**Notas de clase – Taller de Stata**[1]

**Clase 7 – Macros**

## Contenido

1. Creación de macros
2. Uso de macros
3. Diferencias entre local y global
4. Macros y opciones de ejecución al interior de do-files
5. Funciones extendidas y macros
6. Uso directo de macros en expresiones
7. Macros tipo local: tempvar, tempname y tempfile



[Estas notas se diseñaron con base en la versión 16 de *Stata*, conforme el software cambie las notas deben ser actualizadas].

[1] Estas notas están basadas en la guía desarrollada previamente para este curso por Rodrigo Azuero Melo, Nicolás de Roux, Luis Roberto Martínez, Román Andrés Zárate y Santiago Gómez Echeverry.

Taller de Stata 2020-II
Clase 7 – Macros
Miguel Garzón Ramírez, Cristhian Acosta Pardo
Facultad de Economía, Universidad de los Andes

Las macros son objetos donde *Stata* almacena información temporalmente. En particular, las macros pueden entenderse como abreviaciones de una cadena de caracteres (como nombres de variables o listas de ellas) o de números. Las macros pueden ser de dos tipos, local y global.

## 1. Creación de macros

Este tipo de macros en *Stata* pueden contener uno o múltiples objetos (números, letras o caracteres alfanuméricos como nombres de variables). Estas se caracterizan por tener un nombre y un contenido asociado a ese nombre. Una vez se ha definido una macro, *Stata* remplaza el nombre por el contenido cuando se llama a la macro.

Hay dos grandes tipos de macros: *locals* y *globals*. Las macros pueden tener nombres de hasta 31 caracteres y se definen usando los comandos local y global. La sintaxis básica para definir un *local* es:

local *nombre_local* contenido_local

La sintaxis para definir una macro *global* es análoga

Si el contenido del *local* o *global* es un *string* de más de una palabra se debe usar comillas para que *Stata* entienda dónde empieza y dónde termina su contenido. Si el contenido del *local* es una expresión numérica se debe usar el símbolo igual (=).



Un ejemplo de una expresión numérica es:

Ej. 1 ⟶ local *suma* = 2 + 2 que es equivalente a local *suma* = 4

Un ejemplo de una expresión de texto es:

Ej. 2 ⟶ local *vars_exp* "var1 var 2 var3"

## 2. Uso de macros

Para utilizar la información de las macros estas deben ser llamadas. Su sintaxis es diferente cada vez. Para llamar el *local* se escribe el nombre entre el acento grave (`) y la comilla sencilla (').

reg *var_dep `vars_exp'* // Esto es equivalente a reg *var_dep var1 var 2 var3*

Para llamar a un objeto *global* se utiliza el símbolo de dinero ($) antes de su nombre:

Ej. 3 ⟶ global *vars_exp* "var1 var 2 var3"

reg *var_dep* $*vars_exp* // Esto es equivalente a reg *var_dep var1 var 2 var3*

La ventaja de la sintaxis de una macro *local* es que tiene inicio y fin, esto facilita la creación de rutinas completas de programación como enlace o anidación de macros. La sintaxis para llamar a una macro *global* no tiene un final explícito y requiere de un espacio en seguida. Esto es inconveniente, pero tiene solución utilizando las llaves { y } así:

Ej. 4 ⟶ reg *var_dep* ${*vars_exp*}

Llamar a una macro que no existe no genera un error en *Stata*, el programa simplemente remplaza esa macro por el vacío. Esto puede resultar en ejecuciones de código no deseadas. Por ejemplo, si se define el *local controles* como "ingreso edad peso" y se ejecuta: reg infarto ejercicio `controls'. *Stata* simplemente omite ejecuta: reg infarto ejercicio. La razón es que el *local controles* existe mientras que el *local* controls (falta la letra "e" en el nombre) no existe.

### 3. Diferencias entre local y global



Además de las diferencias en la sintaxis de definición y al llamado, *global* y *local* se diferencian en términos de su alcance, lugar de almacenamiento y uso dentro del lenguaje de programación. Un *local* sólo existe y se puede usar en la ejecución continua de un *do-file* en el que fue definido. Un *global* está definido para todos los programas, do-files o ejecuciones de un do-file de una misma sesión de *Stata*. En cambio, un *local* se crea y solo existe mientras se ejecuta cada *do-file* o parte de él) y desaparece cuando la ejecución del *do-file* finaliza. Un *global*, en cambio, posee el mismo contenido a lo largo de la ejecución de varios programas ya que permanece en la memoria de la sesión y puede ser redefinido a lo largo de ella.

Para obtener una lista de los *locals* y *globals* que están definidos en el instante de interés se debe ejecutar el comando macro dir[2]. Esto implica que si se quiere ver los *locals* definidos en la ejecución de un do-file, es necesario que las líneas ejecutadas del do-file (en las que se definen los *locals*) también ejecuten macro dir. En cambio, la lista de *globals* almacenados se puede consultar en cualquier momento.

[2] El comando macro list hace exactamente lo mismo.

## 4. Macros y opciones de ejecución al interior de do-files

Un uso de los *locals* es definir opciones de ejecución al interior de los programas. Para usar los *locals* de esta forma es necesario introducir el uso del condicional *if* en programación.

En notas anteriores se vio que *if* sirve para definir condiciones lógicas que las observaciones deben cumplir en la ejecución de un comando. Por ejemplo,

Ej. 5 ⟶ sum *ingresos* if *edad* >16

Sin embargo, if, como comando, tiene otro uso que es particularmente útil cuando se tiene condiciones no relacionadas a valores de variables y cuando se quiere ejecutar una serie de comandos siempre que se cumpla con una cierta condición. La sintaxis para usar en estos casos es:

```
if    condición lógica {
      conjunto de comandos a ejecutar
}
```

*Stata* evalúa la condición lógica y si es verdadera (si se cumple) entra a ejecutar el conjunto de comandos incluidos entre las llaves. Note que la llave de apertura va en la misma línea de la condición, mientras que la llave de cierre va en una línea independiente posterior a la ejecución de los comandos.



Podemos hacer uso de varios *if*, donde cada uno de estos especifica una condición distinta. Sin embargo, en el caso particular en el que simplemente nos interesa realizar un proceso diferente si la condición inicial es falsa se puede hacer uso de la opción else:

```
if        condición lógica {
          conjunto de comandos a ejecutar
}
else if   condición lógica   {
          conjunto de comandos a ejecutar
}
else      {
          conjunto de comandos a ejecutar si la condición de if es falsa
}
```

Este uso del if es particularmente útil cuando la condición está definida en términos de una macro que el usuario cambia según la manera en que desea ejecutar el programa. La idea es crear

un programa que puede seguir distintos caminos, donde el camino específico que se toma depende de unas macros especificadas al inicio de este.

Por ejemplo,

Ej. 6⟶
```
local opción = 1
if `opcion' == 1 {
        comandos para ejecutar si se escogió la opción 1
}
else{
        comandos para ejecutar si no se escogió la opción 1
}
```
Si la condición involucra texto es necesario el uso de dobles comillas. Por ejemplo, si se define:

Ej. 7 ⟶ local *genero* "femenino"

Para evaluar el *local genero* comparándolo con un *string* es necesario introducirlo dentro de comillas (if " `genero' " == "femenino"). De lo contario, *Stata* entiende que se quiere comparar la variable *genero* (que puede no existir) contra un valor particular de la misma.

## 5. Funciones extendidas y macros



*Stata* cuenta con un conjunto de funciones extendidas que permiten definir tipos particulares de macros tipo *local* o *global*. Estas funciones extendidas tienen una sintaxis común:

local *nombre*: función extendida.

Algunos ejemplos se muestran en la tabla 2.

Tabla 1: Ejemplos de funciones extendidas con macros

| Sintaxis | Resultado |
|---|---|
| local x : type var1 | Tipo de var1 (byte, float, etc.) |
| local x : format var1 | Formato de var1 (%9.0g, %12s, etc.) |
| local x : sortedby | Variables por las que está ordenada la base de datos. |
| local x : label label1 # | El label asociado con el número # según el conjunto de labels 'label1' |
| local x : word count string | Número de palabras en string |

Para obtener la lista completa de funciones extendidas se puede consultar el help de *Stata*. En particular, al ejecutar el comando help *extended_fcn*.

Además de las funciones extendidas, es posible definir macros a partir de la información guardada por *Stata* en las listas: e(), r() y c() . La carpeta c() guarda información del sistema y de la base de datos[3], y, como se vio en clases anteriores, e() guarda los resultados de estimaciones (por ejemplo, regresiones) mientras que r() guarda resultados de otros comandos (sum es un ejemplo). En estos casos se utiliza el símbolo = en vez de los dos puntos (:) característicos de las funciones extendidas. Por ejemplo, para definir un *local* con la fecha del día se ejecuta:

Ej. 8 ⟶ local *fecha* =c(current_date)

### 6. Uso directo de macros en expresiones

Ya hemos visto que es posible definir una macro y después usarla al interior de comandos. Por ejemplo:

Ej. 9 ⟶ local *z* : format x
format y `z'

En este caso los comandos le asignan a la variable *y* el formato de la variable *x*. *Stata* permite obviar el paso de definición de la macro y permite evaluarla directamente en el segundo comando. En el caso anterior se obtendría el mismo resultado al escribir:

Ej. 10 ⟶ format y `:format x'

### 7. Tempvar, tempname y tempfile

Estos comandos son macros tipo *local* que permiten crear nombres que pueden ser usados, sólo temporalmente, para variables, escalares, matrices y archivos. Estos elementos temporales existen mientras el programa o do-file esté corriendo, pero una vez concluye la ejecución automáticamente dejan de existir.

Para la creación de variables temporales debe emplearse el comando tempvar. Suponga que requiere crear una variable (*newvar2*) con la raíz cuadrada de la suma de los cuadrados de dos variables (*var1* y *var2*) de la siguiente forma:

Ej. 11 ⟶ tempvar newvar
gen `newvar' = var1^2 + var2^2

[3] Escriba en el help "creturn list" para ver toda la información disponible.

```
gen newvar2 = sqrt(`newvar')
```

Una de las ventajas de usar nombres de variables temporales es que al final de la ejecución del comando no requiere borrar las variables. *Stata* lo hace cuando se termina el programa automáticamente. El comando tempname es el equivalente de tempvar para obtener nombres para escalares y matrices.

Por último, el comando tempfile permite crear archivos temporales. Por ejemplo:

Ej. 12 ⟶

```
tempfile master
save `master', replace
```

La utilidad de emplear el comando tempfile es evitar la generación de archivos correspondientes a bases intermedias en la carpeta de trabajo, los cuales ocupan memoria en el ordenador.