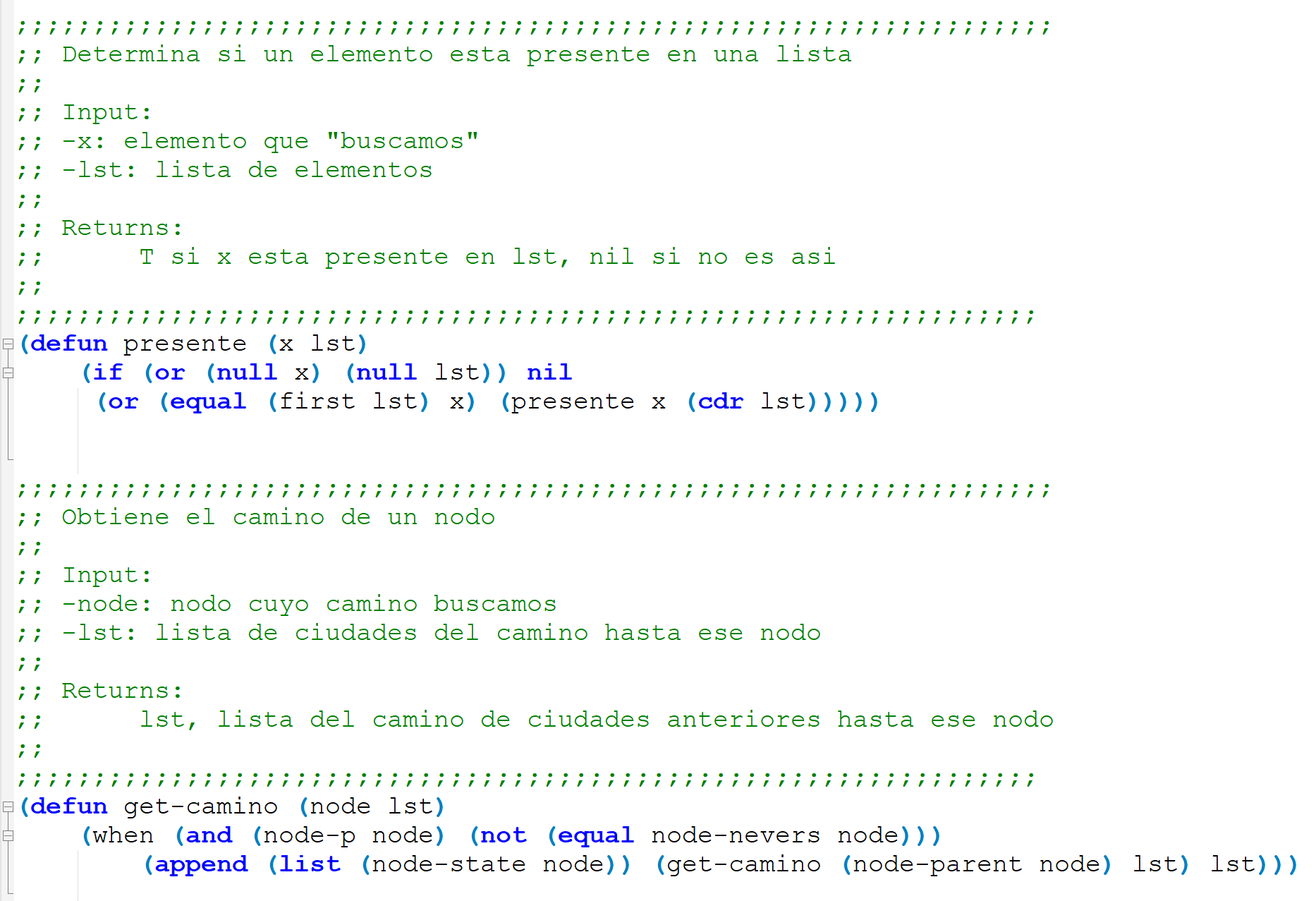
Devovler nil

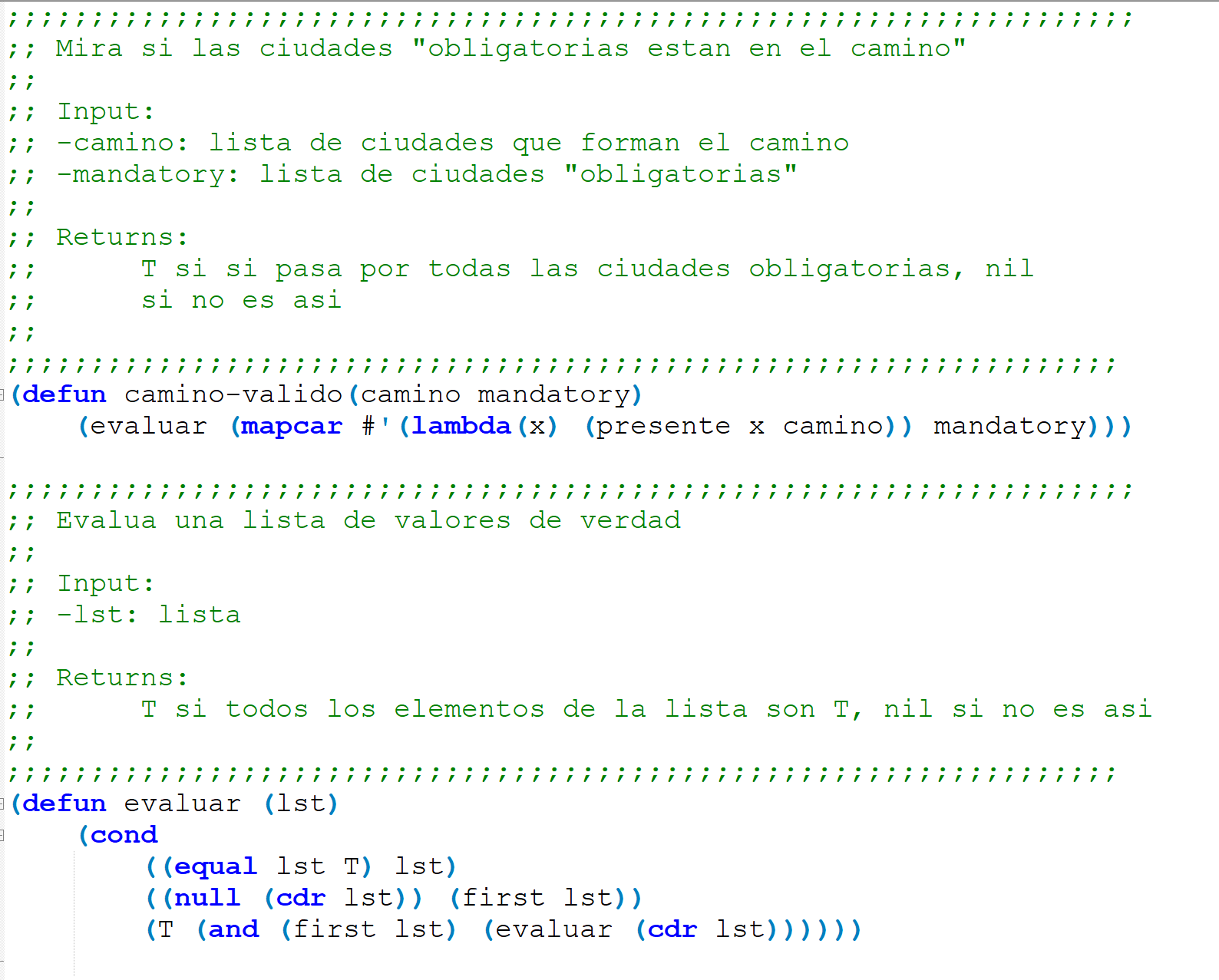
Si no:

Evaluar (camino-valido (get-camino node, nil), mandatory)

**Código implementado:**







**CG-USER**(104): (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))

T

**CG-USER**(105): (f-goal-test node-calais '(Calais Marseille) '(Paris Limoges))

NIL

**CG-USER**(106): (f-goal-test node-paris '(Calais Marseille) '(Paris))

NIL

**CG-USER**(107): (f-goal-test node-calais '(Calais Marseille) '(Paris Nancy))

T

**CG-USER**(108): (f-goal-test nil '(hola hola) '(ninguno))

NIL

\*Como decision hemos tomado que node-nevers no puede ser un nodo de estado final, pues representa que no ha habido ningún camino

**CG-USER**(109): (f-goal-test node-nevers nil nil)

NIL

\*\*function auxiliar

**CG-USER**(110): (get-camino node-calais nil)

(CALAIS REIMS NANCY PARIS)

**CG-USER**(111): (get-camino node-nevers nil)

NIL

**CG-USER**(116): (get-camino node-paris '(madrid barcelona))

(PARIS MADRID BARCELONA)5

**Ejercicio4**

En este ejercicio nos piden crear una función para ver si dos estados de búsqueda son iguales (misma ciudad y mismas ciudades obligatorias visitadas).

Hemos creado dos funciones auxiliares, una que devuelve las ciudades obligatorias no visitadas y otra que devuelve si dos listas son “equivalentes”, es decir, mismos elementos aunque no tienen por que estar en el mismo orden.

**Pseudocódigo de las funciones:**

**Equivalentes (lst1 lst2)**

Input: lst1(lista), lst2(lista)

Output: T si son equivalentes y nil si no es así

Proceso:

Lista, lista2

Para cada i <= longitud(lst1):

Lista[i] = presente(lst1[i], lst2)

Para cada I <= longitude(lista2):

Lista2[i] = presente(lst2[i], lst1)

Devolver evaluar (lista AND lista2)

**No-visitados(camino mandatory lst)**

Input: camino (lista ciudades del camino), mandatory(lista ciudades obligatorias), lst(lista ciudades obligatorias visitadas)

Output: lst(lista de ciudades obligatorias no visitadas)

Proceso:

Si presente(mandatory[0], camino) = nil:

Añadir a lst (mandatory[0])

Devolver no-visitados(camino rest(mandatory) lst)

Si no:

Devolver no-visitados(camino rest(mandatory) lst)

**f-search-state-equal(node-1 node-2 mandatory(opcional))**

Input: node-1(nodo), node-2(nodo), mandatory(ciudades obligatorias)

Output: T si son el mismo estado nil si no

Proceso:

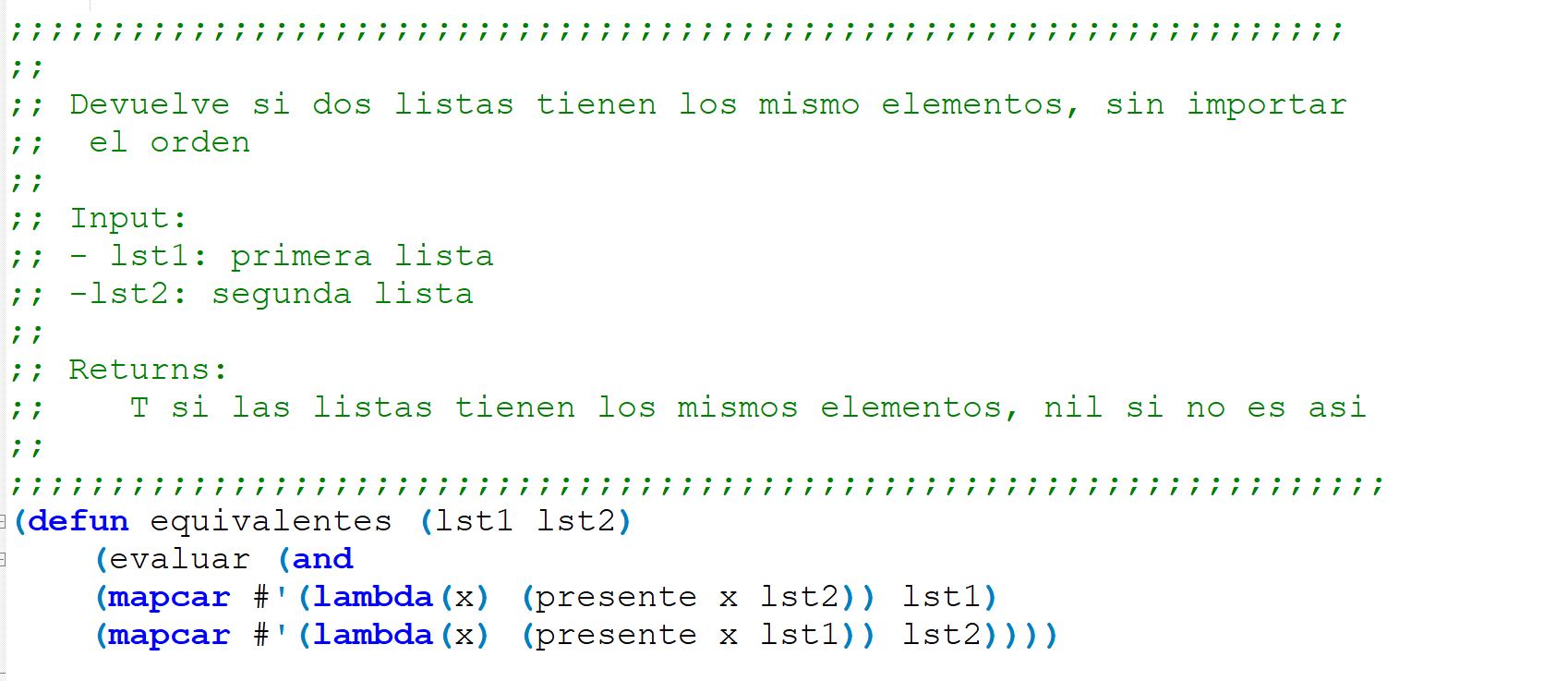
Si mandatory = null:

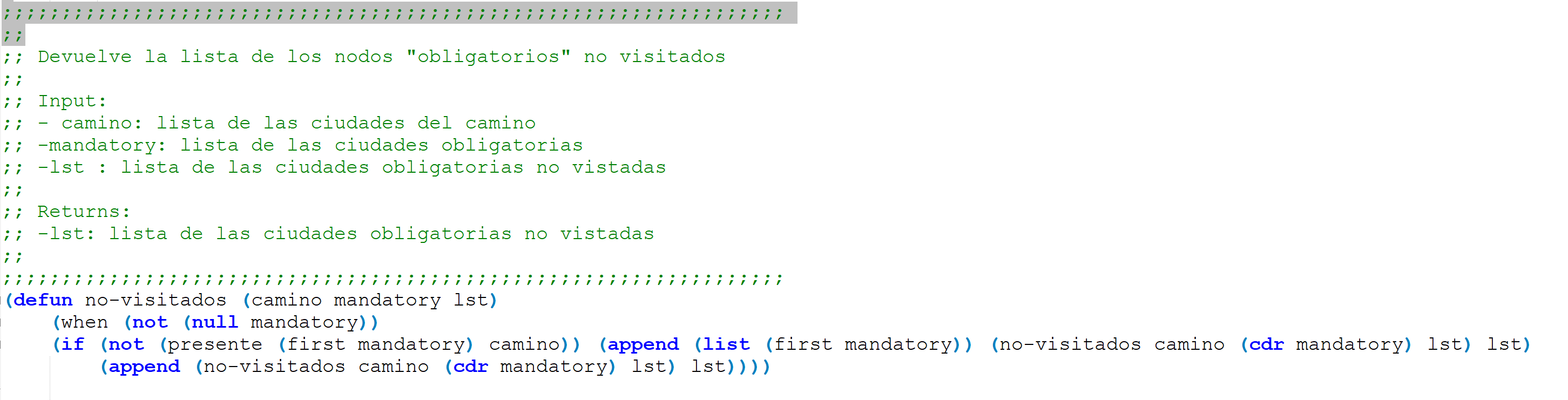
Devolver node-1 == node-2

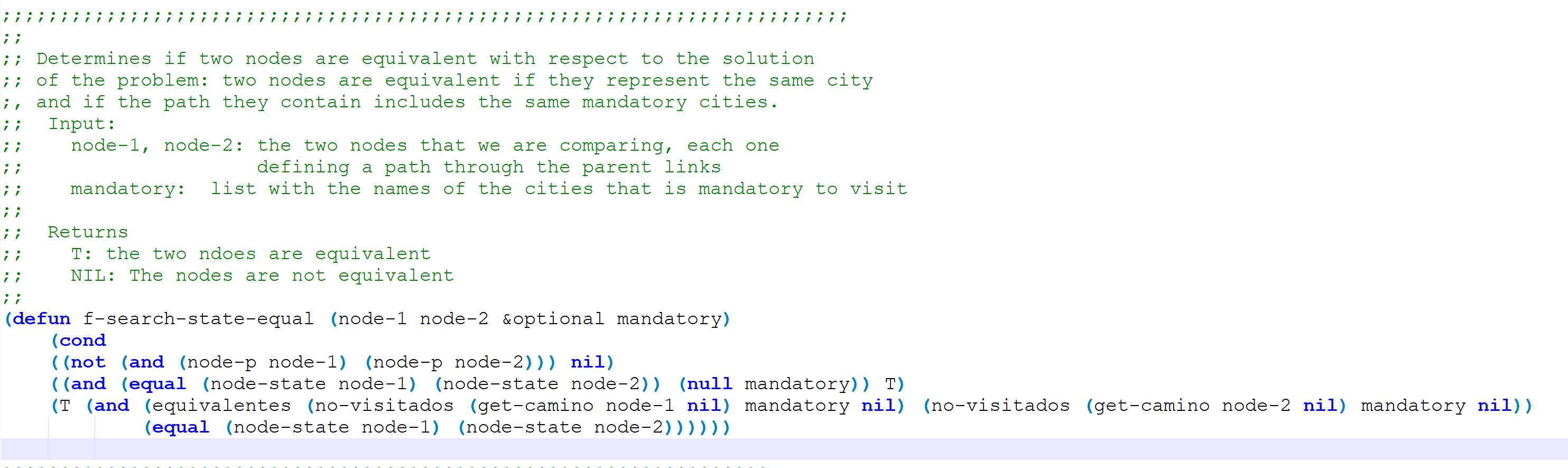
Si no:

Devolver (node-1 == node-2) AND equivalentes( no-visitados(node-1, nil mandatory, nil), no-visitados(node-2, nil, mandatory,nil))

**Código**







**Ejemplos de ejecución ejercicio 4:**

**CG-USER**(144): (f-search-state-equal node-calais node-calais-2 '())

T

**CG-USER**(145): (f-search-state-equal node-calais node-calais-2 '(Reims))

NIL

**CG-USER**(146): (f-search-state-equal node-calais node-calais-2 '(Nevers))

T

**CG-USER**(147): (f-search-state-equal node-nancy node-paris '())

NIL

\*\*Ejemplos de la funcion auxilair

**CG-USER**(148): (no-visitados nil '(madrid) nil)

(MADRID)

**CG-USER**(149): (no-visitados '(madrid barcelona) '(madrid paris lyon) nil)

(PARIS LYON)

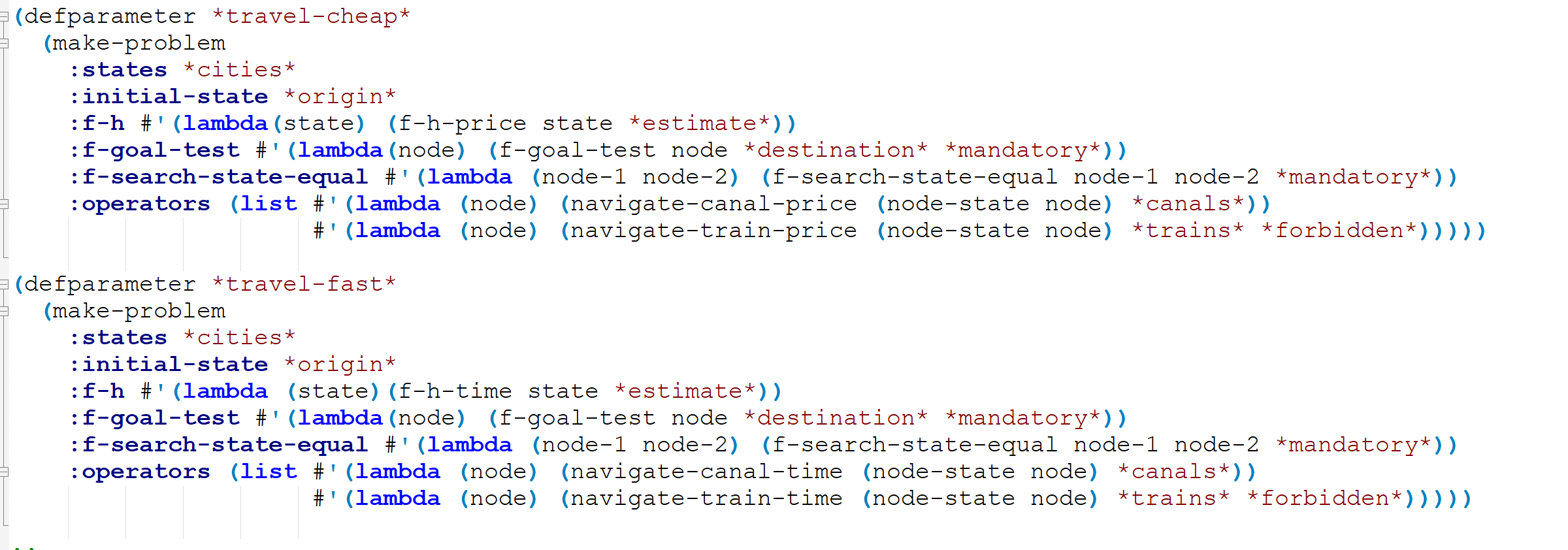
**CG-USER**(150): (no-visitados '(madrid) nil nil)

NIL

**Ejercicio5**

En este ejercicio solo tenemos que definir dos estructuras de problemas, una para cuando se tenga en cuenta el coste de precio y otra para cuando se tenga en cuenta el coste del tiempo.

**Código:**



**Ejercicio 6**

En este ejercicio nos piden que implementemos una función que expanda un nodo, como funciones auxiliares hemos hecho “eliminar-nil” que elimina los nil de una lista y “flatten” que elimina los paréntesis entre elementos de una lista; ésta última la hemos sacado de los apuntes proporcionados en Moodle.

Para expandir un nodo, primero sacamos las acciones que se pueden realizar en ese nodo, utilizando los operadores de la estructura problema. Y después para cada una de las acciones creamos un nodo con los atributos correspondientes.

**Pseudocódigo:**

**Flatten(lst)**

Input:lst (lista)

Output: lst (lista sin paréntesis entre sus elementos)

Proceso:

Si lst == null:

Devovler nil

Si lst[0] es un átomo:

Devolver Añadir lst[0] a flatten(resto(lst))

Si no:

Añadir flatten(lst[0]) a flatten(rest(lst))

**Eliminar-nil (lst)**

Input:lst(lista)

Output: lst (lista sin nil)

Proceso:

Cuando lst >< null:

Si lst[0] == null:

Devovler eliminar-nil(resto(lst))

Si no:

Añadir lst[0] a eliminar-nil (resto(lst))

**Expand-node-operator (node cfun)**

Input: node(nodo), cfun(función operador)

Output: Res (lista de estados finales al aplicar ese operador a ese nodo)

Proceso:

Lista-acciones = cfun(node)

Res = null

Para cada i <= longitud( lista-acciones ):

Añadir a Res estado-final-de(lista-acciones[i])

Devolver Res

**Expand-node-action(parent action problem)**

Input: parent(nodo padre), action (accion), problem(problema)

Output: Node resultante de aplicar esa acción a ese nodo en ese problema

Proceso:

Nodo node

Node.estado = estado-final-de(action)

Node.parent = parent

Node.depth = 1 + Depth-de(parent)

Node.g = g-de(parent) + coste-accion(action)

Node.h= aplicar función problema-f-h(problema) a estado-final(action)

Node.f = node.h + node.g

Devolver node

**Expand-node(node parent)**

Input: node(nodo a expandir), problem

Output: Res(una lista de nodos resultantes de explorar node)

Proceso:

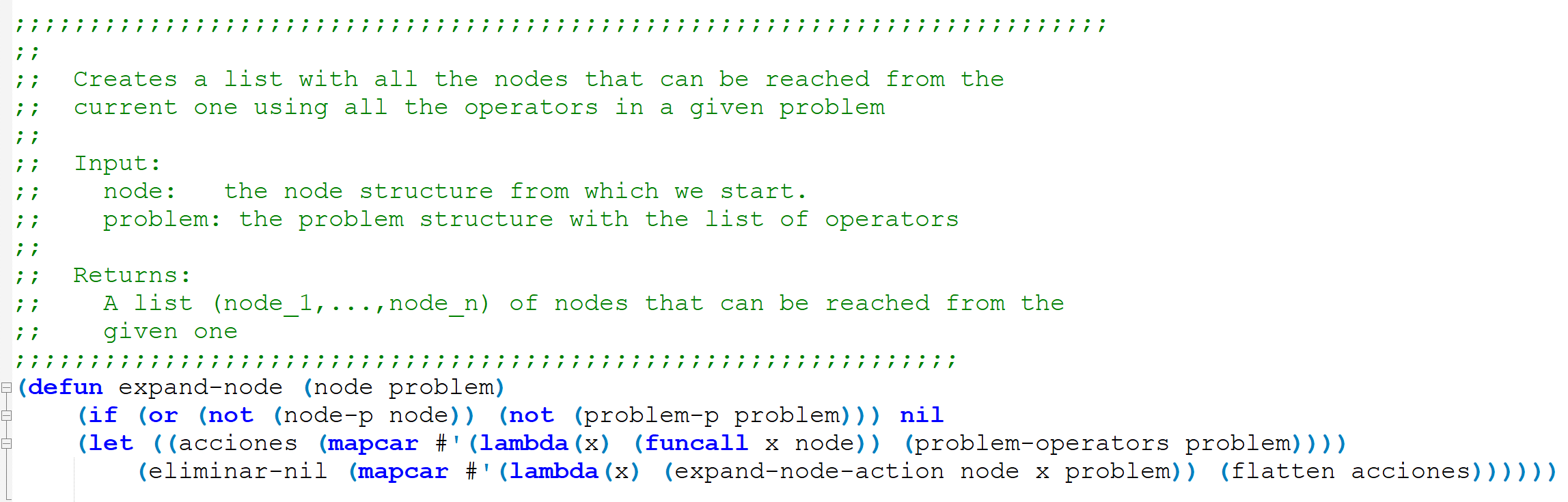
Acciones = aplicar-operadores-deproblem(problem) a node

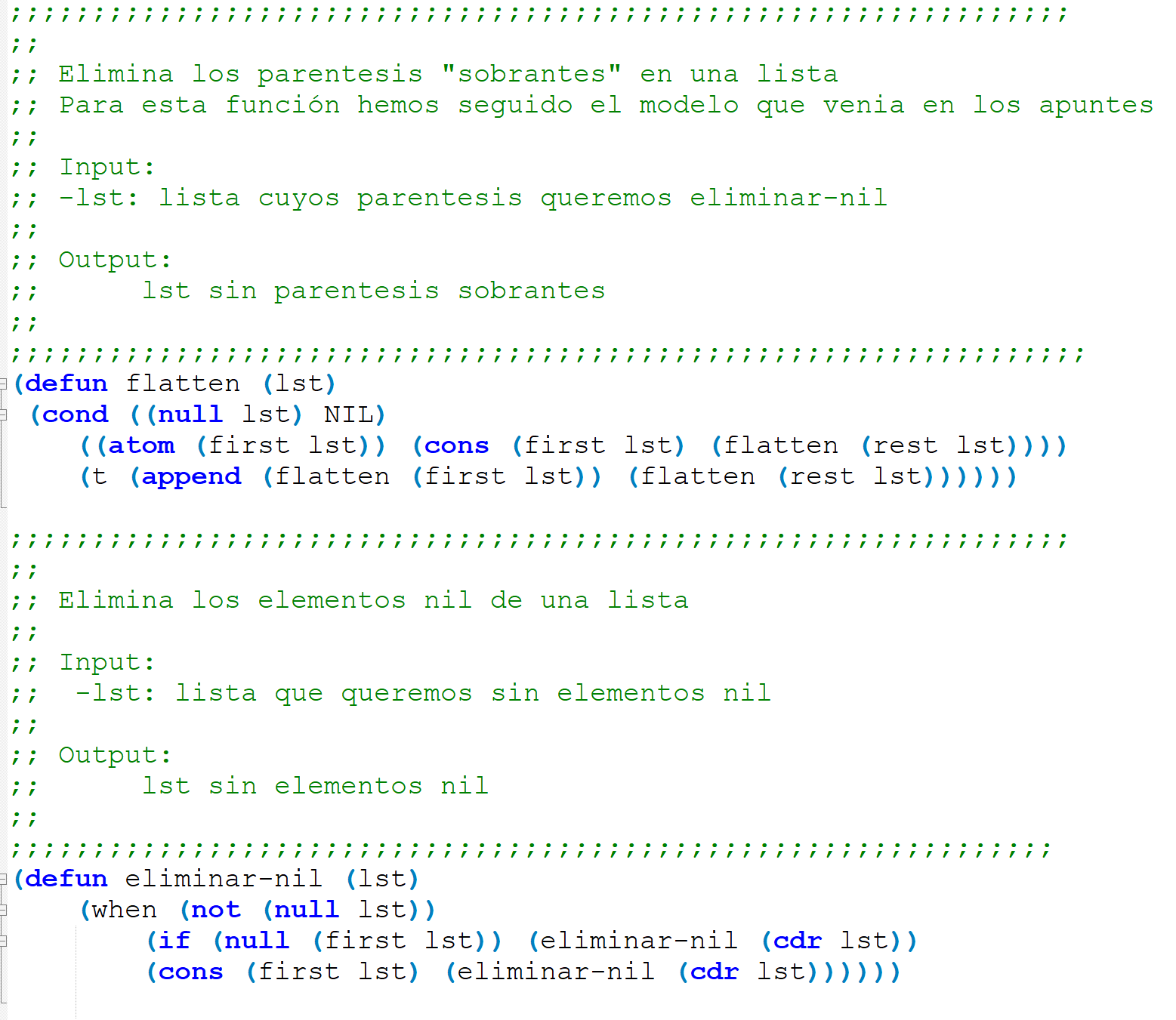
Para i <= longitud(acciones)

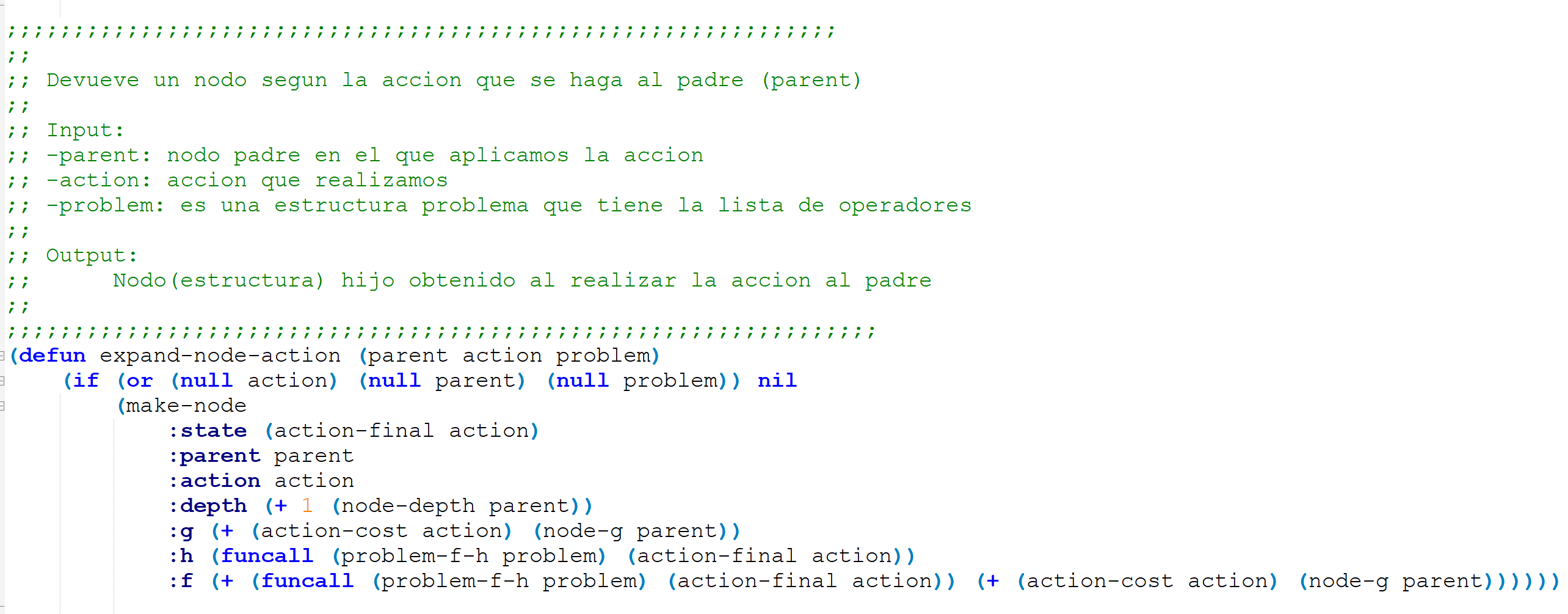
Añadir a Res = eliminar-nil (expand-node-action(acciones[i], node, problem)

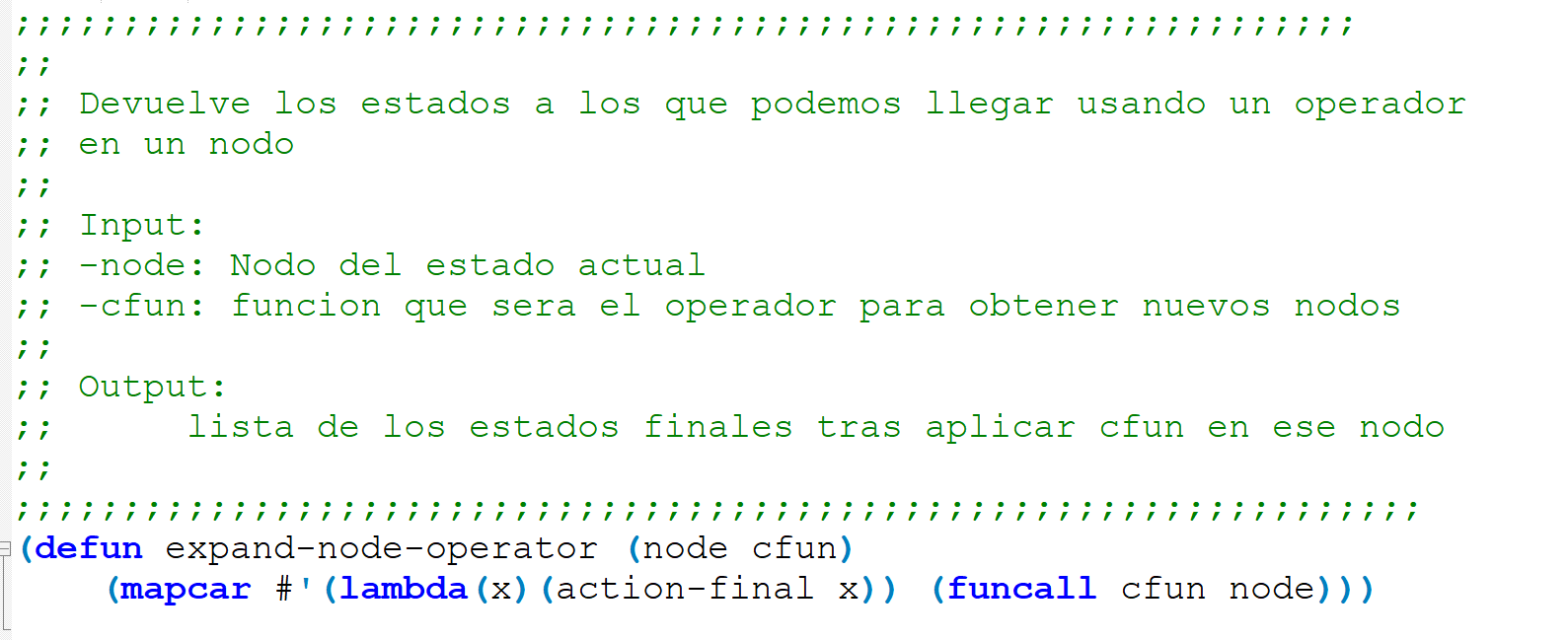
Devolver Res

**Código de las funciones**









**Ejemplos de ejecución ejercicio 6**

\*\*Ejemplo de la función auxiliar expand-node-action

(Con este ejemplo comprobamos que esta función funcione bien)

Hemos ido pidiendo cada uno de los atributos de nodo porque en la terminal no salía el nodo al completo

**CG-USER**(95): (node-f (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

205.0

**CG-USER**(96): (node-g (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

75.0

**CG-USER**(97): (node-h (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

130.0

**CG-USER**(98): (node-depth (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

13

**CG-USER**(99): (node-action (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0)

**CG-USER**(100): (node-parent (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

#S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G ...)

**CG-USER**(101): (node-state (expand-node-action node-marseille-ex6 #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0) \*travel-fast\*))

TOULOUSE

Ejemplos de ejecución de la función expand-node:

**\*ejemplo del enunciado**

CL-USER(53): (defparameter lst-nodes-ex6

(expand-node node-marseille-ex6 \*travel-fast\*))

LST-NODES-EX6

CL-USER(54): (print lst-nodes-ex6)

(#S(NODE :STATE TOULOUSE

:PARENT #S(NODE :STATE MARSEILLE

:PARENT NIL

:ACTION NIL

:DEPTH 12

:G 10

:H 0

:F 20)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME

:ORIGIN MARSEILLE

:FINAL TOULOUSE

:COST 65.0)

:DEPTH 13

:G 75.0

:H 130.0

:F 205.0))

(#S(NODE :STATE TOULOUSE

:PARENT #S(NODE :STATE MARSEILLE

:PARENT NIL

:ACTION NIL

:DEPTH 12

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME

:ORIGIN MARSEILLE

:FINAL TOULOUSE

:COST 65.0)

:DEPTH 13

:G ...))

**\*introduciendo un problema no valido**

**CL-USER**(77): (expand-node node-paris nil)

NIL

**\*introduciendo un nodo no valido**

CL-USER(78): (expand-node nil \*travel-cheap\*)

NIL

**\*Paris con la heuristica sobre el precio.**

**Hemos elegido parís porque tiene muchas conexiones con otras ciudades.**

CL-USER(79): (expand-node node-paris \*travel-cheap\*)

(#S(NODE :STATE REIMS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE

:ORIGIN PARIS

:FINAL REIMS

:COST 10.0)

:DEPTH 1

:G ...)

#S(NODE :STATE NANCY

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE

:ORIGIN PARIS

:FINAL NANCY

:COST 10.0)

:DEPTH 1

:G ...)

#S(NODE :STATE CALAIS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE

:ORIGIN PARIS

:FINAL CALAIS

:COST 60.0)

:DEPTH 1

:G ...)

#S(NODE :STATE NANCY

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE

:ORIGIN PARIS

:FINAL NANCY

:COST 67.0)

:DEPTH 1

:G ...)

#S(NODE :STATE NEVERS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE

:ORIGIN PARIS

:FINAL NEVERS

:COST 75.0)

:DEPTH 1

:G ...)

#S(NODE :STATE ORLEANS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE

:ORIGIN PARIS

:FINAL ORLEANS

:COST 38.0)

:DEPTH 1

:G ...)

#S(NODE :STATE ST-MALO

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE

:STATE

NEVERS

:PARENT

NIL

:ACTION

NIL

:DEPTH

0

:G

...)

:ACTION NIL

:DEPTH 0

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE

:ORIGIN PARIS

:FINAL ST-MALO

:COST 70.0)

:DEPTH 1

:G ...))

**Ejercicio7**

En este ejercicio nos piden implementar un función que inserte una lista de nodos en otra(ya ordenada) de manera que quede la lista ordenada en función de un estrategia, pasada como parámetro de entrada.

En este ejercicio nos proporcionan el modelo de dos funciones que necesitaremos crear, aún así, nosotros hemos creado una tercera que inserta un único nodo en una lista ya ordenanda.

**Pseudocódigo de la funciones:**

**Insert-node (node lst-nodes node-compare-p)**

Input: node(nodo), lst-nodes(lista de nodos ya ordenada), node-compare-p (función que compara dos nodos)

Output: lst-nodes (lista de nodos ordenada)

Proceso:

Si lst-nodes = null:

Devolver Lst-nodes

// Como esto es una función interna si pasa eso habría habido un error

// al llamar a esa función

Si no es un nodo node:

Devolver nil

si node-compare-p (node, lst-nodes[i]) = True:

añadir-delante(node, lst-nodes)

si no:

añadir delante(lst-nodes[0], insert-node(node, rest(lst-nodes)))

**insert-nodes (nodes lst-nodes node-compare-p)**

Input: nodes(lista de nodos), lst-nodes(lista de nodos ya ordenada), node-compare-p (función que compara dos nodos)

Output: lista ya ordenada con todos los nodos de ambas listas

Proceso:

If nodes = null:

Devolver lst-nodes

Si no:

Lst-nodes-2 = insert-node(nodes[i], lst-nodes, node-compare-p)

Devolver insert-nodes (rest(nodes), lst-nodes-2, node-compare-p)

**Insert-nodes-strategy(nodes lst-nodes strategy)**

Input: nodes(lista de nodos), lst-nodes(lista de nodos ya ordenada), strategy(estrategia para saber como ordenar los nodos)

Output: lista con todos los nodos ya ordenada

Proceso:

Si nodes = null

Devovler lst-nodes

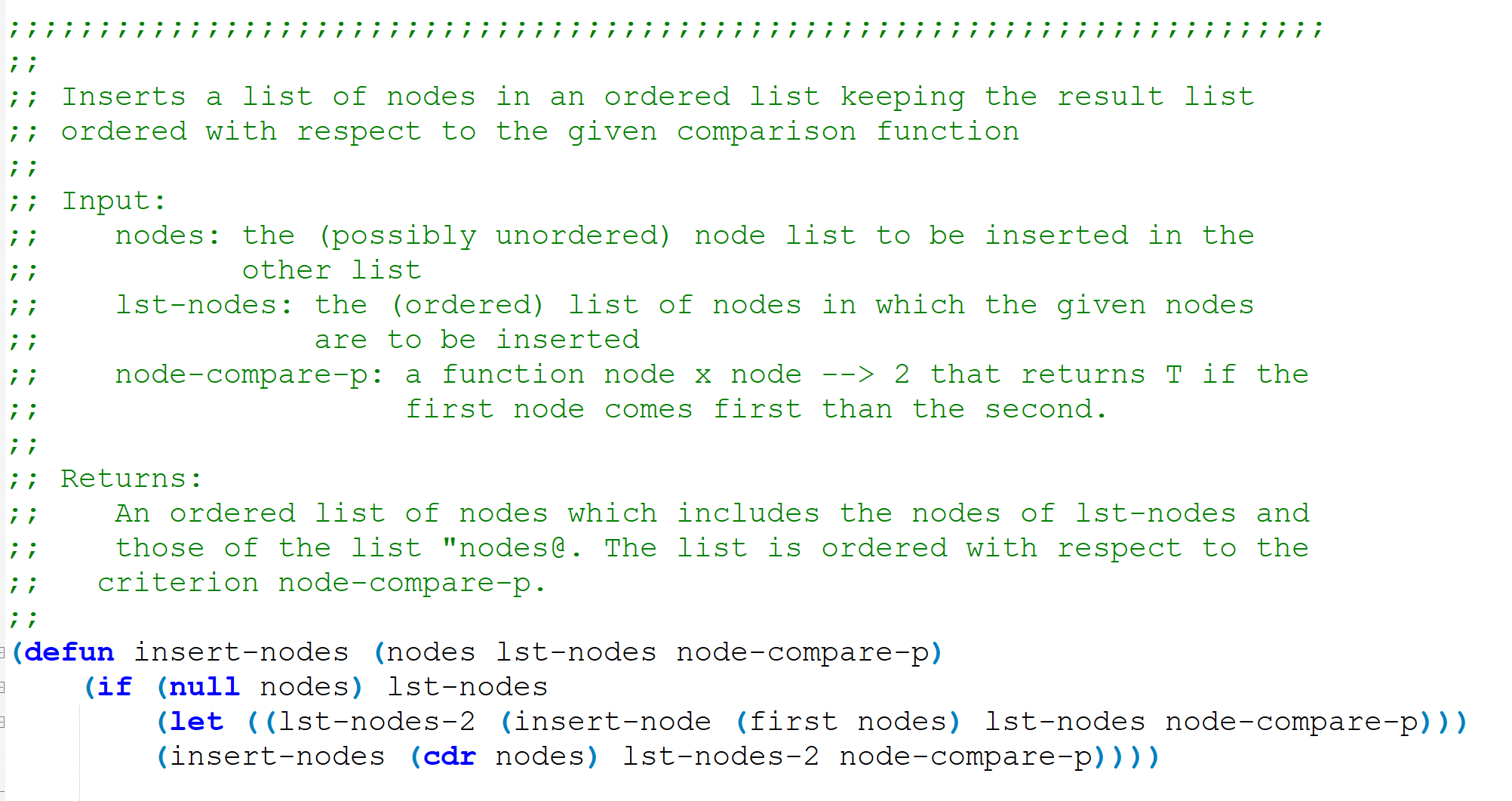
Si lst-nodes = null

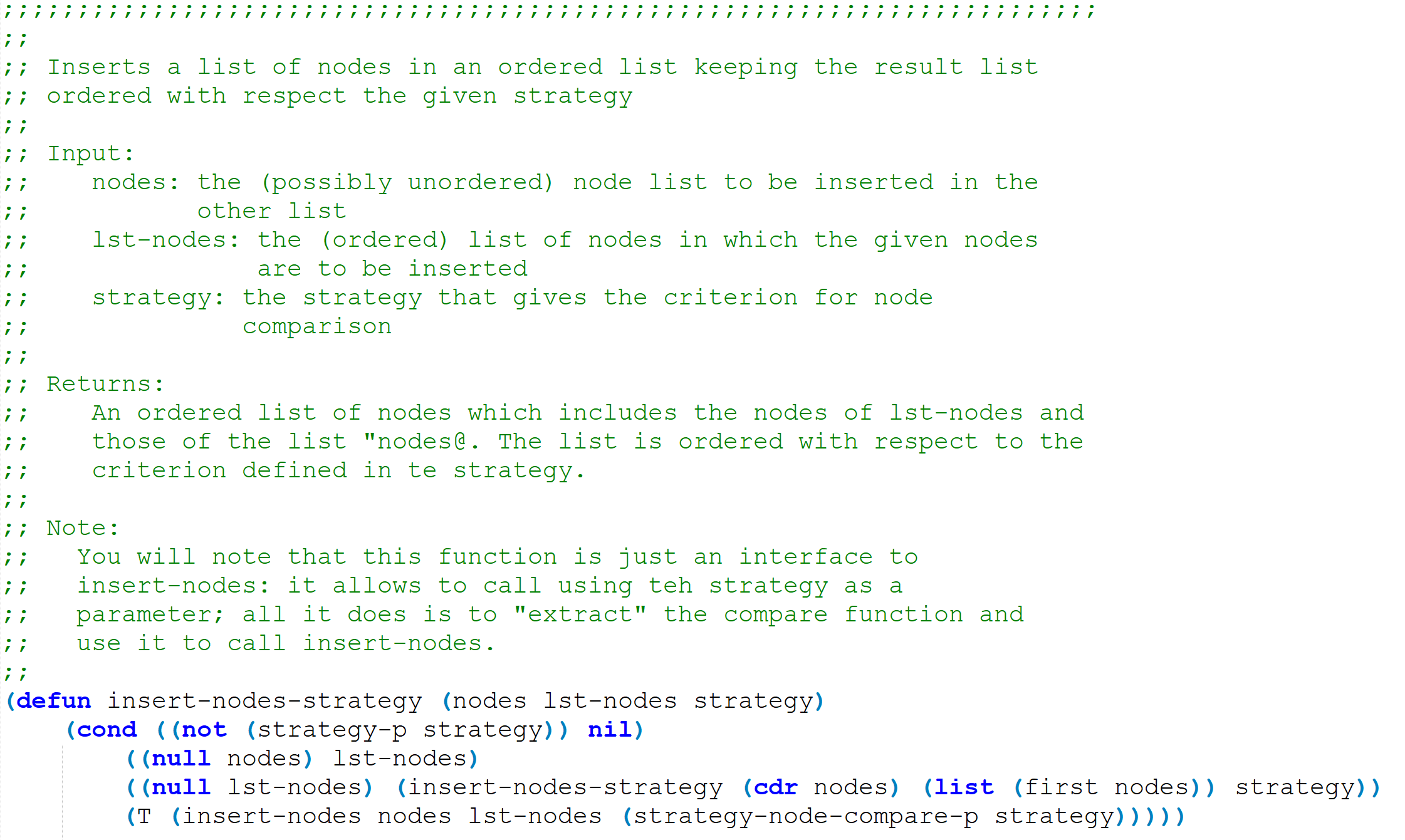
Devolver insert-nodes-strategy(rest(nodes), lista-de-nodes[0], strategy)

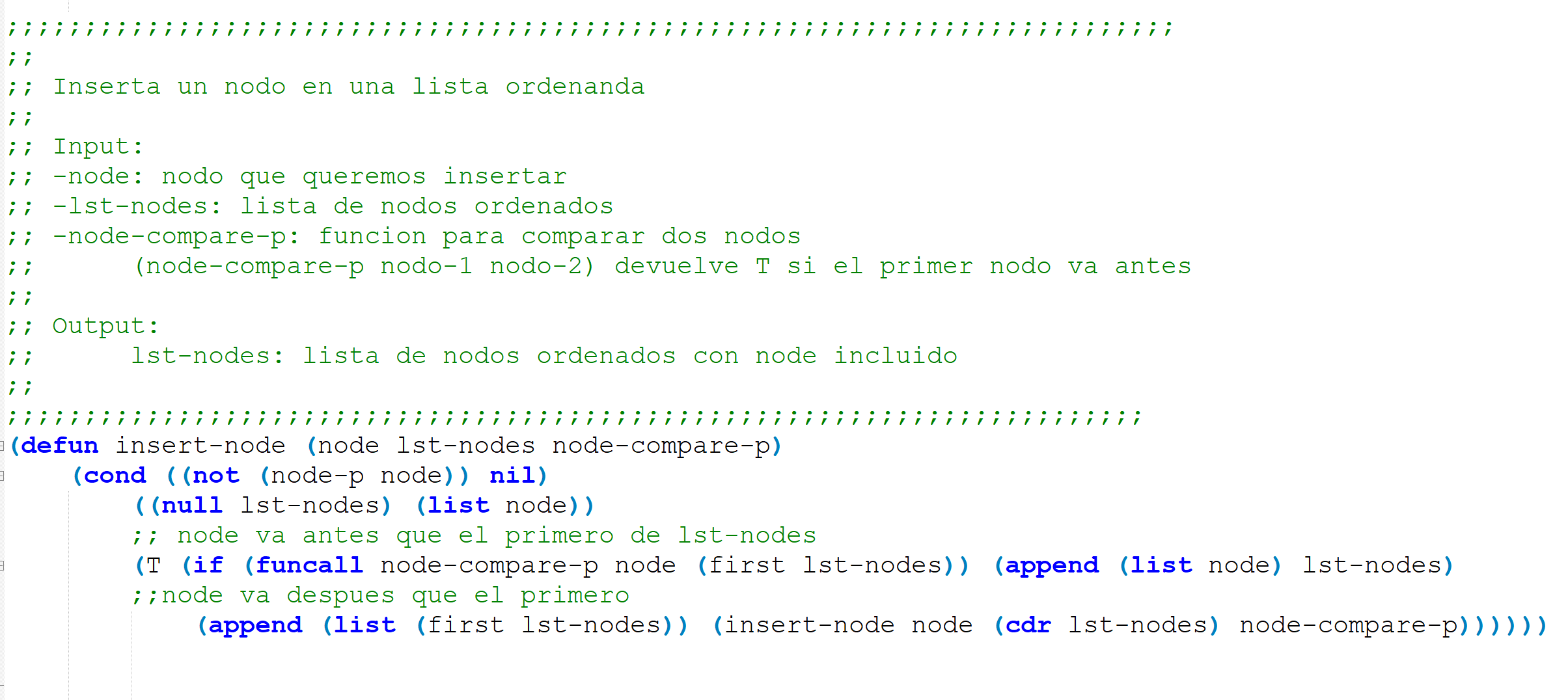
Si no:

Devolver insert-nodes(nodes, lst-nodes, funcion-comparar-de(strategy))

**Código de las funciones:**







**Ejemplos de ejecución ejercicio7:**

\*\*función auxiliar insert-node:

**CG-USER**(91): (defparameter aux (insert-node node-paris-ex7 (list node-nancy-ex7) (strategy-node-compare-p \*uniform-cost\*)))

AUX

**CG-USER**(92): (mapcar #'(lambda (x) (node-state x)) aux)

(PARIS NANCY)

(a esta función se le llama desde otras el control de errores de los otros dos parámetros ya se habría realizado previamente)

**CG-USER**(93): (insert-node nil (list node-nancy-ex7) (strategy-node-compare-p \*uniform-cost\*))

NIL

Insert-nodes-strategy básicamente llama a insert-nodes, también realiza los primeros controles de errores, pasándole la función comparación, por lo que directamente vamos a mostrar los resultados de ejecutar esta función:

**CG-USER**(97): sol-ex7

(#S(NODE :STATE PARIS :PARENT NIL :ACTION NIL :DEPTH 0 :G ...) #S(NODE :STATE NANCY :PARENT NIL :ACTION NIL :DEPTH 2 :G ...)

#S(NODE :STATE TOULOUSE

:PARENT #S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0)

:DEPTH 13

:G ...))

**CG-USER**(98): (mapcar #'(lambda (x) (node-state x)) sol-ex7)

(PARIS NANCY TOULOUSE)

**CG-USER**(99): (mapcar #'(lambda (x) (node-g x)) sol-ex7)

(0 50 75.0)

Si no introducimos ninguna estrategia válida:

**CG-USER**(132): (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7)

lst-nodes-ex6

nil)

NIL

Si no introducimos nodos a ordenar:

**CG-USER**(133): (insert-nodes-strategy nil

lst-nodes-ex6

\*uniform-cost\*)

(#S(NODE :STATE TOULOUSE

:PARENT #S(NODE :STATE MARSEILLE :PARENT NIL :ACTION NIL :DEPTH 12 :G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0)

:DEPTH 13

:G ...))

**CG-USER**(134): (equal (insert-nodes-strategy nil

lst-nodes-ex6

\*uniform-cost\*) lst-nodes-ex6)

T

Si no introducimos una lista de nodos ordenados:

**CG-USER**(136): (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7) nil

\*uniform-cost\*)

(#S(NODE :STATE PARIS :PARENT NIL :ACTION NIL :DEPTH 0 :G ...) #S(NODE :STATE NANCY :PARENT NIL :ACTION NIL :DEPTH 2 :G ...))

**CG-USER**(137): (mapcar #'(lambda (x) (node-state x)) (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7) nil

\*uniform-cost\*)

)

(PARIS NANCY)

**CG-USER**(138): (mapcar #'(lambda (x) (node-g x)) (insert-nodes-strategy (list node-paris-ex7 node-nancy-ex7) nil

\*uniform-cost\*))

(0 50)

**Ejercicio 8**

En este ejercicio solo nos piden inicializar una variable global cuyo valor sea la estrategia para realizar la búsqueda A\*.

**Pseudocódigo de la función que compara dos nodos:**

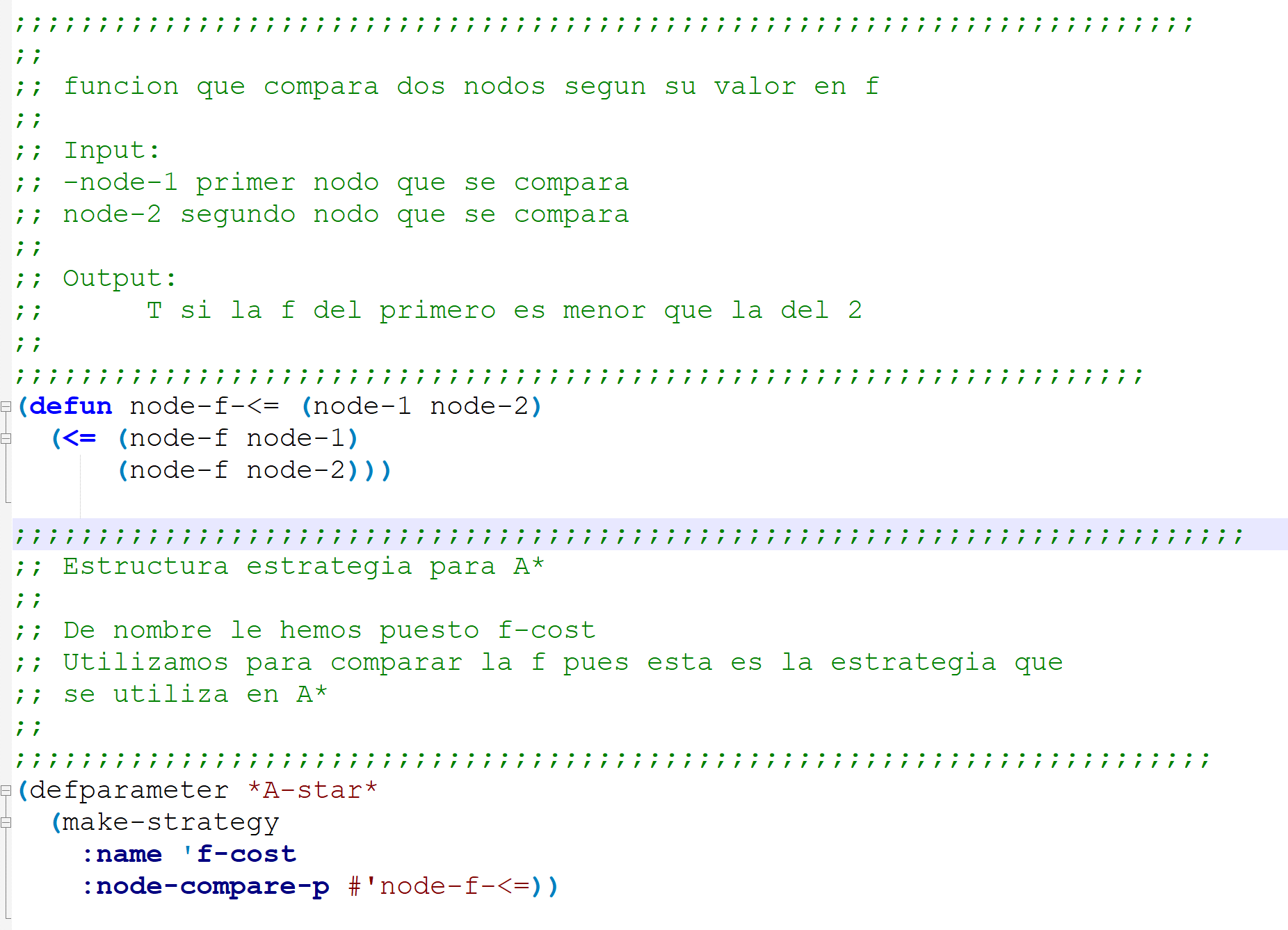
**Node-f-<= (node-1 node-2)**

Input: node-1(nodo), node-2(nodo)

Output: T si el node-1 va antes que el 2(tiene menor f) o nil si no es así

Proceso:

Devolver obtener-f(node-1) **<=** obtener-f(node-2)



**Ejercicio9**

En este ejercicio nos piden codificar una función de búsqueda en un grafo, y otra que sea concretamente búsqueda utilizando la estrategia A\*.

Para este ejercicio hemos seguido el pseudocódigo proporcionado en el enunciado.

**Pseudocódigo de las funciones:**

**Devolver-nodo(node lst)**

Input: node(nodo que buscamos), lst(lista donde lo buscamos)

Output: Res(nodo que sea igual(según la función de mismo estado) al node y pertenezca a lst)

Proceso:

Si lst = null:

Devolver nil

Si node = lst[0]:

Devolver lst[0]

Si no:

Devolver devolver-nodo(node, rest(lst))

**Graph-search-aux(problem open-nodes closed-nodes strategy)**

Input: problem(problema, que contiene la informacion general), open-nodes(lista de nodos descubiertos pero no explorados), closed-nodes(lista de nodos ya explorados), strategy(estrategia que vamos a seguir para resolver la búsqueda).

Output: Res, nodo que corresponde al estado objetivo o nil en caso de no encontrarse solución

Proceso:

Si open-nodes = null:

Devolver nil

Si no:

Primer-nodo = open-nodes[0]

Si primer-nodo = OBJETIVO:

Devolver primer-nodo

Copia = devolver-node(primer-nodo, closed-nodes)

Si copia = null:

Open-nodes = añadir-a-cdr-open-nodes(expand-node-strategy(primer-nodo))

Devolver graph-search-aux(problema, open-nodes,

Añadir-aprimer-node, closed-nodes), strategy)

Si obtener-g(primer-nodo) **<=** obtener-g(copia):

Open-nodes = añadir-a-cdr-open-nodes(expand-node-strategy(primer-nodo))

Devolver graph-search-aux(problema, open-nodes,

Añadir-aprimer-node, closed-nodes), strategy)

Si no:

Devolver graph-search-aux(problem, rest(open-nodes), añadir-a(primer-nodo, closed-nodes), strategy)

**Graph-search(problem strategy)**

Input: problem(estructura problema, contiene la información general), strategy(estrategia para seguir en la búsqueda)

Output: Nodo final objetivo o nil si no se ha encontrado

Proceso:

Devolver graph-search-aux(problema, crear-lista(make-node(problema-inicio)),nil, strategy)

**a-star-search (problem)**

Input: problem(problema, contiene la información general que necesitaremos para resolver la búsqueda con A\*)

Output: Output: Nodo final objetivo o nil si no se ha encontrado

Proceso:

Devolver: graph-search(problema **\*A-star\***)

**Ejemplos de ejecución ejercicio 9:**

**CG-USER**(13): (graph-search \*travel-cheap\* \*A-star\*)

#S(NODE :STATE CALAIS

:PARENT #S(NODE :STATE REIMS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE :STATE NEVERS

:PARENT #S(NODE :STATE LIMOGES :PARENT # :ACTION # :DEPTH 2 :G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS :COST 60.0)

:DEPTH 3

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS :COST 10.0)

:DEPTH 4

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS :COST 10.0)

:DEPTH 5

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0)

:DEPTH 6

:G ...)

\*\*Si introducimos parámetros de entrada no válidos devolverá nil

**CG-USER**(14): (graph-search nil \*A-star\*)

NIL

**CG-USER**(15): (graph-search \*travel-cheap\* nil)

NIL

**CG-USER**(16): (graph-search \*travel-cheap\* 'jola)

NIL

**CG-USER**(17): (graph-search 'novalido \*A-star\*)

NIL

**CG-USER**(23): (a-star-search \*travel-fast\*)

#S(NODE :STATE CALAIS

:PARENT #S(NODE :STATE PARIS

:PARENT #S(NODE :STATE ORLEANS

:PARENT #S(NODE :STATE LIMOGES

:PARENT #S(NODE :STATE TOULOUSE :PARENT # :ACTION # :DEPTH 1 :G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN TOULOUSE :FINAL LIMOGES :COST 25.0)

:DEPTH 2

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LIMOGES :FINAL ORLEANS :COST 55.0)

:DEPTH 3

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN ORLEANS :FINAL PARIS :COST 23.0)

:DEPTH 4

:G ...)

:ACTION #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS :COST 34.0)

:DEPTH 5

:G ...)

**Ejercicio10**

En esta función nos piden calcular el camino(nombre de las ciudades) hasta llegar a un nodo (resultado de un búsqueda) y la secuencia de acciones para llegar a un nodo.

**Pseudocódigo de las funciones:**

**Solution-path-aux(node lst)**

Input: node(nodo del que buscamos el camino), lst(lista de estados acumulados)

Output: lista de estados

Proceso:

Si node-parent = null:

Devolver añadir-alista(node-state, lst)

añadir-a-lista (node-state, lst)

devolver añadir-a-lista(node-path-aux(node-parent), lst)

**solution-path(node)**

Input: node(nodo del que buscamos el camino)

Output: lista de estados hasta llegar al nodo

Proceso:

Devolver solution-path-aux(node, nil)

**Action-sequence-aux(node, lst)**

Input: node(nodo del que buscamos secuencias de acciones), lst(acciones acumuladas hasta ahora)

Output: lista de secuencia de acciones

Proceso:

Si node-parent = null

Devolver: añadir-a-lista(node-accion, lst)

Si no:

Añadir-a-lista(node-accion, lst)

Devolver añadir-a-lista(action-sequence-aux(node, lst), lst)

**Action-sequence(node)**

Input: node(nodo del cual queremos saber la secuencia de acciones hasta él)

Output: lista de acciones

Proceso:

Si node-parent = null

Devolver: node-accion

Si no:

Devolver action-sequence-aux(node-parent,lista-de(node-action))

**Ejemplos de ejecución del ejercicio 10:**

**CG-USER**(30): (solution-path nil)

NIL

**CG-USER**(31): (solution-path (a-star-search \*travel-fast\*))

(MARSEILLE TOULOUSE LIMOGES ORLEANS PARIS CALAIS)

**CG-USER**(32): (solution-path (a-star-search \*travel-cheap\*))

(MARSEILLE TOULOUSE LIMOGES NEVERS PARIS REIMS CALAIS)

**CG-USER**(58): (action-sequence (a-star-search \*travel-fast\*))

(#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 65.0)

#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN TOULOUSE :FINAL LIMOGES :COST 25.0)

#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN LIMOGES :FINAL ORLEANS :COST 55.0) #S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN ORLEANS :FINAL PARIS :COST 23.0)

#S(ACTION :NAME NAVIGATE-TRAIN-TIME :ORIGIN PARIS :FINAL CALAIS :COST 34.0))

**CG-USER**(59): (action-sequence (a-star-search \*travel-cheap\*))

(#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN MARSEILLE :FINAL TOULOUSE :COST 120.0)

#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN TOULOUSE :FINAL LIMOGES :COST 35.0)

#S(ACTION :NAME NAVIGATE-TRAIN-PRICE :ORIGIN LIMOGES :FINAL NEVERS :COST 60.0) #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN NEVERS :FINAL PARIS :COST 10.0)

#S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN PARIS :FINAL REIMS :COST 10.0) #S(ACTION :NAME NAVIGATE-CANAL-PRICE :ORIGIN REIMS :FINAL CALAIS :COST 15.0))

**Ejercicio11**

En este ejercicio nos piden definir estrategias y funciones de comparación de nodos para búsqueda en anchura y búsqueda en profundidad.

**Pseudocódigo de las funciones:**

**Depth-first-node-compare-p(node-1 node-2)**

Input: node-1, node-2 (nodos que queremos comparar)

Output: T si node-1 tiene una profundidad mayor o igual a node-2 y nil en caso contrario.

Proceso:

Si profundidad(node-1) >= profundidad(node-2):

Devolver T

Si no:

Devolver nil

**Breadth-first-node-compare-p(node-1 node-2)**

Input: node-1, node-2 (nodos que queremos comparar)

Output: T si node-1 tiene una profundidad menor o igual a node-2 y nil en caso contrario.

Proceso:

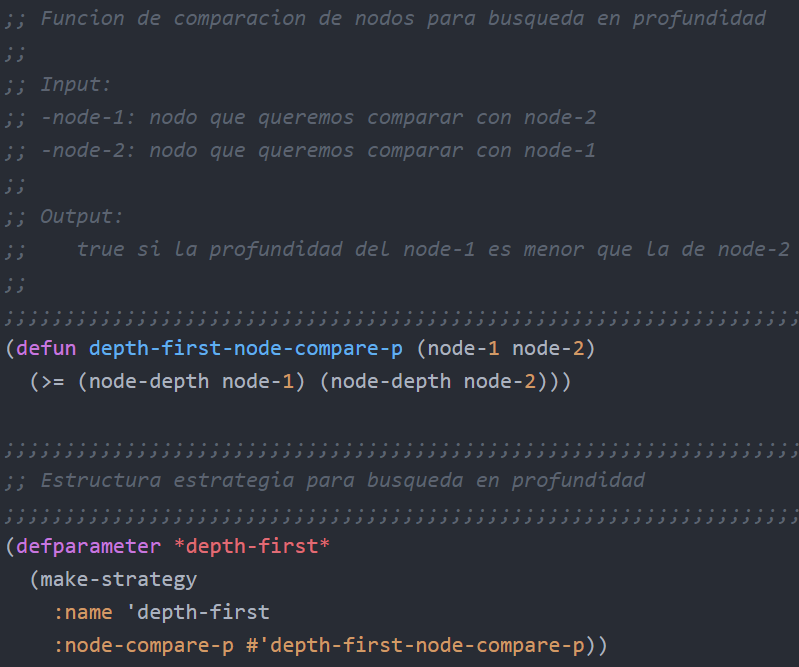
Si profundidad(node-1) <= profundidad(node-2):

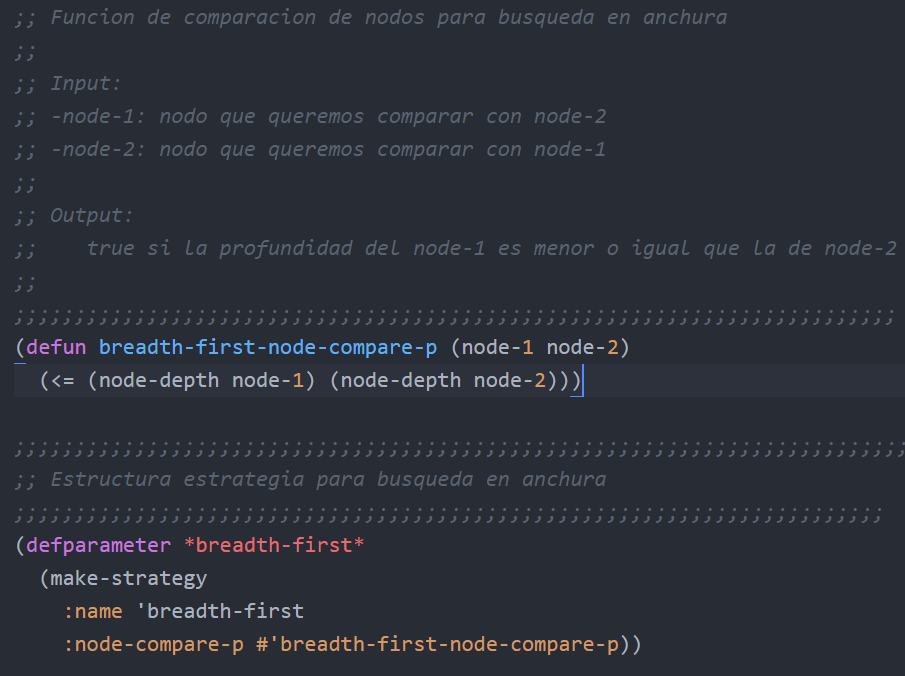
Devolver T

Si no:

Devolver nil

**Código de las funciones:**





**Ejemplos de ejecución ejercicio11:**

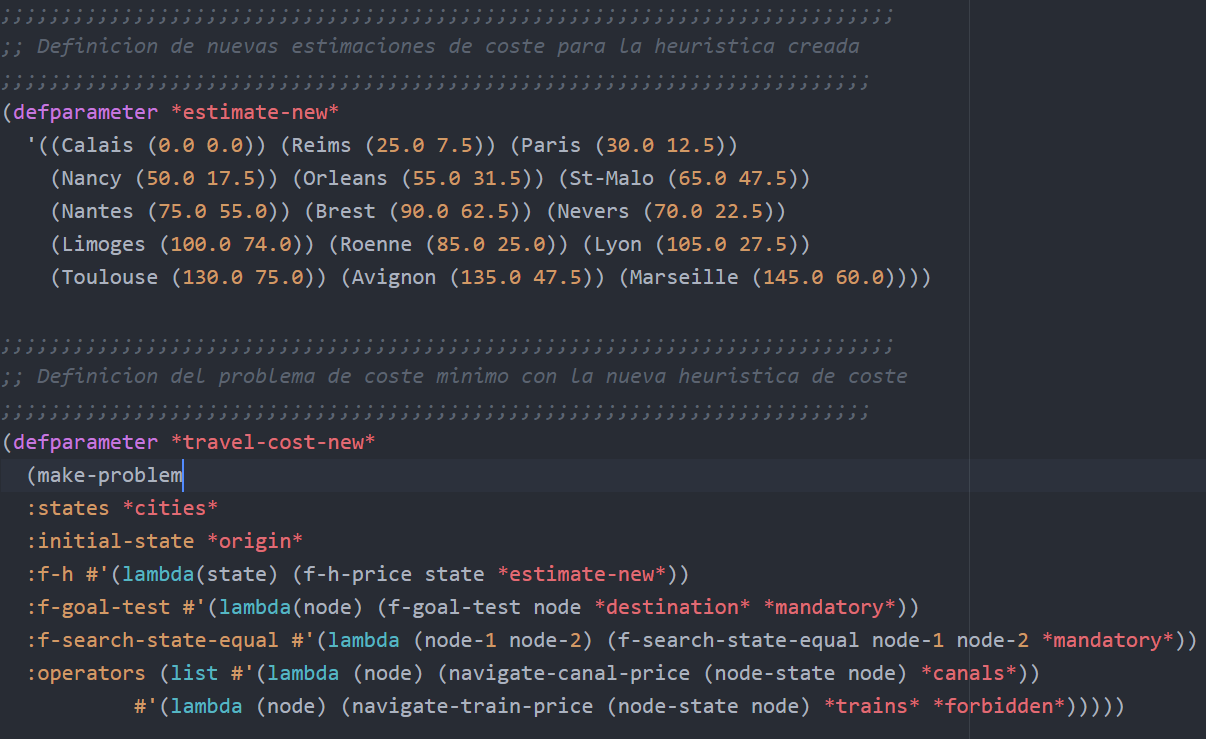




**Ejercicio12**

En este último ejercicio tenemos que crear una heurística admisible para la opción de viajar barato. Definimos la heurística de manera que a cada nodo le corresponde la mitad del coste del camino hasta el nodo objetivo. Esto, teniendo en cuenta el camino más corto y, si se puede viajar en canal, teniendo en cuenta el precio del canal dado que es más barato. Esta heurística es admisible ya que por definición es menor que el coste real en cualquier caso.

Después de realizar los cálculos obtenemos la siguiente lista de estimaciones y la siguiente definición del problema de coste mínimo con la nueva heurística:



A la hora de medir los tiempos de ejecución de la búsqueda con la heurística de coste cero y la recién definida por nosotros obtenemos los siguientes resultados en los que podemos notar una notable mejora en el tiempo de búsqueda y una buena reducción del espacio ocupado:

