# MoJo Language Specification

## Overview

MoJo es un lenguaje diseñado para facilitar la transformación de tablas bidimensionales. El lenguaje parte de la gramática ASL proporcionada en el laboratorio y sigue conteniendo todas las instrucciones que ya habían (while, asignación, etc), MoJo extiende un poco más las funcionalidades del lenguaje con una sintaxis distinta.

El lenguaje tiene la mayoría de las operaciones por defecto especificadas como funciones pre implementadas, sólo para operaciones que requieran modificar o ejecutar una expresión de la gramática son definidas como instrucciones del lenguaje.

Las tablas se pueden crear dentro del lenguaje o leyendo un CSV con los contenidos de la tabla. Para cada tabla es obligatorio tener un nombre por columna de tipo String y cada elemento de la misma columna debe ser del mismo tipo. El tipo de la columna se define al agregar a la columna su primer elemento, agregar un elemento que no sea del tipo de la columna resultará en un error.

Existen varias formas de acceder o modificar los elementos de una tabla. Las instrucciones select, filter y update son muy parecidas a las de SQL y permiten un acceso por fila. Luego también está la notación matricial donde el primer elemento representa la fila y el segundo la columna. El identificador de una fila es un entero y el de una columna puede ser tanto un string (nombre de la columna) o un entero (el orden de las columnas no está garantizado).

Las instrucciones del lenguaje pueden estar separadas por puntos y comas o saltos de línea. Los comentarios se hacen con almohadillas (“#”) y los comentarios de varias líneas con almohadilla y guión (“#- ... -#”). La ejecución del programa siempre empieza por la función principal main, esta también puede tener argumentos en caso de que se quiera ejecutar desde otro script.

Los tipos soportados por el lenguaje son void, entero, booleano, string, lista y tabla. La lista puede ser de cualquier tamaño y contener múltiples tipos. Las tablas son únicamente bidimensionales y el tipo dentro de cada columna debe ser homogéneo.

## Ejemplos

Todas las funciones que aparezcan en los ejemplos están predefinidas dentro del lenguaje o estarán definidas dentro del mismo ejemplo. Las especificaciones de cada una están al final del documento, son muy fáciles de entender a simple vista por lo que se puede entender los ejemplos sin ningún problema sin saber nada a priori.

### Ejemplo 1

En este ejemplo se leen dos tablas de distintos archivos .csv, se aplica un merge entre ambos y luego un sort sobre la tabla resultante. De esta tabla se escogen 20 filas con probabilidad uniforme y se escribe el resultado en un archivo de texto.

```
function mergesort(table1,table2)   # Esto no es un mergesort de verdad
    tmp = merge(table1, table2)
    return sort(tmp)
end

function main()     # Se ejecuta por defecto
    x = read_file("samples/sample1.csv")
    y = read_file("samples/sample2.csv")
    tmp = mergesort(x,y)
    tmp = sample(8,tmp)
    write_file(tmp, "samples/example1.csv")
    return num_rows(tmp) = 8
end
```

### Ejemplo 2

Aquí se muestra como acceder a una tabla con expresión matricial. Para el acceso matricial es necesario aportar los dos identificadores de la casilla. También se puede acceder con un solo identificador pero devuelve un diccionario que representa la fila.

```
function main()
    tmp = read_file("samples/numbers.csv")
    i = 0
    j = 0
    nrows = num_rows(tmp)
    names = column_names(tmp)
    while (i < nrows)   # Acceso a la tabla por número de row y nombre de columna
        while (j < num_columns(tmp))
            tmp[i,names[j]] = i+j
            j = j + 1
        end
        i = i + 1
    end
```

```
end
```

### Ejemplo 3

Las operaciones especiales estilo SQL se usan mediante el token ‘from’. Para acceder a la casilla correspondiente a una columna dentro de una fila se usa el token ‘:’ seguido de una String, una variable o un número. Todas las operaciones especiales dentro de un ‘from’ son concatenadas con el resultado de la operación anterior. Por lo que aux1 sería equivalente a “`select (:age <= 5 and :1 = "MJ RULES")`”. Filter es simplemente el negado de el select, la condición “`(:var > 5  or :"bio" != "MJ RULES")`” es la negación de la condición de el select por lo que daran el mismo resultado.

```
function main()
    x = read_file("samples/sample3.csv")
    write "Table x%n"
    writeln x
    var = "age"
    aux1 = from x
        select (:var <= 5)
        select (:2 = "MJ RULES")
    end
    writeln "Table aux1"
    writeln aux1
    var = 1
    aux2 = from x filter (:var > 5  or :"bio" != "MJ RULES") end
    writeln "Table aux2"
    writeln aux2
    writeln "Table result"
    writeln (aux1 = aux2)
    return (aux1 = aux2)
end
```

El contenido de sample3.csv:

```
index, age, bio
0, 5, 'MJ RULES'
1, 5, 'MJ RULES'
2, 6, 'MJ RULES'
3, 5, 'MJ MAYBE RULES'
4, 6, 'MJ NOT RULES'
5, 5, 'MJ NOT RULES'
6, 6, 'MJ NOT RULES'
```

### Ejemplo 4

Aquí se muestra un poco un ejemplo de ejecución de otro script. Los parámetros en la lista de la función source son recogidos por la función main del script llamado.

```
function main()
	x = source("example1.mj");
	if (x)
		writeln "Your instruction functions work!"
	else
		writeln "Check your language implementation <.<"
	end
end
```

### Ejemplo 5

En este ejemplo se muestra la forma correcta de crear y extender tablas en el lenguaje. El operador de suma en la lista es la concatenación. También se puede ver el uso de la instrucción update, el parámetro when es opcional.

```
function main()
    column_names = ["name", "surname"] + ["age", "skill"] # Concatenation
    x = create_table(column_names)
    add_row(x, ["Jon", "Snow", 26, "Knowing nothing"])
    add_row(x, ["Sansa", "Stark", 21, "Inmortality"])
    add_column(x, ["lorey money"])
    add_row(x, ["Marc", "Ortiz", 22, "Making neural networks and playing basket", 1000000])
    bankrupt = true

    from x update "lorey money" when bankrupt with 0 end
    write x
end
```

### Ejemplo 6

Se puede inicializar directamente diccionarios en el lenguaje con corchetes. Las tablas se van creando mediante la concatenación vertical de diccionarios.

```
function main()
    columns = ["nom","edat","professió", "residència"]
    table = create_table(columns)
    row = {"nom":"Juan","edat":22,"professió":"Not yet"}
    row2 = {"nom":"Juan","edat":10,"professió":"Not yet"}
    add_row!(table,row,row2)
    table[0,"edat"] = 12
    table[0,"residència"] = "Caracas"
    write(table)
end
```

### Ejemplo 7

En el lenguaje se diferencian las listas anidadas de las matrices. A diferencia de las tablas la listas anidadas pueden tener más dimensiones.

```
function main()
    column_names = ["name", "surname"] + ["age", "skill"] # Concatenation
    Countries = ["Info",[["China","Planty"],["North
Korea","Dangerous"],["Spain","Awesome"]]]

    x = create_table(column_names) # El tipo de las columnas se defininira cuando se añada
un elemento
    add_row!(x, {"name":"Jon", "surname":"Snow", "age":26, "skill":"Knowing nothing"})

    j = 0
    write("Content of the table:  %n")

    while (j<num_columns(x))
        write(x[0,column_names[j]])
        write(", ")
        j = j + 1
    end
    write("%n")
    write("People from Spain are: ")
    writeln(Countries[1,2,1])

end
```

### Ejemplo 8

Hay tres modalidades de drop en una tabla, se parece mas a un clear que a una instrucción drop SQL. Si se pasa un numero como segundo parámetro se borra una fila, si se pasa una string entonces se borra una columna. En el caso de no haber parámetros adicionales se vacía la tabla dejando únicamente los labels.

```
function main()
    tmp = read_file("samples/numbers.csv")
    writeln tmp
    drop(tmp, 1)
    writeln tmp
    drop(tmp, "a")
    writeln tmp
    drop(tmp)
    writeln tmp
end
```

**Ejemplo 9**

También se puede usar el operador unario + para unir tablas y concatenar strings.

```
function main()
    tmp1 = read_file("samples/sample1.csv")
    tmp2 = read_file("samples/sample2.csv")
    writeln tmp1+tmp2

    writeln "Baby " + "Don't " + "Hurt " + "Me " + "No " + "More"
end
```

# Gramàtica del lenguaje

Nuestro lenguaje como Asl consta de una lista de funciones, todas ellas separadas entre sí por saltos de línea y una definición de parámetros de entrada.
El cuerpo de las funciones estará compuesto por una sèrie de instrucciones, que en nuestro caso serán las que ya vienen definidas en el lenguaje ASL más instrucciones diseñadas especialmente para el tratamiento con tablas. Además de nuevas instrucciones se ha expandido el lenguaje con nuevos átomos. Finalmente se ha dado la posibilidad de añadir comentarios en el lenguaje, los comentarios serán una cadena de caracteres escritos detrás del símbolo '#'.

## Instrucciones

**From:** *FROM expr from_instructions END*.

La instrucción se compone del token *'from'* seguido de una expresión *'expr'* que en el intérprete se comprobarà que sea una tabla y un conjunto de sub-instrucciones '*from_instructions'* separadas por saltos de línea o punto y coma, que se aplicarán a esa expresión.

El conjunto de sub-instrucciones '*from_instructions'* puede ser:

- **Select** *expr:* Selecciona las filas de la tabla que cumplan la condición *expr*.
- **Filter** *expr:* Selecciona las filas de la tabla que no cumplan la condición *expr.*
- **Update** *expr1 (WHEN! expr2)? WITH! expr3 :* Harà update de *expr1* si se cumple *expr2* y lo reemplazarà con *expr3*

Podemos ver a continuación un ejemplo de la estructura de la instrucción **from**.

```
richPeople = from people
        select (:age > 18 && :age < 65)
        filter (:health = “sick”)
        update (:wealth = 0)
end
```

## Átomos

Podemos ver a continuación nuevas reglas para el reconocimiento de átomos en el lenguaje MoJo.

**Atom**:

- Column
  - Language: ':' *name*
  - Reconoce a un conjunto de caracteres que empiezan por ‘:’ y seguido de INT, STRING o Var.
- List
  - Language: '[' expr_list? ']'
  - Reconoce a un conjunto de expresiones encapsuladas con ‘[‘ y ‘]’.
- Dict
  - Lenguaje: '{' dict_list? '}'
  - Reconoce a un conjunto de elementos de diccionario *dict_list* encapsulados con ‘{‘ y ‘}’.
    - dict_list
      - Lenguaje: STRING ':' *expr*

# Semántica del lenguaje

Una de las principales modificaciones en la semántica de Asl ha sido la definición e implementación de nuevos tipos de datos con los que poder trabajar.

# Tipos de datos

MoJo cuenta con 6  nuevos tipos de datos respecto a el lenguaje base ASL. Los nuevos tipos són: List, Dictionary, Table, String y Void.

## List

List és un contenedor de elementos, donde el tipo de los elementos en la lista es uniforme y puede ser de cualquier tipo. Podemos tener por ejemplo listas de Diccionarios, listas de listas o listas de tablas.

Operaciones soportadas sobre el tipo list:

- **Arithmetic operations**
  - **List1 + List2**: La operación suma en listas se traduce a la concatenación de dos listas.

- **Relational operations**
  - **List1 = List2**: Compara dos listas y si el contenido de las dos listas es el mismo devuelve un booleano con valor true.
  - **List1 != List2**: Compara dos listas y si el contenido es diferente devuelve un booleano con valor true.

- **Access operations**
  - **List[i]**: Accede al elemento i-ésimo de la lista. El parámetro ‘i’ tiene que ser de tipo Integer.

Ejemplo de variable tipo list:

```
# Inicialización
matrix= [[0,1], [0,1]] + [[0,1], [0,1]]

# Concatenation
column_names = ["name", "surname"] + ["age", "skill"]

# Acceso al primer elemento
write(column_names[0])

#Comparación
List2 = ["name", "surname"]
Equals = (column_names = List2) #Equals = false
```

## Dictionary

El tipo diccionario es un contenedor de elementos donde los elementos son un conjunto (clave,valor), donde la clave es de tipo String y el valor puede ser de cualquier tipo.

Operaciones soportadas sobre el tipo Dictionary:

- **Arithmetic operations**
  - **Dictionary1+ Dictionary2**: La operación suma en Dictionary se traduce a la unión de dos diccionarios, si existen conjuntos con misma clave en Dictionary1 y Dictionary2, se escogen los de Dictionary2.

- **Relational operations**
  - **Dictionary1 = Dictionary2**: Compara dos diccionarios y si el contenido de los dos diccionarios es el mismo devuelve un booleano con valor true.
  - **Dictionary1 != Dictionary2**:Compara dos diccionarios y si el contenido de los dos diccionarios es diferente devuelve un booleano con valor true.

- **Access operations**
  - **Dictionary[key]**: Devuelve el valor asociado a la clave *'key'*. Si no existe tal valor, se crea en Dictionary un conjunto (key,elem), dónde *'elem'* es de tipo Void. El parámetro key tiene que ser de tipo String.

Ejemplo de variable tipo Dictionary:

```
# Inicialización
Row = {“name” : “Marc”, “age” : 22, “residence” : “Olot”}

# Concatenation
Union_set = {“id1” : “Marc”, “id2” : “Juan”} + {“id3” : “Pepe”, “id1” : “MoJo”}

# Acceso al primer elemento
write(Union_set[“id1”])  # Should be “MoJo”

#Comparación
Row2 = {“name” : “Marc”, “age” : 22, “residence” : “Olot”}
Equals = (Row = Row2) #Equals = true
```

## Table

Table es el tipo de datos principal del programa y en el que giran gran parte de las instrucciones y funciones predefinidas.

El tipo de datos Table se compone de una lista de diccionarios. Cada elemento de la lista, es decir cada diccionario conceptualiza una fila o row en la tabla, y cada elemento del diccionario de la forma (key,val) contiene el valor para la columna 'key' de la tabla.

Al crear una tabla vacía podemos especificar la forma de la tabla, es decir, qué columnas tendrá la tabla. Si no especificamos las columnas, las tendremos que añadir posteriormente antes de insertar ninguna fila.

Cada columna del tipo tabla tiene asignado un tipo y se representan con el tipo String, este tipo nos sirve para assegurar que para todas las filas de la tabla, el valor de esa columna siempre será del mismo tipo.

Ejemplo:

```
#Las columnas las representamos como Strings
column_names =[ "name", "age", "residence"]
Table1 = create_table(column_names)

#Añadimos la primera row, el tipo de "age" passarà a ser Integer para siempre.
add_row!(Table1, {"name" : "Marc", "age" : 22, "residence" : "Olot"} )

#Error, la columna age es de tipo Integer
add_row!(Table1, {"name" : "Fool", "age" : "patata"}
```

Operaciones soportadas sobre el tipo Table:

- **Arithmetic operations**
  - **Table1 + Table2**: Al sumar dos tablas se hace un merge de las tablas.

- **Relational operations**
  - **Table1 = Table2**: Compara dos tablas y si el contenido de las dos tablas es el mismo devuelve un booleano con valor true.
  - **Table1 != Table2**:Compara dos tablas y si el contenido de las dos tablas es diferente devuelve un booleano con valor true.

- **Access operations**
  - **Table[index]**: Devuelve la fila de la tabla que se encuentra en la posición index. La fila como se ha citado anteriormente es de tipo Dictionary. El parámetro index es de tipo Integer.
  - **Table[index, column]**: Devuelve el valor de la columna 'column' de la fila en la posición 'index'. El tipo devuelto puede ser cualquiera. El parámetro index tiene que ser de tipo Integer y column tiene que ser de tipo String.

Ejemplo de operaciones sobre tablas:

```
#Las columnas las representamos como Strings
```

```
column_names =[ “name”, “age”, “residence”]
Table1 = create_table(column_names)
Table2 = create_table(column_names)

#Añadimos la primera row, el tipo de “age” passarà a ser Integer para siempre.
add_row!(Table1, {“name” : “Marc”, “age” : 22, “residence” : “Olot”} )
add_row!(Table2, {“name” : “Pepe”, “age” : 80, “residence” : “Olot”} )

#Merge
TableMerged = Table1 + Table2

#Print row : {“name” : “Marc”, “age” : 22, “residence” : “Olot”}
write(TableMerged[1])

#Print value : “Pepe”
write(TableMerged[1,”name”])
```

## Void

Simula el elemento vacío en el lenguaje. No permite operaciones sobre este tipo y se usa básicamente cuando en el acceso a un tipo contenedor cómo Dictionary no existe el elemento, se devuelve void.

## String

El tipo string nos permite representar y trabajar con cadenas de caracteres en MoJo.

Operaciones soportadas sobre el tipo String:

- **Arithmetic operations**
  - **String1 + String2**: La operación suma en Strings se traduce en la concatenación de las dos cadenas de caracteres.

- **Relational operations**
  - **String1 = String2**: Compara dos Strings y si el contenido de los dos Strings es el mismo devuelve un booleano con valor true.
  - **String1 != String2**:Compara dos Strings y si el contenido de los dos Strings es diferente devuelve un booleano con valor true.

La otra gran característica de nuestro lenguaje es la gran variedad de funciones predefinidas con las que poder trabajar. Hay multitud de funciones que permiten operar con los diferentes tipos de datos definidos.

Al implementar las funciones predefinidas hemos querido simular el pasar por **referencia o por valor**. Todas las funciones cuyo nombre acabe con '!' son funciones que simulan el paso por referencia y por lo tanto, las operaciones se aplicarán sobre los parámetros, mientras que en las otras, las operaciones se aplicarán sobre una copia y se retornará la cópia.

Para dar comodidad al programar con MoJo **se permite en ciertas funciones un paso de parámetros variable**. Un ejemplo es la función `add_row!(table, row1, row2,...)` en la que le podemos passar todas las rows que queramos añadir a table. El único requisito es que al menos se pasen dos parámetros: table y row1.

## Instrucciones y Funciones Predefinidas

- **create_table**([column_names]): Crea una tabla vacía con columnas con nombres column_names.
    - Requisito: [column_names] tiene que ser una lista de Strings.

- **add_row**(table, row) : Añade una nueva fila.
    - Requisito: table es de tipo Table, row es de tipo Dictionary
    - Retorna: retorna una copia de table con la nueva row añadida.

- **add_row!**(table, row) : Añade la fila [values] a la tabla table.
    - Requisito: table es de tipo Table, row es de tipo Dictionary
    - Retorna: nada.

- **add_column**(table, column): Añade una nueva columna a una tabla.
    - Requisito:  table es de tipo Table, column es de tipo String
    - Retorna: Una copia de table con la columna añadida.

- **add_column!**(table, column): Añade la columna con nombre column a la tabla table
    - Requisito: table es de tipo Table, column es de tipo String
    - Retorna: Nada.

- **drop**(table): Elimina la tabla de la memòria.

- **drop**(table, column): Elimina la columna de la tabla table.
    - Requisito: table es de tipo Table, column es de tipo String.
    - Retorna: Nada

- **drop**(table, row):  Elimina la row de la tabla table.
    - Requisito: table es de tipo Table, row es de tipo Dictionary.
    - Retorna: nada.

- **sample**(porcentaje, table): Selecciona con una probabilidad 100/porcentaje. columnas de la tabla.

    - Requisito: porcentaje es de tipo Integer, table es de tipo Table.
    - Retorna: Una nueva tabla con las columnas que han sobrevivido.

- **source**(file): Llama a la función main() del programa file y devuelve el resultado de la ejecución.
    - Requisito: file es de tipo String.
    - Retorna: Cualquier tipo en función del programa.

- **read_file**(file): Lee una tabla de un fichero .csv
    - Requisito: file es un String
    - Retorna: Una tabla.

- **write_file**(table, file): Escribe una tabla en un fichero en formato .csv
    - Requisito: table es de tipo Tabla, file es de tipo String.
    - Retorna: Nada.

- **sort**(table, col): Ordena la tabla teniendo en cuenta el valor de la columna col.
    - Requisito: table es de tipo Table, col es de tipo String.
    - Retorna: Una tabla ordenada.

- **merge**(table1, table2): Hace un merge de las dos tablas.
    - Requisito: table1 y table2 son de tipo Table.
    - Retorna: La tabla resultante de hacer el merge.

- **num_rows**(table): Devuelve el número de filas que contiene una tabla pasada por parámetro.
    - Requisito: table es de tipo Table.

- **num_columns**(table): Devuelve el número de columnas de la tabla.
    - Requisito: table es de tipo Table.

- **column_names**(table): Retorna una lista con el nombre de las columnas de una tabla como elementos.
    - Requisito: table es de tipo Table.

- **length**(list): Retorna el tamaño de una lista.
    - Requisito: list es una lista.

# Roadmap

Hubo ciertas features que nos hubiese gustado añadir:

- Funciones anónimas
- Ejecución de otros scripts pasando argumentos
- Permitir overloading de funciones
- Bucle for