Instrucciones de uso del programa laberinto

Pedro González Ruiz

13 de agosto de 2017

1. Introducción

Este programa resuelve el problema de cómo salir de un laberinto. Está hecho en *Common Lisp*. No es necesario conocer el lenguaje. Siguiendo las instrucciones podrá utilizarlo sin ningún problema. En las secciones siguientes se muestran los pasos a seguir.

2. Descargar e instalar el intérprete de common lisp

El intérprete de Lisp utilizado es `clisp`. Descárguelo e instálelo. En un emulador de terminal, entrar:

```
clisp
```

y debe aparecer lo siguiente:

```
pedro@servidor2:~/common-lisp/programas/laberinto/fuentes> clisp
  i i i i i i i       ooooo    o        ooooooo   ooooo   ooooo
  I I I I I I I      8     8   8           8     8     o  8    8
  I  \ `+' /  I      8         8           8     8        8    8
   \  `-+-'  /       8         8           8      ooooo   8oooo
    `-__|__-'        8         8           8           8  8
        |            8     o   8           8     o     8  8
  ------+------       ooooo    8ooooooo  ooo8ooo   ooooo   8

Welcome to GNU CLISP 2.49 (2010-07-07) <http://clisp.cons.org/>

Copyright (c) Bruno Haible, Michael Stoll 1992, 1993
Copyright (c) Bruno Haible, Marcus Daniels 1994-1997
Copyright (c) Bruno Haible, Pierpaolo Bernardi, Sam Steingold 1998
Copyright (c) Bruno Haible, Sam Steingold 1999-2000
Copyright (c) Sam Steingold, Bruno Haible 2001-2010

Type :h and hit Enter for context help.
```

Veamos algunas pruebas con el intérprete para ver que todo va bien. Comencemos por un par de cálculos:

```
[1]> (* 2 5)
10
[2]> (expt 2 100)
1267650600228229401496703205376
```

es decir $2 \times 5 = 10$, y $2^{100} = 1267650600228229401496703205376$. Si cometemos un error, entramos en depuración, del cual se sale con `Ctrl-D`. Por ejemplo:

```
[3]> (+ 2 5 a)

*** - SYSTEM::READ-EVAL-PRINT: variable A has no value
Es posible continuar en los siguientes puntos:
USE-VALUE      :R1      Input a value to be used instead of A.
STORE-VALUE    :R2      Input a new value for A.
ABORT          :R3      Abort main loop
Break 1 [4]>
```

En esta situación pulsamos `Ctrl-D` y volvemos al prompt habitual, es decir:

```
[5]>
```

3. Ejecución del programa

Los archivos necesarios son `laberinto.lsp` y `ejemplos.lsp`. El primero es el programa y el segundo contiene algunos ejemplos de laberintos. Han de estar juntos en el mismo directorio. El primero carga al segundo.

Si ejecuta el intérprete en un emulador de terminal, es posible que lo que se muestra aquí no se vea allí correctamente debido a los caracteres gráficos utilizados. He utilizado *xterm* y con los valores por defecto, no va bien. Para que funcione, haga lo siguiente: ejecute *xterm* y acto seguido pulse la tecla `Ctrl` y mientras la mantiene presionada pulse el botón derecho del ratón. Sale un menú en el que la opción por defecto en *VT Fonts* es `Default`. Cambie a la que sigue, es decir, `Unicode Best` y todo debe ir bien.

Si ejecuta el intérprete dentro de *emacs*, todavía mejor. Entre en *emacs* y pulse `Alt x`. En la línea inferior (minibuffer) aparece `M-x`. Escriba la palabra `shell`, y pulse enter. Esto crea un emulador de terminal dentro de *emacs*. Cámbiese al directorio donde está el programa y ejecute el intérprete, es decir, entre `clisp`. En *emacs*, para moverse en el histórico de órdenes, las combinaciones de teclas son `Alt p` y `Alt n`, o bien, moverse con las flechas arriba y abajo, y pulsar enter. Si es posible, haga las prácticas que siguen con *emacs* mejor que con un emulador de terminal.

En fin, dentro del intérprete de Lisp, entrar:

```
[1]> (load "laberinto.lsp")
;; Loading file laberinto.lsp ...
;;  Loading file ejemplos.lsp ...
;;  Loaded file ejemplos.lsp
;; Loaded file laberinto.lsp
T
```

Veamos ahora algunos ejemplos. El primer laberinto:

```
[2]> lab01
#2A((0 0 1 0) (1 1 1 1) (0 1 0 0) (0 0 1 0))
```

No muy interesante, pero importante. Un laberinto es una matriz bidimensional (cuadrada o no) de ceros y unos. Un 0 indica muro, es decir, por ahí no se puede pasar, mientras que un 1 indica paso libre, es decir, por ahí, sí se puede pasar. Veamos una imagen más gráfica del laberinto:

```
[3]> (imprime-laberinto lab01)
     00  01  02  03
   +---+---+   +---+
00 |///|///|   |///|
   +---+---+   +---+
01
   +---+   +---+---+
02 |///|   |///|///|
   +---+---+---+---+
03 |///|///|   |///|
   +---+---+   +---+
NIL
```

Los laberintos incluidos en `ejemplos.lsp` son:

`lab01, lab02, lab03, lab04, lab05`

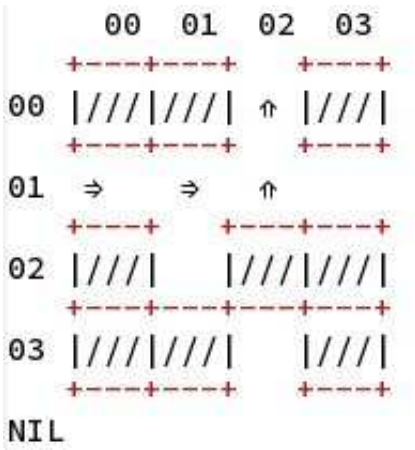
Si los quiere mostrar todos, introduzca:

```
[3]> (mapcar #'imprime-laberinto (list lab01 lab02 lab03 lab04 lab05))
```

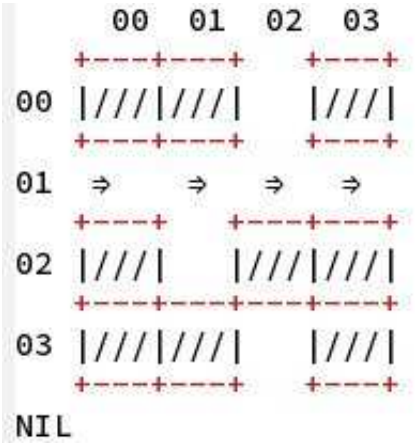
Veamos ahora como salir del laberinto. La función para ello es `solab`. Entramos:

```
[3]> (solab lab01 '(1 0))
```

que quiere decir, ¿cómo salir del laberinto `lab01`, comenzando en la posición (1 0)? (el apóstrofo `'` es necesario por la naturaleza del lenguaje *Lisp*). La solución es:



o también

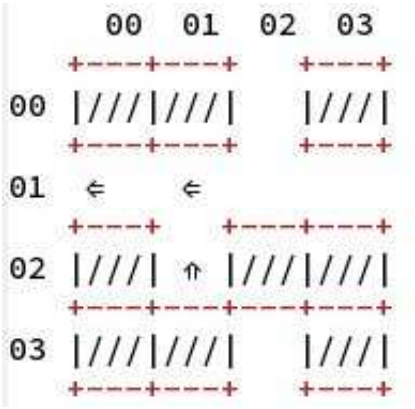


En otras palabras, ejecutando la misma función con los mismos argumentos, la solución puede ser distinta. Esto es así, porque cuando el programa llega a un lugar donde hay varias opciones para seguir, elige aleatoriamente una de ellas.

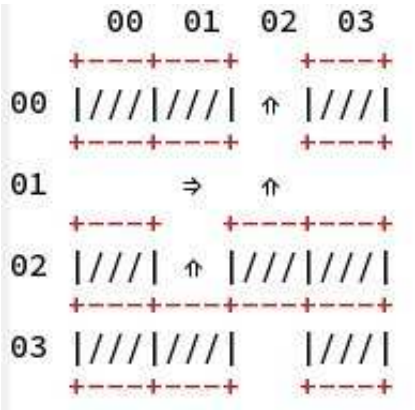
Veamos más ejemplos:

```
[3]> (solab lab01 '(2 1))
```

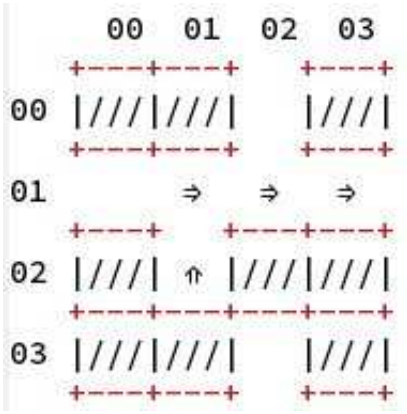
y obtenemos:



o bien



también



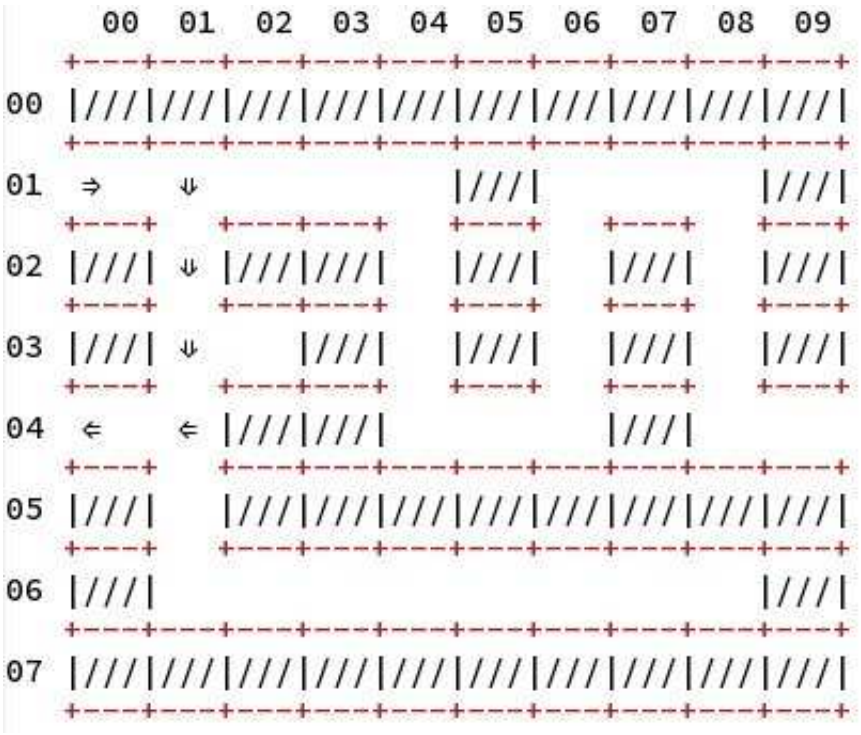
Hagamos prácticas con uno más grande:

```
[13]> (imprime-laberinto lab04)
     00  01  02  03  04  05  06  07  08  09
   +---+---+---+---+---+---+---+---+---+---+
00 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
01                     |///|           |///|
   +---+   +---+---+   +---+   +---+   +---+
02 |///|   |///|///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
03 |///|       |///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
04         |///|///|           |///|
   +---+   +---+---+---+---+---+---+---+---+
05 |///|   |///|///|///|///|///|///|///|///|
   +---+   +---+---+---+---+---+---+---+---+
06 |///|                               |///|
   +---+---+---+---+---+---+---+---+---+---+
07 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
NIL
```

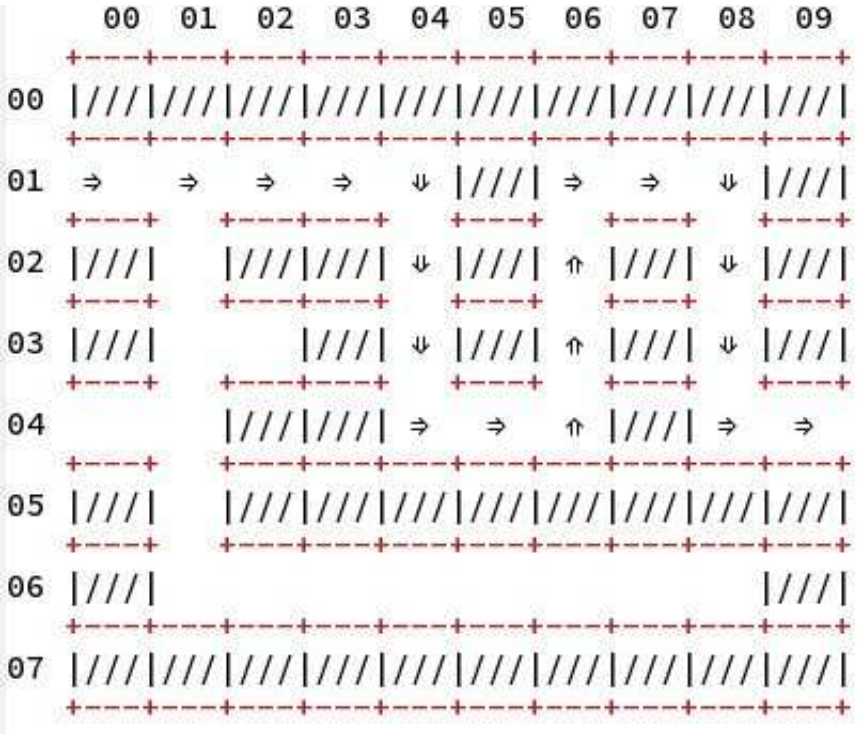
Veamos algunos ejemplos. Entramos por una puerta:

```
[5]> (solab lab04 '(1 0))
```

y obtenemos:

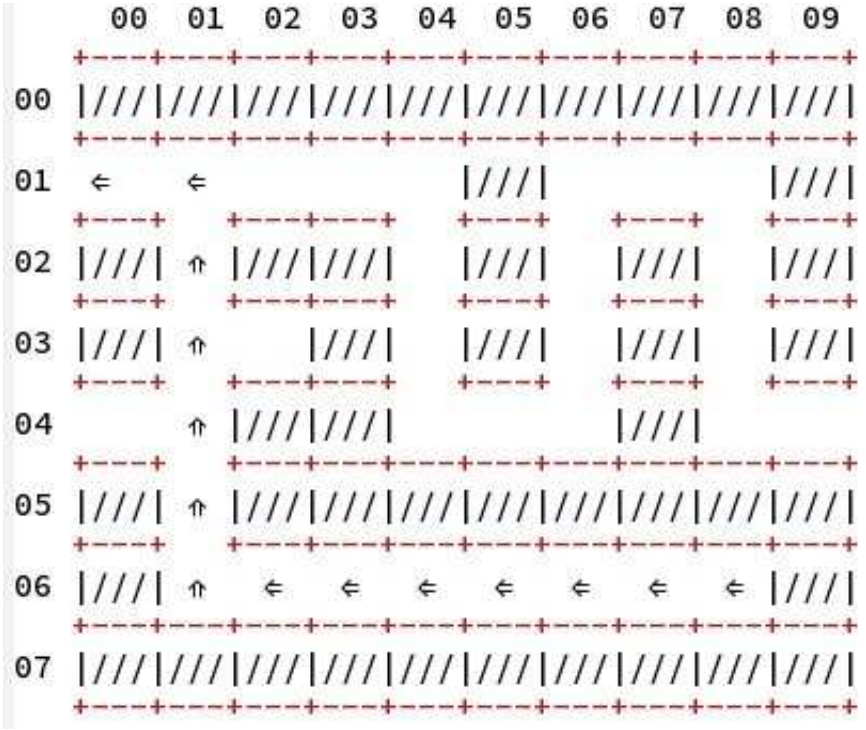


o bien



Aunque también podemos comenzar el recorrido en cualquier punto del interior:

```
[6]> (solab lab04 '(6 8))
```



Pruebe varias veces la misma orden para obtener diferentes soluciones. El laberinto tiene tres puertas: la (1 0), (4 0) y la (4 9). Podemos cerrar interactivamente la que queramos, para ello, utilizamos la siguiente expresión:

```
(setf (aref lab04 1 0) 0)
```

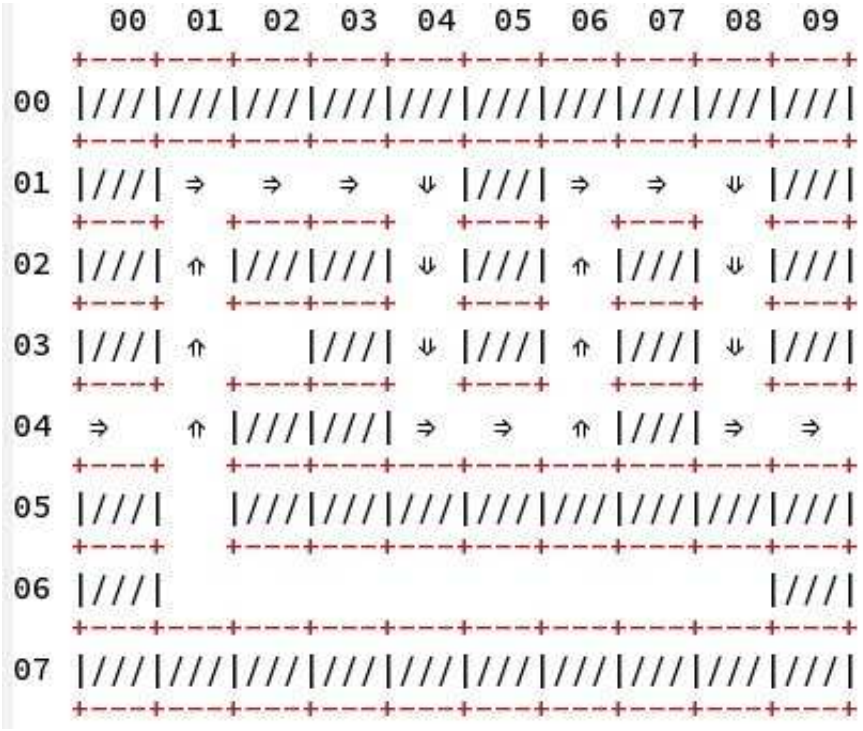
La función `aref` es (array reference) y lo que quiere decir es que ponemos a 0 el elemento de coordenadas (1 0) de la matriz `lab04`. Comprobamos:

```
[25]> (imprime-laberinto lab04)
     00  01  02  03  04  05  06  07  08  09
   +---+---+---+---+---+---+---+---+---+---+
00 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
01 |///|               |///|           |///|
   +---+   +---+---+   +---+   +---+   +---+
02 |///|   |///|///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
03 |///|       |///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
04         |///|///|           |///|
   +---+   +---+---+---+---+---+---+---+---+
05 |///|   |///|///|///|///|///|///|///|///|
   +---+   +---+---+---+---+---+---+---+---+
06 |///|                               |///|
   +---+---+---+---+---+---+---+---+---+---+
07 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
NIL
```

En fin:

```
[10]> (solab lab04 '(4 0))
```

y obtenemos:



Cerramos otra puerta:

```
(setf (aref lab04 4 9) 0)
```

Comprobamos:

```
[29]> (imprime-laberinto lab04)
     00  01  02  03  04  05  06  07  08  09
   +---+---+---+---+---+---+---+---+---+---+
00 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
01 |///|               |///|           |///|
   +---+   +---+---+   +---+   +---+   +---+
02 |///|   |///|///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
03 |///|       |///|   |///|   |///|   |///|
   +---+   +---+---+   +---+   +---+   +---+
04         |///|///|           |///|   |///|
   +---+   +---+---+---+---+---+---+---+---+
05 |///|   |///|///|///|///|///|///|///|///|
   +---+   +---+---+---+---+---+---+---+---+
06 |///|                               |///|
   +---+---+---+---+---+---+---+---+---+---+
07 |///|///|///|///|///|///|///|///|///|///|
   +---+---+---+---+---+---+---+---+---+---+
NIL
```

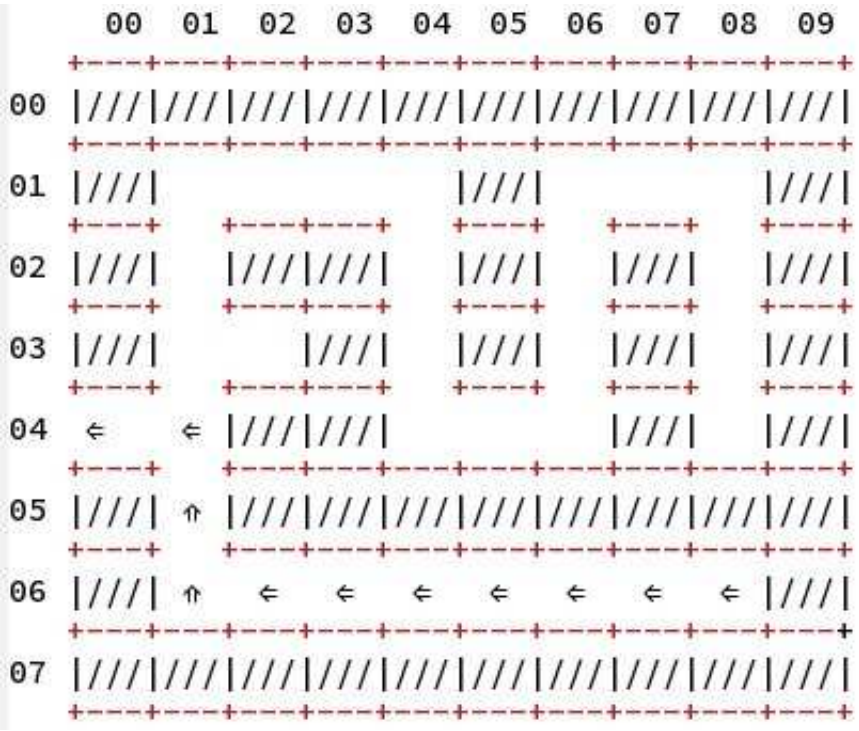
Ahora no es lo mismo:

```
[30]> (solab lab04 '(4 0))
Laberinto sin solución.
NIL
```

En fin, como se ve trivialmente, no se puede. Ahora bien:

```
[11]> (solab lab04 '(6 8))
```

Sin pegas, hemos salido sin problemas.



4. Algoritmo del laberinto

El algoritmo de salida del laberinto (función `solab`) en palabras, es el siguiente:

1. En primer lugar, comprueba si hemos salido, es decir, si el problema está resuelto. Esto ocurre cuando estamos situados en una puerta, es decir, un lugar de paso en el borde del laberinto distinto de la posición inicial. Si esto es así, hemos resuelto el problema y hemos acabado.

2. En caso contrario, buscamos posiciones adyacentes al lugar donde nos encontramos **no visitadas** (para evitar entrar en un bucle sin fin). El lugar donde nos encontramos se llama **posición actual**, y al comienzo, ésta coincide con la **posición original**.

   Pueden ocurrir dos cosas:

   - Que **existan** tales posiciones. En éste caso, el programa elige al azar una de ellas, ignorando las demás. La elegida se marca como visitada, se introduce en `camino` y pasaa ser la `actual`. Volvemos al punto 1.

- Que **no existan** tales posiciones. Hemos entrado en un callejón sin salida y no podemos seguir avanzando. La elección que hicimos en el apartado anterior fue errónea. Hemos de retroceder y rectificar el error, para ello, eliminamos la posición `actual` del `camino`, e intentamos retroceder yendo a la anterior. Y pueden pasar otra vez dos cosas:
  - **Hay anterior**. Aquí se produce el retroceso. Cogemos el primer elemento de `camino` y éste es el nuevo `actual`. Volvemos al punto 1.
  - **No hay anterior**. En otras palabras, no se puede retroceder. Si esto ocurre, el problema no tiene solución y hemos acabado.

La función `solab` maneja tres variables que son:

- `actual`. Es la posición del laberinto en la que nos encontramos, en forma de coordenadas $(x\ y)$, siendo x el número de la fila e y el número de la columna. La numeración comienza en 0, por tanto, para una matriz $m \times n$ (m filas y n columnas), debe ser:

$$0 \leq x < m,\ 0 \leq y < n$$

- `lv`. Lista de posiciones visitadas, es decir:

$$\texttt{lv} = \big((x_0\ y_0)\ (x_1\ y_1)\ \ldots\ (x_r\ y_r)\big)$$

  Al comienzo es:

$$\texttt{lv} = \big(\text{(posición original)}\big)$$

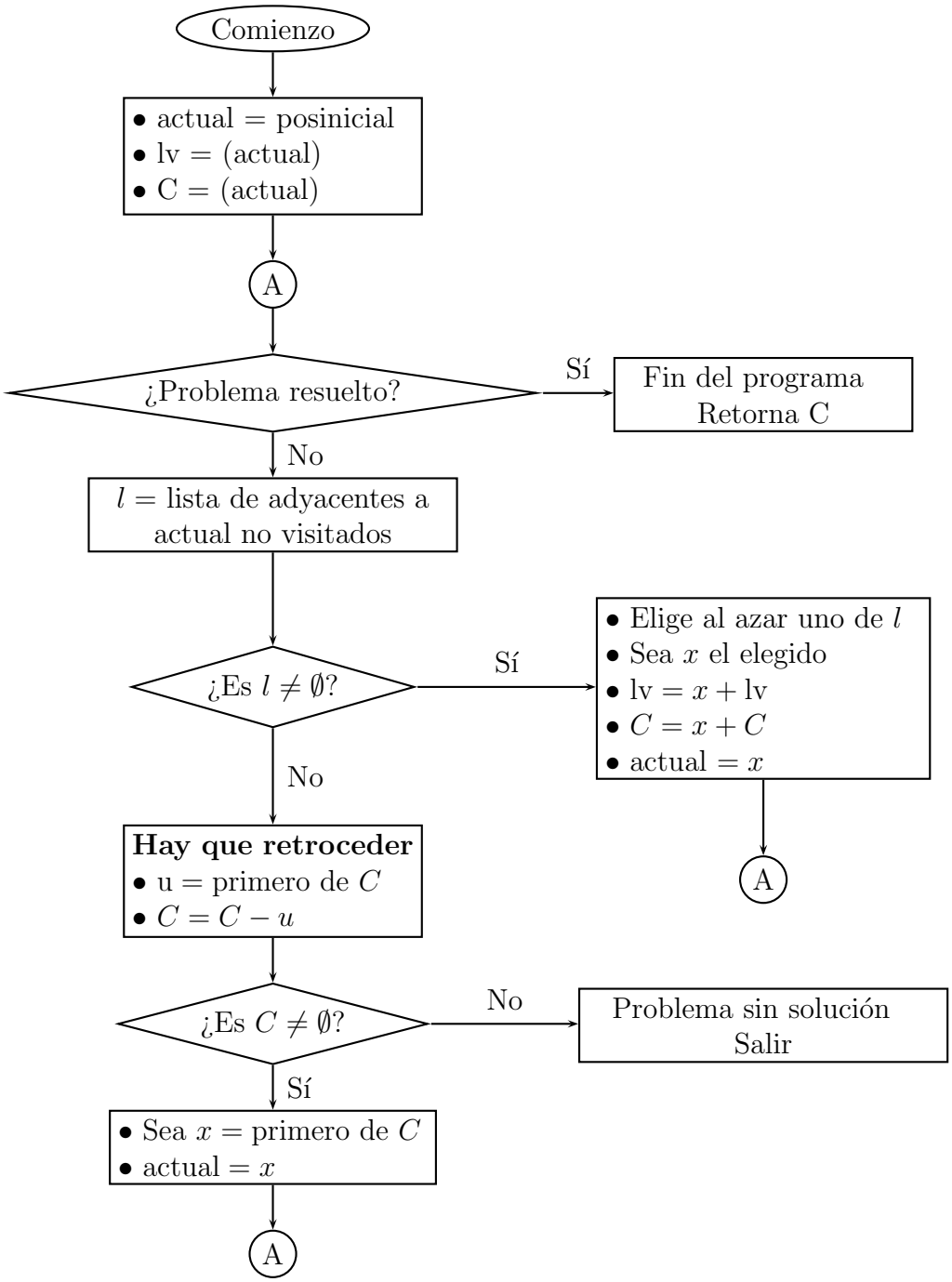
- `C`. Camino seguido, una lista con la misma forma que la anterior, es decir:

$$\texttt{C} = \big((u_s\ v_s)\ (u_{s-1}\ v_{s-1})\ \ldots\ (u_0\ v_0)\big)$$

  Cuando hay solución, la función retorna esta lista. Las inserciones se producen al comienzo (ya que en Lisp es menos costoso que hacerlo al final). Debido a esto, el camino está invertido y hay que ponerlo derecho. Al comienzo es:

$$\texttt{C} = \big(\text{(posición original)}\big)$$

El diagrama de flujo del algoritmo es el siguiente:



4.1. Condiciones para que un laberinto sea válido

Para que un laberinto sea considerado como válido y la función `solab` funcione bien, deben cumplirse las condiciones que a continuación se explican. Utilizamos la notación matricial: si el laberinto es l, tiene m filas y n columnas, entonces, sus elementos se escriben en la notación habitual, es decir, $l_{i,j}$, $0 \leq i < m$, $0 \leq j < n$.

1. $m \geq 3$ y $n \geq 3$.

2. La posición inicial posini = $(x_0\ y_0)$ (parámetro que se envía a `solab`), ha de cumplir:

$$0 \leq x_0 < m,\ 0 \leq y_0 < n$$

3. La posición inicial posini = $(x_0\ y_0)$ tiene que ser un blanco, es decir:

$$l_{x_0,y_0} = 1$$

   En otras palabras, no podemos empezar el recorrido en un muro.

4. Las cuatro esquinas del laberinto tienen que ser muros, es decir:

$$l_{0,0} = l_{0,n-1} = l_{m-1,0} = l_{m-1,n-1} = 0$$

5. No puede haber dos puertas seguidas en el contorno del laberinto. Se entiende por **puerta** una posición abierta (= 1) en el contorno.

Si alguna no se cumple, la variable externa **caderro** contiene una descripción del error.

Veamos algunos ejemplos:

```
[2]> (imprime-laberinto lab01)
     00  01  02  03
   +---+---+   +---+
00 |///|///|   |///|
   +---+---+   +---+
01
   +---+   +---+---+
02 |///|   |///|///|
   +---+---+---+---+
03 |///|///|   |///|
   +---+---+   +---+

[5]> (comprueba-laberinto lab01 '(0 2))
T
```

Sin pegas. `T` quiere decir `True` (verdadero, en inglés), es decir, vale. Ahora bien:

```
[6]> (comprueba-laberinto lab01 '(0 1))
NIL
[7]> caderro
"Error en el punto de comienzo. No es blanco.
"
```

Forcemos la situación:

```
[8]> (setf (aref lab01 0 1) 1)
1
[9]> (imprime-laberinto lab01)
     00  01  02  03
   +---+       +---+
00 |///|       |///|
   +---+       +---+
01
   +---+   +---+---+
02 |///|   |///|///|
   +---+---+---+---+
03 |///|///|   |///|
   +---+---+   +---+
```

Y ahora:

```
[10]> (comprueba-laberinto lab01 '(0 1))
NIL
[11]> caderro
"Dos blancos consecutivos en la fila 0: (0 1), (0 2)
"
```

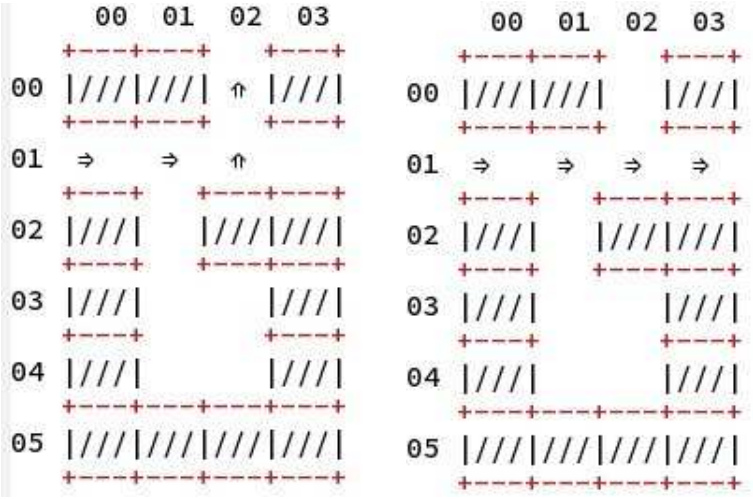
5. Todas las soluciones

El procedimiento expuesto antes da una solución, pero no todas. En este apartado, resolvemos éste problema, es decir, partiendo de una posición cualquiera en blanco del laberinto (una puerta o interior), mostraremos todas las formas de salir. El algoritmo es un arreglo del anteriormente expuesto, en concreto, cuando llegábamos a un cruce con varias posibilidades, elegíamos una al azar no visitada previamente, y seguíamos por ahí ignorando las otras. Ahora no. Cuando lo anterior ocurra, de las posibles alternativas, elegimos la primera y el resto de movimientos se guarda en una pila para posteriormente volver a ellos. El programa ha acabado cuando la pila está vacía (pila = $\emptyset$), es decir, no queda movimiento que hacer. La función es `tsolab`, igual que `solab`, salvo que tiene una `t` delante, por todos.

Veamos algunos ejemplos:

```
[4]> (imprime-laberinto lab02)
     00  01  02  03
   +---+---+   +---+
00 |///|///|   |///|
   +---+---+   +---+
01
   +---+   +---+---+
02 |///|   |///|///|
   +---+   +---+---+
03 |///|       |///|
   +---+       +---+
04 |///|       |///|
   +---+---+---+---+
05 |///|///|///|///|
   +---+---+---+---+
[5]> (tsolab lab02 '(1 0))
```

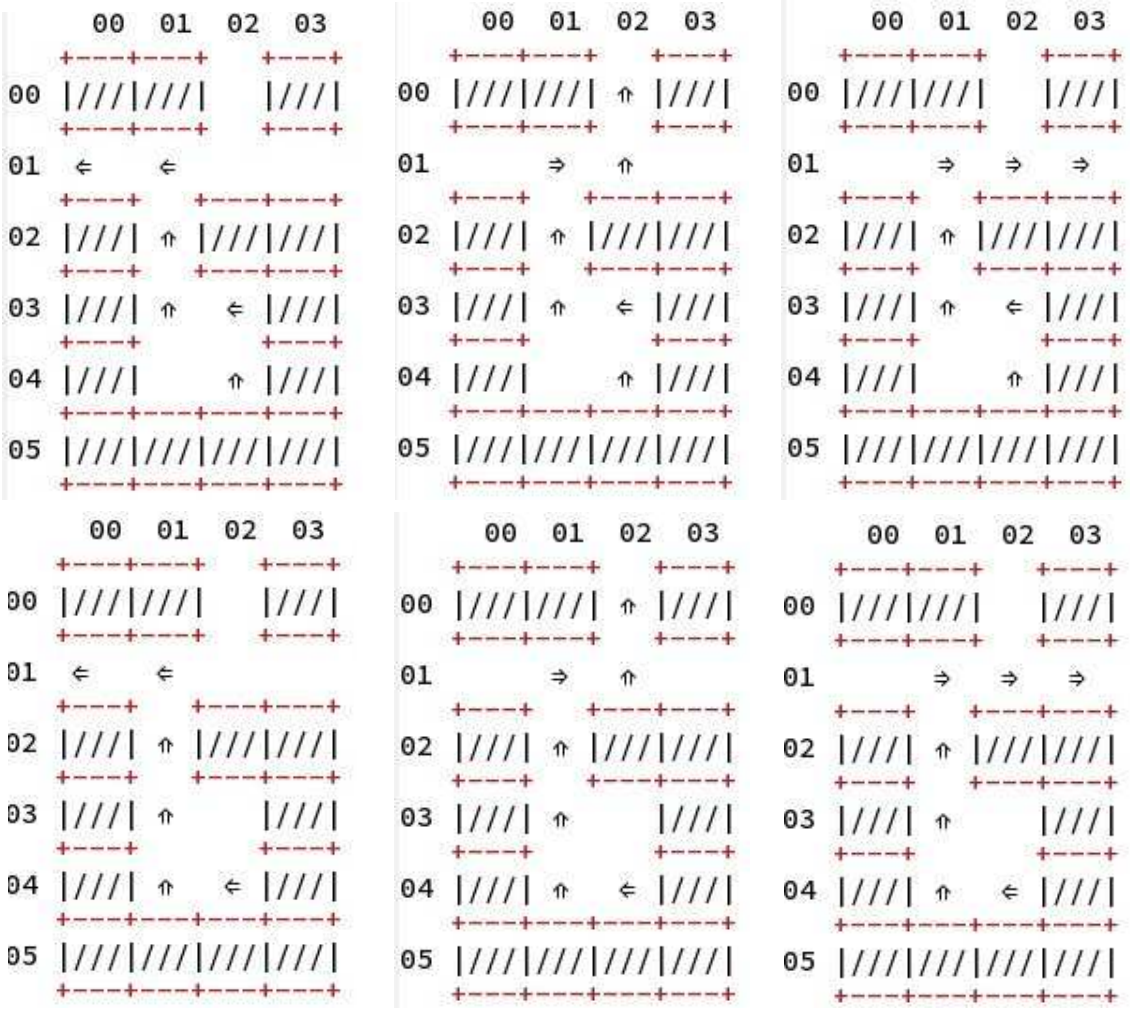
y la salida es:



Más complicado:

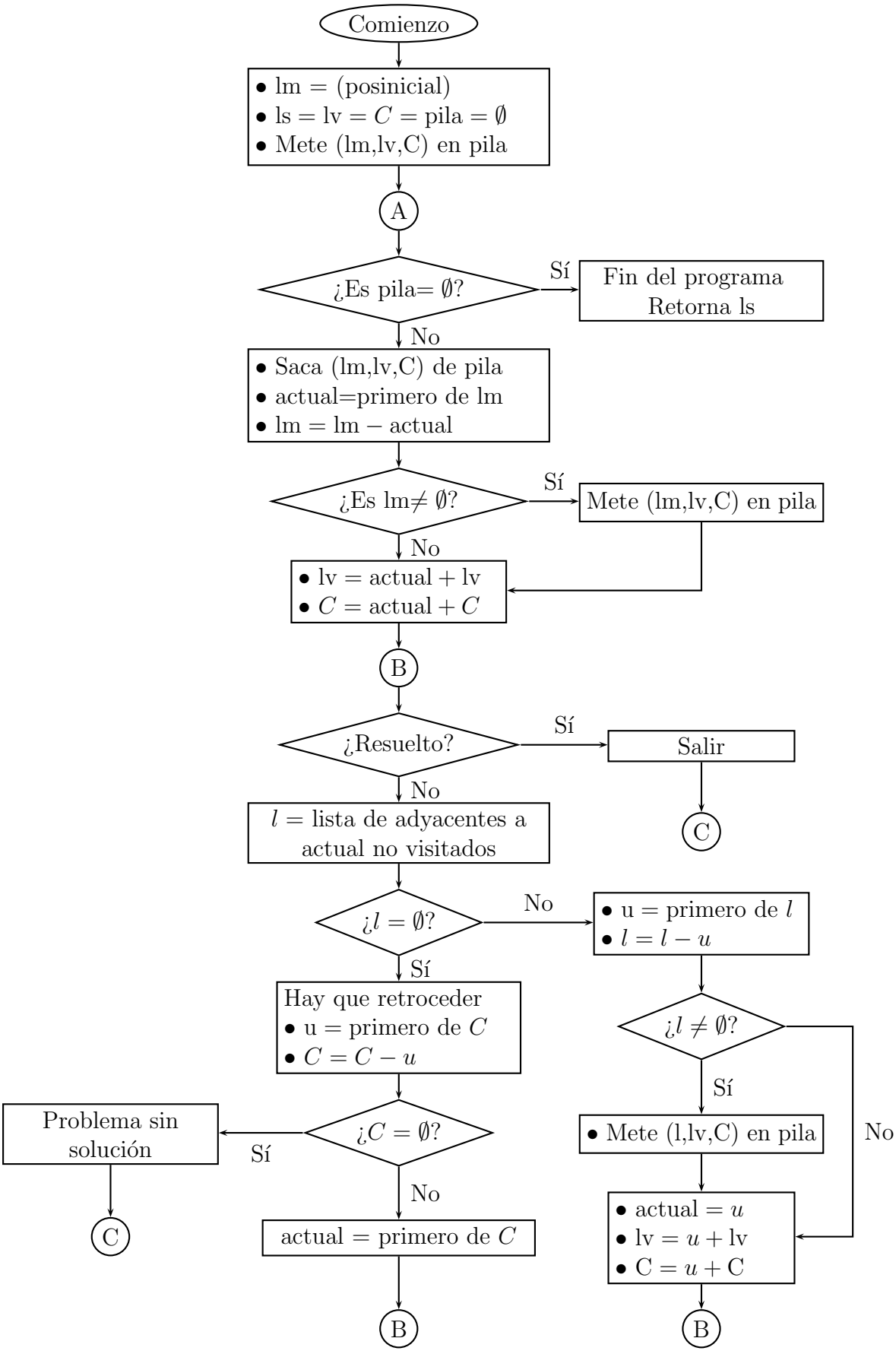
```
[7]> (tsolab lab02 '(4 2))
```

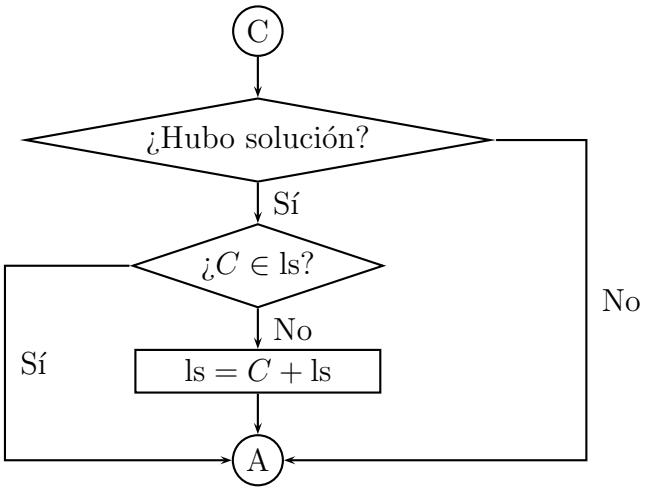
y el resultado:



5.1. Diagrama de flujo

El diagrama de flujo del algoritmo para la función `todas-las-soluciones` es el siguiente:


Comienzo
• lm = (posinicial)
• ls = lv = C = pila = $\emptyset$
• Mete (lm,lv,C) en pila
A
¿Es pila= $\emptyset$?
Sí
Fin del programa
Retorna ls
No
• Saca (lm,lv,C) de pila
• actual=primero de lm
• lm = lm − actual
¿Es lm≠ $\emptyset$?
Sí
Mete (lm,lv,C) en pila
No
• lv = actual + lv
• C = actual + C
B
¿Resuelto?
Sí
Salir
C
No
l = lista de adyacentes a actual no visitados
¿$l = \emptyset$?
No
• u = primero de l
• $l = l - u$
Sí
Hay que retroceder
• u = primero de C
• $C = C - u$
¿$l \neq \emptyset$?
Sí
No
Problema sin solución
¿$C = \emptyset$?
Sí
• Mete (l,lv,C) en pila
No
• actual = u
• lv = u + lv
• C = u + C
C
actual = primero de C
B
B



Las variables `actual`, `lv` y `C` son idénticas a las ya explicadas. La nueva variable `lm` (lista de movimientos) es una lista del tipo:

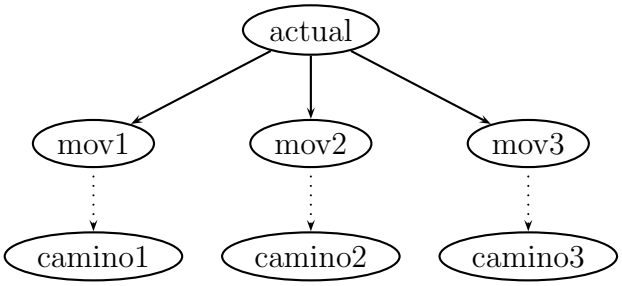
$$\text{lm} = \big((x_0\ y_0)\ (x_1\ y_1)\ \ldots\ (x_r\ y_r)\big)$$

donde cada $(x_i\ y_i)$ es una casilla en blanco no visitada y a la que hay que volver. Cuando hay éxito, la función retorna una lista `ls` (lista de soluciones), de la forma:

$$\text{ls} = (s_0\ s_1\ \ldots\ s_j)$$

siendo cada s_i una solución del laberinto.

Queda una pega por comentar y que sería necesario mejorar, en concreto: el algoritmo puede calcular varias veces el mismo camino. Supongamos que llegamos a un cruce, es decir, imaginemos la siguiente situación (estamos situados en `actual`):



Tal y como he explicado, la función elige `mov1` y pone en la pila lm = (`mov2` `mov3`). Si `mov1` conduce a un callejón sin salida, retrocederemos hasta `actual` y elegiremos `mov2`, etc. Como posteriormente, sacamos de la pila lm = (`mov2` `mov3`) y elegimos `mov2`, el camino que sigue a `mov2` es calculado dos veces, lo cual es un gasto de cálculo innecesario que debería evitarse.

El programa lo soluciona de la siguiente forma: una vez encontrada una solución, mira a ver si ya está en `ls`. Si lo está, no hace nada (para evitar la repetición), pero si no lo está, la incorpora y listo. De esta forma se evitan las repeticiones de soluciones, aunque no la de recorridos iguales.

5.2. Recorrido mínimo

Una vez que sabemos cómo obtener todas las soluciones, es muy simple encontrar la(s) de recorrido mínimo. En efecto, calculamos la longitud de cada una de las soluciones y nos quedamos con la mínima. Acto seguido, seleccionamos todas aquellas cuya longitud es la mínima, y listo.

La función `muestra-minimos` hace eso. De todas las salidas posibles, muestra en pantalla las de recorrido mínimo. Veamos algunos ejemplos (no se muestran los resultados):

```
(tsolab lab04 '(1 0))
(muestra-minimos lab04 '(1 0))
(tsolab lab04 '(6 8))
(muestra-minimos lab04 '(6 8))
(tsolab lab05 '(2 1))
(muestra-minimos lab05 '(2 1))
(tsolab lab05 '(2 5))
(muestra-minimos lab05 '(2 5))
```