Inteligencia Artificial

Taller I

Marzo 16 del 2015

Carlos Fernando Motta

En este taller debíamos diseñar e implementar un programa en lisp que resuelva un juego de sudoku haciendo dominio-consistencia, arco-consistencia y backtracking.

A continuación se explica en detalle el diseño de la solución propuesta por mi, junto con las restricciones del diseño actual. Adicionalmente, se presentan algunas pruebas realizadas al programa y finalmente se listan las conclusiones del taller.

Diseño de la solución

Con el fin de solucionar el problema, se debía lograr hacer que el programa recorriera una matriz de un sudoku(9x9 en este caso) desde la primera posición en la esquina superior izquierda del tablero hasta la última en la esquina inferior derecha, con el fin de poder ir analizando cada casilla por la que pasaba para saber que numero introducir de acuerdo al dominio posible, validando que el número introducido fuera correcto de acuerdo a las restricciones del juego y eliminando los que no fueran correctos.

Para nuestro caso en particular, se debía trabajar un backtracking utilizando dominio-consistencia y arco-consistencia. Así bien, para validar que el algoritmo tuviera dominio consistencia no se debió hacer muchos procedimientos, pues el dominio se sabe de antemano que son todos los números dentro del rango de 1 a 9, los cuales no van a variar y por lo tanto todos son correctos dentro del dominio posible. Para el caso de arco-consistencia ya no era tan simple, se debía escoger algún método para validar los arcos posibles. Ahora bien, como se vio en clase, debíamos trabajar este problema como un CSP, donde se debían considerar variables, dominios y restricciones. En el caso del sudoku era bastante visible, pues las variables identifican las casillas, el dominio son los números del 1 al 9 y la restricciones son tres, que estén todos los números del 1 al 9 en la misma fila, columna y en el cuadro al que pertenece el numero que se esta mirando.

Así bien, una vez definido esto, es posible entonces empezar a plantear una solución en Lisp. Se comienza entonces escogiendo una representación del tablero para saber como ir tomando cada valor, sin embargo, se nos facilitó una posible representación del tablero, el cual fue subido en moodle y seguía la siguiente estructura:

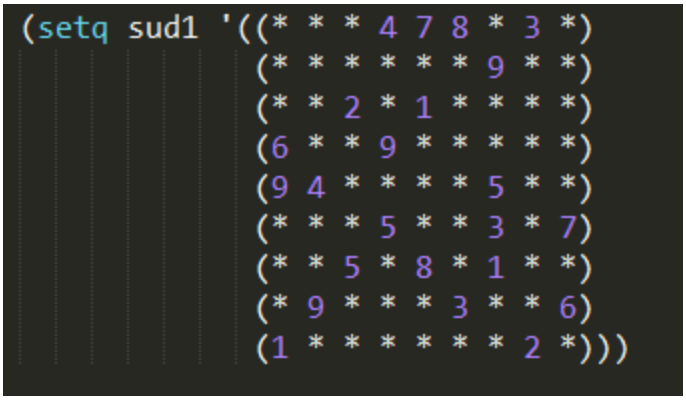

Figura 1: Representación de un tablero de sudoku en Lisp

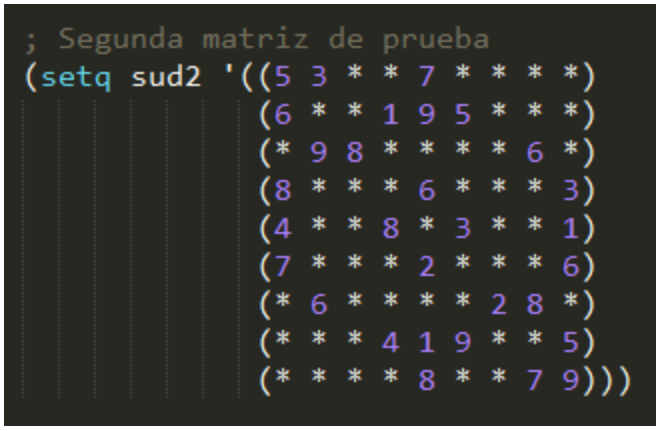

Figura 2: Representación de un tablero de sudoku en Lisp

Esta representación en particular consideraba cada fila como lista y en cada lista se indican los números por defecto del tablero y se marcaban los espacios vacíos con un asterisco.

Así bien, con esta representación lo primero que hice fue hacer funciones auxiliares que me permitiera moverme por todo el tablero con facilidad. Estas funciones me daban información de donde estaba parado en cada momento, en que fila estaba, que elementos estaban en toda la columna en la que estaba parado, cuales eran los elementos del cuadro donde estaba parado, cual era el inicio del cuadro en el que estaba, etc. Todas estas funciones auxiliares las hice de manera tal que cada una cumpliera con su rol y nada más, es decir, se hicieron lo más modulares posibles y se pueden ver en el archivo donde esta todo el taller adjunto a este documento llamado *Sudoku.lisp*.

Una vez definidas estas funciones, procedi a realizar una solución del problema guiandome con un algoritmo de backtracking que hice en C++, sin embargo la profesora Gloria me indico que no debía trabajar de manera iterativa en un programa funcional como Lisp. Por lo tanto, tuve que diseñar una solución recursiva, pero antes tenía que identificar cómo iba a

hacer que se cumpliera con el principio de arco-consistencia. Por lo cual, hice una función que dada la matriz del sudoku, me devolviera una matriz con listas de todos los posibles valores que podía tomar una casilla dadas las restricciones del juego, tal y como se muestra en la figura 3 para la matriz mostrada en la figura 1.

Figura 3: Matriz de arco-consistencia

Así bien, una vez obtenida la matriz de arco-consistencia se procedió a hacer el backtracking. Para esto se tomaron dos funciones, la función del backtracking y una función auxiliar. Estas funciones se muestran en la figura 4.

```
; Solucion de backtracking recursiva

(defun bt (sudoku arcos dominio k)
  (setq fila (floor k 9))
  (setq col (mod k 9))
  (cond
    ((null arcos) sudoku)
    ((equal (obtenerValor sudoku fila col 0 0) '*) (aux sudoku (car arcos) arcos dominio k))
    (t (bt sudoku (cdr arcos) dominio (+ k 1)))
  )
)

(defun aux (sudoku arco arcos dominio k)
  (setq fila (floor k 9))
  (setq col (mod k 9))
  (cond
    ((null arco) nil)
    ((esValido (insertarValor sudoku fila col 0 0 (car arco)) k fila col 0 0)
      (or (bt (insertarValor sudoku fila col 0 0 (car arco)) (cdr arcos) dominio (+ k 1))
          (aux sudoku (cdr arco) arcos dominio k)
      )
    )
    (t (aux sudoku (cdr arco) arcos dominio k))
  )
)
```

Figura 4: Backtracking recursivo implementado en Lisp

Como se puede ver, la función bt permite ir recorriendo el árbol en anchura, mientras que la función auxiliar permite ir recorriendo el árbol en profundidad. La variable de entrada arcos identifica la matriz de arco-consistencia en la función bt, mientras que en la función auxiliar esta la variable arco y arcos, arcos es la misma que en bt, mientras que arco identifica los valores posibles para la casilla actual que se está revisando. El algoritmo entonces va recorriendo la matriz de arco-consistencia y evaluando en auxiliar para cada casilla si ese valor es correcto, si lo es, entonces sigue a la siguiente casilla y así hasta que la matriz de arcos está vacía y entonces el algoritmo procederá a retornar la matriz del sudoku con la solución encontrada. En el caso en que el algoritmo se meta en una rama que no es, retorna nil en auxiliar y la función OR permite que se devuelva en el árbol de recursión y siga intentando posibles valores.

Pruebas realizadas

Con el fin de realizar pruebas, se tomaron ejemplos de internet; de google para ser más específico. Estos se pusieron como pruebas y luego se procedió a correr el algoritmo. Para los casos en los que la matriz estaba mal hecha, es decir, donde los valores predeterminados eran incorrectos, el algoritmo arrojaba **nil**, como se muestra en la figura 5.

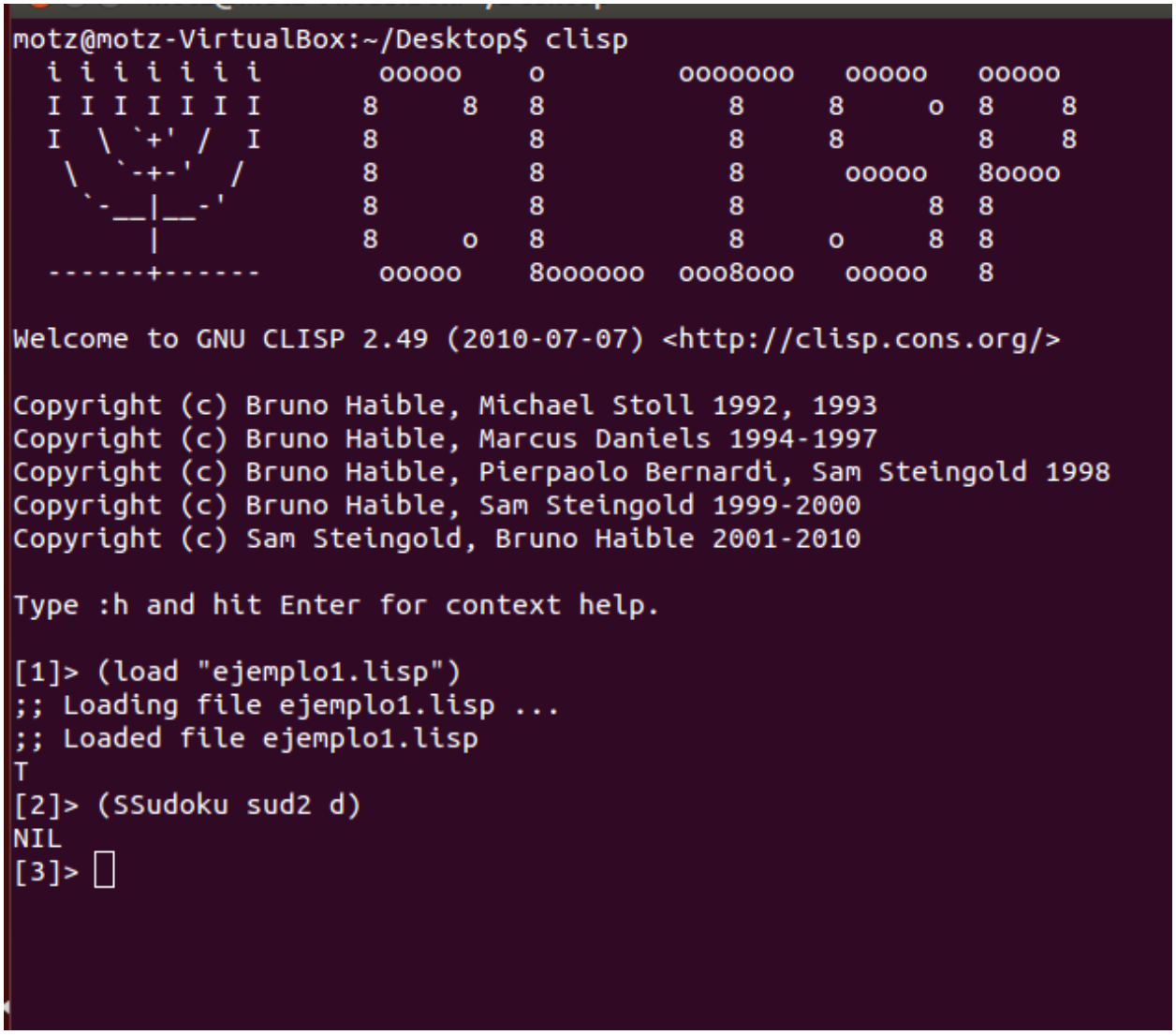

Figura 5: Representación errónea del sudoku

Y para los demás arrojaba una solución posible, donde dependiendo de la complejidad del sudoku se podría demorar más tiempo en arrojar el resultado. La figura 6 y 7 se muestra la solución encontrada para las matrices mostradas en la figura 1 y 2.

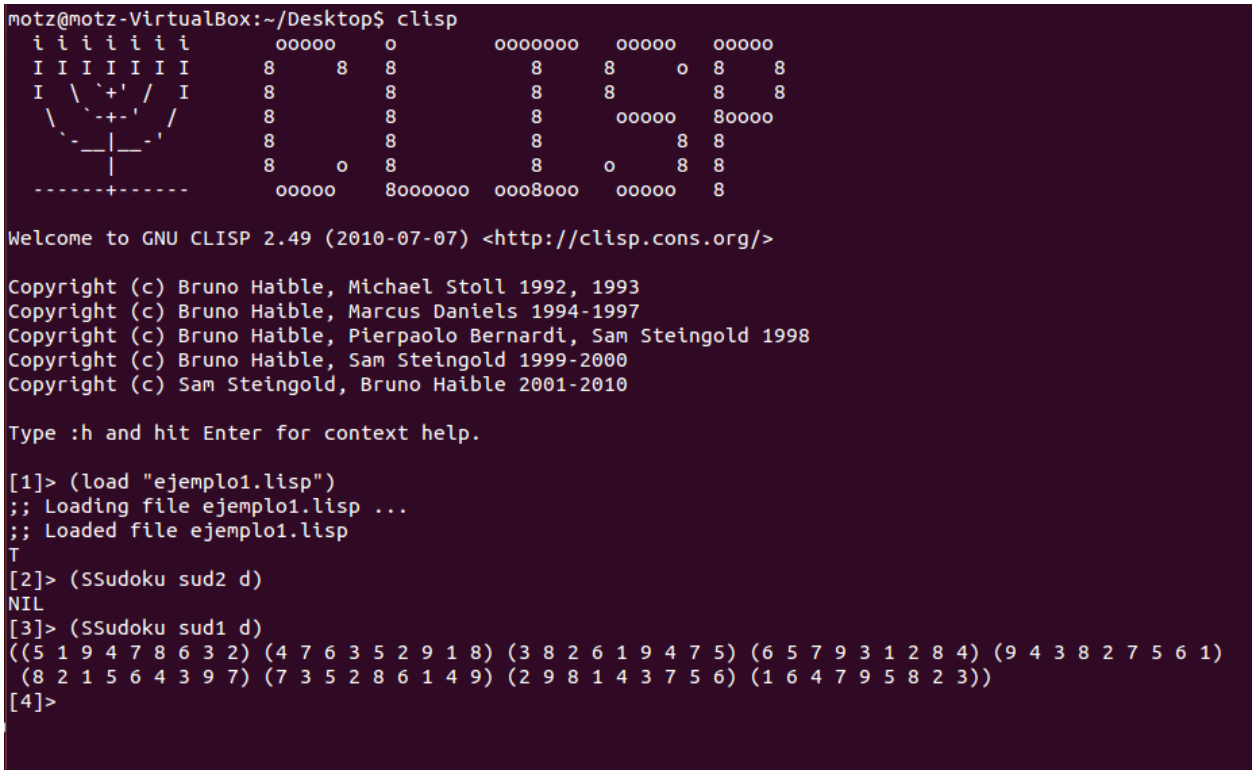

Figura 6: Solución del algoritmo dada una matriz y el dominio

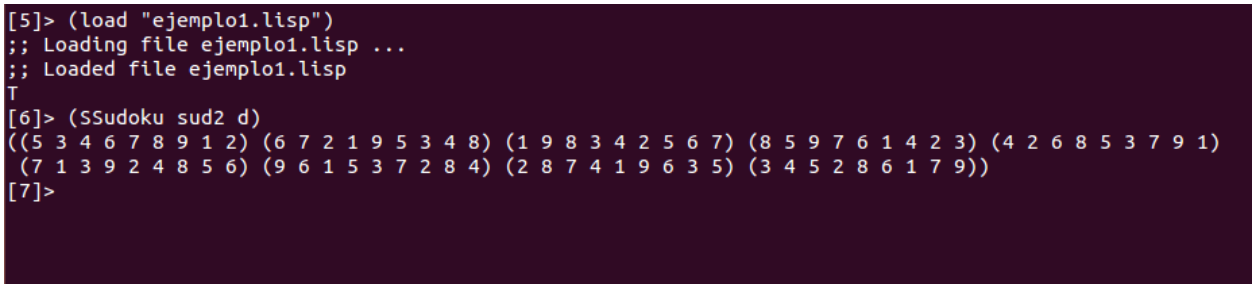

Figura 7: Solución del algoritmo dada una matriz y el dominio

Conclusiones

- Los lenguajes funcionales poseen una estructura y unas funciones muy expresivas, las cuales facilitan procesos que en el paradigma de programación imperativa hay que hacer un mayor esfuerzo por expresar lo mismo.
- Pasar la programación imperativa a la programación funcional es un gran salto, no es fácil adaptarse y por lo tanto entender conceptos y programar puede llevar más tiempo.
- La teoría vista en clase de los problemas CSP se pudo ver aplicada en el desarrollo del taller, lo cual permitió aterrizar aún más el concepto.

- A pesar de ser un ejercicio simple, me llevo mas de lo que esperaba para lograr que mis ideas al pasarlas a Lisp funcionaran como era debido.
- Todavía no estoy en la capacidad de pensar como hacer todo totalmente recursivo y por ende parte del código sigue teniendo cierta influencia de la programación imperativa.