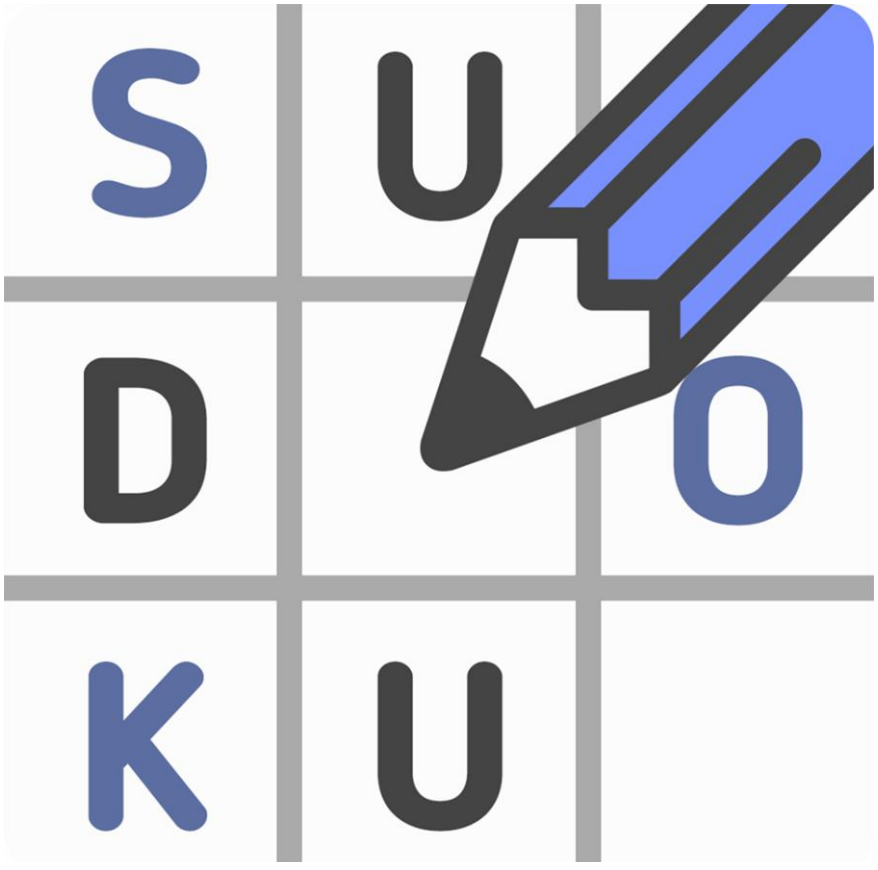
EL ROMPECABEZAS DE SUDOKU

INTELIGENCIA ARTIFICIAL



David Marquez Mínguez – 47319570Z

INTRODUCCIÓN

Este ejercicio consiste en crear un programa que empleando el lenguaje de programación racket, que sea capaz de resolver sudokus mediante dos algoritmos de búsqueda, la búsqueda en profundidad, y la búsqueda en anchura.

Las reglas del juego son las que ya conocemos, debemos rellenar un tablero de 9x9 fichas con dígitos desde el 1 hasta el 9 de forma que estos dígitos no se repitan ni en la filas, ni en las columnas, ni en las subcuadrículas.

	6		1		4		5	
		8	3		5	6		
2								1
8			4		7			6
		6				3		
7			9		1			4
5								2
		7	2		6	9		
	4		5		8		7	

Matriz inicial

9	6	3	1	7	4	2	5	8
1	7	8	3	2	5	6	4	9
2	5	4	6	8	9	7	3	1
8	2	1	4	3	7	5	9	6
4	9	6	8	5	2	3	1	7
7	3	5	9	6	1	8	2	4
5	8	9	7	1	3	4	6	2
3	1	7	2	4	6	9	8	5
6	4	2	5	9	8	1	7	3

Solución

En cuanto al proyecto, este se divide en dos archivos rkt donde se presenta la solución del ejercicio. Uno de los archivos(sudoku.rkt) presenta la estructura principal del problema donde se incluyen funciones esenciales para la resolución del problema, el otro archivo(funciones-aux.rkt) son funciones auxiliares que se encargarán de que la solución del problema sea lo mas clara posible, incluye funciones para obtener y modificar filas, columnas y diagonales del tablero.

Por otro lado, además en el proyecto se incluye una carpeta nombrada como 'test' donde se incluyen algunos sudokus de prueba para poder comprobar que el algoritmo funciona correctamente. En este mismo directorio se incluyen algunos sudokus que son imposibles de resolver para comprobar que el algoritmo se comporta correctamente ante este tipo de problemas.

```
1   ; test1
2   ;+-------+-------+-------+
3   ;| 5 0 0 | 0 0 0 | 0 0 0 |
4   ;| 0 2 8 | 4 0 0 | 5 0 3 |
5   ;| 1 0 0 | 2 7 0 | 0 0 6 |
6   ;+-------+-------+-------+
7   ;| 0 0 3 | 0 5 2 | 1 9 0 |
8   ;| 7 0 6 | 0 1 0 | 2 0 8 |
9   ;| 0 1 9 | 7 4 0 | 3 0 0 |
10  ;+-------+-------+-------+
11  ;| 6 0 0 | 0 9 4 | 0 0 2 |
12  ;| 8 0 1 | 0 0 6 | 7 5 0 |
13  ;| 0 0 0 | 0 0 0 | 0 0 4 |
14  ;+-------+-------+-------+
```

El programa se encarga de leer el txt necesario y pasarlo a la estructura de datos necesaria para su resolución. Si lo deseamos podemos añadir mas tipos distintos de sudokus creando un archivo txt y formar un sudoku respetando la estructura del resto para que la lectura sea correcta. La estructura de los archivos txt para que el programa los lea es la que se muestra en la imagen.

En cuanto a las funciones que se encargan de leer de forma correcta un sudoku a partir de un archivo txt son las siguientes:

- delete-lists: esta función se encarga de eliminar las líneas del archivo txt que no me interesan, es decir, todas aquellas líneas que no tengan dígitos.
- limpiar-lista: una dispongo de las líneas del txt que me interesan, debo limpiarlas, es decir, eliminar los símbolos que hacen que la lectura no sea limpia. Una vez aplicada esta función convierto el archivo txt en una matriz compuesta únicamente por los dígitos deseados.

ANALISIS DEL CODIGO

En cuanto al código, cuando se ejecuta el programa aparece un menú inicial donde podemos interactuar con él e indicar el sudoku a resolver y el algoritmo de resolución. Antes de empezar a resolver el sudoku me encargo de que este sudoku sea válido, es decir, de que el sudoku no tenga erratas, algún numero repetido en columnas, algún digito que no esté disponible…

A continuación se van generando sudokus validos que de forma que dependiendo del algoritmo se generaran de una forma u otra.

La forma en la que se generan los sudokus es de la siguiente manera. En primer lugar se van determinando las posiciones válidas para cada posición y dependiendo del algoritmo la resolución se realiza de una forma u otra.

45	4578	3	49	2	147	6	5789	57
9	24678	47	3	47	5	78	278	1
25	257	1	8	79	6	4	23579	2357
345	345	8	1	3456	2	9	34567	34567
7	123459	49	459	34569	4	1	13456	8
1345	13459	6	7	3459	8	2	1345	345
134	1347	2	6	478	9	5	1478	47
8	1467	47	2	457	3	17	1467	9
46	4679	5	4	1	47	3	24678	2467

Si resolvemos el sudoku mediante búsqueda en profundidad el algoritmo colocaría en una determinada posición del tablero un número valido e iría generando sus sucesores a partir del mismo. En este caso en la posición (0, 0) colocaría un 4 o un 5 e iría desarrollando el resto del sudoku de la misma forma. Su llegamos a un nodo hoja donde el árbol no puede seguir desarrollándose, se exploran otras opciones. Por otro lado en la búsqueda en anchura en vez de desarrollar la misma rama del árbol continuamente, generaríamos todas las posibles

opciones de una posición y a partir de ahí iríamos generando el resto de posibles sudokus a partir del nodo anterior.

Las funciones encargadas de comprobar si un digito es válido en una determinada posición del tablero son las siguientes:

- operación-valida?: Esta función simplemente se encarga de comprobar si la operación de insertar un elemento en una determinada posición del tablero es válida. Si existe algún número repetido en la fila, en la columna o en el cuadrante, la operación no será válida.
- get-lista-validos: A partir de la función anterior generamos una lista de validos para una posición. A partir de esta función vamos a generar los diferentes nodos del árbol.

En cuanto a las funciones auxiliarles como ya he dicho antes se encargan de obtener partes del tablero que necesite en cada momento. Estas funciones son las siguientes:

- get-columna: devuelve una lista con la columna del tablero determinada
- get-elem: devuelve cualquier elemento del tablero conociendo su fila y su columna
- get-cuadrante: devuelve el cuadrante completo en el que se encuentre una determinada ficha.

Finalmente se implementan las dos funciones básicas de este problema, la búsqueda en anchura y la búsqueda en profundidad.

Como hemos visto en teoría la búsqueda en anchura gestiona la lista de abierto como una lista de orden FIFO(First In First Out). La función implementada es la siguiente:

```
#|
sudoku, lista -> lista
OBJ: resolver sudoku mediante busqueda en anchura
PRE:
|#
(define (resolver-sudoku-bea sudoku abiertos)
  (cond
   [(sudoku-completo? sudoku) (imprimir-sudoku sudoku)]
   [else (resolver-sudoku-bea (last(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos))
                              (reverse(cdr (reverse(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos)
```

Como podemos observar se realizan llamadas recursivas a la función comprobando si algún sudoku es completo o no. En el momento en el que algún sudoku presenta una solución completa, el algoritmo se detiene y se imprime la solución. Como podemos observar la función presenta dos valores de entrada, lo que nosotros en los ejercicios denominamos ACTUAL, y lo que nosotros llamamos lista de ABIERTOS. Dichos valores se van modificando en cada iteración, en cuanto a ACTUAL como podemos observa se selecciona el ultimo sudoku generado, y en cuanto a la lista de abiertos funciona como una pila, el elemento de más arriba se elimina.

Por otro lado en la búsqueda en profundidad la lista de ABIERTOS se gestiona de distinta forma que en la búsqueda en anchura. En este caso no se gestiona como una lista FIFO sino como una lista de orden LIFO(Last In First Out), es decir, el ultimo en entrar es el primero en salir.

A continuación se muestra el fragmento de código que ejecuta la función de búsqueda en profundidad:

```
#|
sudoku, lista -> lista
OBJ: resolver sudoku mediante busqueda en profundidad
PRE:
|#
(define (resolver-sudoku-bep sudoku abiertos)
  (cond
   [(sudoku-completo? sudoku) (imprimir-sudoku sudoku)]
   [else (resolver-sudoku-bep (car(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos)
                              (cdr(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos)
```

Los elementos de entrada son exactamente iguales que en la búsqueda en anchura, pero en este caso se gestionan de forma diferente. En este caso como podemos observar el sudoku a analizar no es el ultimo generado sino el primero, al igual que la lista de ABIERTOS donde se elimina el primer sudoku generado, es decir, el sudoku con el que estamos trabajando.

FUNCIONAMIENTO DEL PROGRAMA

A continuación voy a ejecutar el programa para comprobar el correcto funcionamiento del mismo.

En cuanto ejecuto el programa aparece un menú, donde se puede seleccionar el tipo de algoritmo para resolver el sudoku.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket, with debugging; memory limit: 128 MB.
--------- BIENVENIDO AL SUDOKU SOLVER ----------
Mediante que algoritmo desea resolver el sudoku BEA o BEP
```

Una vez se ha seleccionado el tipo de algoritmo debemos selecciona el sudoku a resolver, hay una serie de sudokus a escoger pero si queremos probar nuestro propia sudoku como ya he dicho antes es posible hacerlo creando un txt respetando la estructura de los sudokus ya existentes.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket, with debugging; memory limit: 128 MB.
--------- BIENVENIDO AL SUDOKU SOLVER ----------
Mediante que algoritmo desea resolver el sudoku BEA o BEP
BEA
Inidque el sudoku que desea resolver (sudoku1, sudoku2, ...,
sudoku20)
```

Los sudokus que aparecen en la carpeta del proyecto son los sudokus proporcionados en la plataforma virtual de la asignatura para comprobar el correcto funcionamiento del código.

Una vez se selecciona el sudoku que queremos resolver, se imprime por pantalla el mismo y la ejecución del programa termina.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket, with debugging; memory limit: 128 MB.
--------- BIENVENIDO AL SUDOKU SOLVER ----------
Mediante que algoritmo desea resolver el sudoku BEA o BEP
BEA
Inidque el sudoku que desea resolver (sudoku1, sudoku2, ..., sudoku20)sudoku2

 -------SUDOKU RESUELTO-----
 5 6 3 7 8 4 1 2 9
 2 7 1 6 5 9 4 3 8
 8 9 4 1 2 3 5 6 7
 7 1 5 3 4 8 6 9 2
 9 8 6 2 7 5 3 4 1
 4 3 2 9 1 6 8 7 5
 6 5 8 4 9 7 2 1 3
 3 2 7 8 6 1 9 5 4
 1 4 9 5 3 2 7 8 6
>
```

Ahora voy a realizar el mismo procedimiento pero para resolver el sudoku mediante la búsqueda en profundidad.

```
Welcome to DrRacket, version 7.7 [3m].
Language: racket, with debugging; memory limit: 128 MB.
--------- BIENVENIDO AL SUDOKU SOLVER ----------
Mediante que algoritmo desea resolver el sudoku BEA o BEP
BEP
Inidque el sudoku que desea resolver (sudoku1, sudoku2, ..., sudoku20)sudoku3

 -------SUDOKU RESUELTO-----
 7 8 4 5 3 9 2 6 1
 1 6 5 4 7 2 8 3 9
 9 3 2 8 1 6 4 5 7
 4 5 3 2 6 7 9 1 8
 2 9 7 3 8 1 5 4 6
 6 1 8 9 5 4 7 2 3
 8 2 6 1 9 5 3 7 4
 5 7 9 6 4 3 1 8 2
 3 4 1 7 2 8 6 9 5
>
```

Obviamente si intentamos resolver el mismo sudoku con algoritmos diferentes, es posible que la solución no sea la misma. Esto es porque pueden que existan varias soluciones y cada algoritmo presenta aquella solución que encuentre lo antes posible.

CODIGO FUENTE

(funciones-aux.rkt)

```
#lang racket

(provide (all-defined-out))

#|
numero, numero -> numero
OBJ: devolver el valor de una posicion del sudoku
PRE: 0 < x-coor and y-coor < 9
|#
(define (get-elem x-coor y-coor sudoku)
  (list-ref(list-ref sudoku x-coor) y-coor))

#|
numero, numero, lista -> numero
OBJ: obtiene un numero de una lista dada
PRE: indice < len(lista)
|#
(define (get-num-columna indice lista)
  (cond
    [(equal? indice 0) (car lista)]
    [else (get-num-columna (- indice 1) (cdr lista))]))

#|
numero, numero, sudoku -> lista
OBJ: obtener una columna determinada del sudoku
PRE: columna < len(lista)
|#
(define (get-columna columna sudoku)
```

```
 (cond
   [(empty? sudoku) '()]
   [else (cons (get-num-columna columna (car sudoku)) (get-columna columna (cdr sudoku)))]))



#|
sudoku, numero, numero -> lista
OBJ: devolver las filas a partir de un numero
PRE: fila and columna < 9
|#
(define(get-filas sudoku fila columna)
 (cond
   [(< columna fila) (cons (list-ref sudoku columna) (get-filas sudoku (+ columna 1) fila) )]))


#|
sudoku, numero, numero -> lista
OBJ: obtener un cuadrante del sudoku
PRE: num-fila y num-columna deben ser 0 o 1 o 2
|#
(define (get-cuadrante sudoku num-fila num-columna)
 (list (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (* num-fila 3))) (* num-columna 3))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (* num-fila 3))) (+ (* num-columna 3) 1))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (* num-fila 3))) (+ (* num-columna 3) 2))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 1))) (* num-columna 3))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 1))) (+ (* num-columna 3)
1))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 1))) (+ (* num-columna 3)
2))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 2))) (* num-columna 3))
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 2))) (+ (* num-columna 3)
1))
```

```
      (list-ref (car(get-filas sudoku (+ (* num-fila 3) 3) (+ (* num-fila 3) 2))) (+ (* num-columna 3) 2))))
```

(sudoku.rkt)

```
#lang racket

(require "funciones-aux.rkt")
(require 2htdp/batch-io)

#|
sudoku -> boolean
OBJ: determinar si el sudoku es valido
PRE: el formato de entrada debe ser el correcto
|#
(define (sudoku-valido? sudoku)
  (not (comprobar-sudoku sudoku)))



#|
sudoku -> boolean
OBJ: determinar si un cierto numero esta en el sudoku
PRE:
|#
(define (esta? sudoku num)
  (for*/first ([i 9]
               [j 9]
               #:when (equal? (get-elem i j sudoku) num))
    (list i j)))

#|
lista -> number
```

```
OBJ: determinar el tamanno de una lista
PRE:
|#
(define (len lst)
 (cond
   [(empty? lst)  0]
   [(cons? lst)   (+ 1 (length (rest lst)))]))

#|
numero, lista -> boolean
OBJ: determinar si un numero esta en una fila
PRE:
|#
(define (in-fila? numero fila)
 (for/first ([i fila]
       #:when (equal? i numero))
        true))

#|
numero, numero, lista -> numero
OBJ: contar cuantos veces aparece un cierto numero en una lista
PRE:
|#
(define (cuantos? num lista)
 (cond
   [(empty? lista) 0]
   [(equal? num (car lista)) (+ 1 (cuantos? num (cdr lista)))]
   [else (cuantos? num (cdr lista))]))

#|
numero, numero, lista -> boolean
```

```
OBJ: determinar si un elemento esta repetido en una lista
PRE:
|#
(define (num-repetido? num lista)
  (cond
    [(> (cuantos? num lista) 1) true]
    [else false]))

#|
lista -> boolean
OBJ: determinar la posicion de un numero en una lista a partir de su cuadrante
PRE: pos(i<9, j<9)
|#
(define (pos-cuadrante pos)
  (+ (* (floor (/ (car pos) 3)) 3)  (floor (/ (car (cdr pos)) 3))))

#|
numero, numero, lista -> lista
OBJ: colocar en una determinada posicion de la lista un numero
PRE: posicion < len(lista)
|#
(define (colocar-num numero posicion lista)
  (cond
    [(equal? 0 posicion) (cons numero (cdr lista))]
    [else (cons (car lista)(colocar-num numero (- posicion 1) (cdr lista)))]))

#|
lista, numero, sudoku -> sudoku
OBJ: reemplazar una lista por otra en el sudoku
PRE: len(lista) == 9 and 0 < pos < 9
|#
```

```
(define (colocar-list lista pos sudoku)
 (cond
  [(equal? pos 0) (cons lista (cdr sudoku))]
  [else (cons (car sudoku) (colocar-list lista (- pos 1) (cdr sudoku)))]))


#|
sudoku -> nil
OBJ: imprimir el sudoku por pantalla
PRE:
|#
(define (imprimir-sudoku sudoku)
 (for*
    [(i 9)(j 9)]
   (cond
    [(equal? j 8) (display " ")(write (get-elem i j sudoku)) (display "\n")]
    [else (display " ")(write (get-elem i j sudoku))]
    )))

#|
sudoku -> boolean
OBJ: determinar si el sudoku esta completo, es decir, no hay ningun cero
PRE:
|#
(define (sudoku-completo? sudoku)
 (cond
   [(equal? (esta? sudoku 0) false) true]
   [else false]))

#|
sudoku -> boolean
```

```
OBJ: recorrer el sudoku determinando su validez

PRE: el formato de entrada debe ser el correcto

|#

(define (comprobar-sudoku sudoku)

 (for*/first ([i 9]

        [j 9]

        #:when [or (< 9 (get-elem i j sudoku))

              (> 0 (get-elem i j sudoku))

              (and(not(= 0 (get-elem i j sudoku)))

                 (or (num-repetido? (get-elem i j sudoku) (list-ref sudoku i)) ;compruebo si hay algun elemento repetido en las filas

                    (num-repetido? (get-elem i j sudoku) (get-columna j sudoku)) ;compruebo si hay algun elemento repetido en las columnas

                    (num-repetido? (get-elem i j sudoku) (get-cuadrante sudoku (floor (/ i 3)) (floor (/ j 3)))) ;compruebo si hay algun elemento repetido en los cuadrantes

                    ))]) true))


#|

sudoku, numero, lista -> boolean

OBJ: determinar si una insercion de un numero es valida

PRE: posicion(i<9, j<9)

|#

(define (operacion-valida? sudoku num posicion)

 (cond

  [(or (num-repetido? num (colocar-num num (car (cdr posicion))(list-ref sudoku (car posicion))))

     (num-repetido? num (colocar-num num (car posicion) (get-columna (car (cdr posicion)) sudoku)))

     (num-repetido? num (colocar-num num (pos-cuadrante posicion) (get-cuadrante sudoku (floor (/ (car posicion) 3)) (floor (/ (car (cdr posicion)) 3)))))

     ) false]

  [else true]))
```

```
#|
sudoku, numero, lista, numero -> lista
OBJ: determinar los numeros validos para una posicion
PRE: posicion(i<9, j<9) and iteracion = 9
|#
(define (get-lista-validos sudoku posicion iteracion)
  (cond
    [(equal? iteracion 0) '()]
    [(operacion-valida? sudoku iteracion posicion) (reverse (cons iteracion (get-lista-validos
sudoku posicion (- iteracion 1))))]
    [(get-lista-validos sudoku posicion (- iteracion 1))]))

#|
lista, lista, sudoku, pila -> pila
OBJ: generar sudokus validos
PRE:
|#
(define (generar-sudokus lista-validos pos sudoku pila)
  (cond
    [(empty? lista-validos) pila]
    [else (generar-sudokus (cdr lista-validos) pos sudoku (cons (colocar-list (colocar-num (car
lista-validos) (car (cdr pos)) (list-ref sudoku (car pos))) (car pos) sudoku) pila))]))

#|
sudoku, lista -> lista
OBJ: resolver sudoku mediante busqueda en profundidad
PRE:
|#
(define (resolver-sudoku-bep sudoku abiertos)
  (cond
   [(sudoku-completo? sudoku) (imprimir-sudoku sudoku)]
```

```
  [else (resolver-sudoku-bep (car(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos))
                             (cdr(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos)))]))


#|
sudoku, lista -> lista
OBJ: resolver sudoku mediante busqueda en anchura
PRE:
|#
(define (resolver-sudoku-bea sudoku abiertos)
 (cond
  [(sudoku-completo? sudoku) (imprimir-sudoku sudoku)]
  [else (resolver-sudoku-bea (last(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos))
                             (reverse(cdr (reverse(generar-sudokus(get-lista-validos sudoku (esta? sudoku 0) 9)(esta? sudoku 0) sudoku abiertos)))))]))


#|
lista -> lista
OBJ: eliminar las listas innecesarias
PRE: len lista > 9
|#
(define (delete-lists lista)
 (cond
   [(equal? (len lista) 9) lista]
   [(member ";|" (car lista)) (delete-lists (append (cdr lista) (list (car lista))))]
   [else (delete-lists (cdr lista))]))


#|
lista -> lista
OBJ: limpiar una lisra para su posterior lectura
```

```
PRE: len lista > 9
|#
(define (limpiar-lista lista)
 (cond
   [(equal? (len lista) 9) lista]
   [(or (equal? "|" (car lista))(equal?  ";|" (car lista))) (limpiar-lista (cdr lista))]
   [else (limpiar-lista (append (cdr lista) (list (string->number(car lista)))))]))

#|
lista, number -> lista
OBJ: limpiar un sudoku completo
PRE:
|#
(define (limpiar-sudoku lista rep)
 (cond
   [(equal? rep 0) lista]
   [else (limpiar-sudoku (append (cdr lista) (list (limpiar-lista (car lista)))) (- rep 1))]))

#|
nil -> nil
OBJ: interfaz de usuario
PRE:
|#
(define (iniciar-sudoku-solver)
 (display "--------- BIENVENIDO AL SUDOKU SOLVER ---------- \n")
 (display "Mediante que algoritmo desea resolver el sudoku BEA o BEP \n")
 (define metodo (read-line (current-input-port) 'any))
 (display "Inidque el sudoku que desea resolver (sudoku1, sudoku2, ..., sudoku20)")
 (define sudoku (read-line (current-input-port) 'any))
 (display "\n -------SUDOKU RESUELTO----- \n")
 (cond
```

```
   [(equal? (sudoku-valido? (limpiar-sudoku (delete-lists (read-words/line (string-append
"./test/" sudoku ".txt"))) 9)) false) (display "El sudoku no se puede resolver")]

  [(equal? metodo "BEA")(resolver-sudoku-bea (limpiar-sudoku (delete-lists (read-words/line
(string-append "./test/" sudoku ".txt"))) 9) '())]

  [(equal? metodo "BEP")(resolver-sudoku-bep (limpiar-sudoku (delete-lists (read-words/line
(string-append "./test/" sudoku ".txt"))) 9) '())]

  [else (display "El metodo seleccionado no existe, selecciona BEA o BEP")]))


;(limpiar-sudoku (delete-lists sudoku) 9)

;(resolver-sudoku-bep (limpiar-sudoku (delete-lists sudoku1) 9) '())

;(display "\n")

;(resolver-sudoku-bea test1 '())


(iniciar-sudoku-solver)
```