JEM174 Stata for Economists
Institute of Economic Studies
Charles University in Prague

Radek Janhuba
janhuba@gmail.com
Spring 2018

Contents

6 Basics of Programming **1**
6.1 Macros in Stata 1
6.1.1 Two Types of Macros 1
6.1.2 Calling a Macro 1
6.1.3 Assigning Macro Content 2
6.1.4 Using Macros 4
6.1.5 The Importance of Calling the Right Macro Name 6
6.1.6 Example: Automatic Processing of Log Files 7
6.2 Loops 7
6.2.1 `foreach` Loops 7
6.2.2 `forvalues` Loops 12
6.2.3 Nested Loops 13
6.3 If and Then Branching 14
6.4 Programming Techniques and Regressions 15
6.4.1 Drawing Predictions at Means of Other Regressors 15
6.4.2 State Specific Regressions 17
6.5 Custom Programs 18
6.6 Extended Macro Functions 20

6 Basics of Programming

In this last section of the course, we will learn how to make our do files more automatic.

6.1 Macros in Stata

Before we start with loops, we need to understand the concept of macros. There are two kinds of macros in Stata: *local* and *global*. For the purpose of introduction, we will use mainly local macros - they are the most commonly used, moreover, nothing really distinguishes one from the other at this stage.

6.1.1 Two Types of Macros

The difference between local and global macros is that you can only access the local macros in the session/do file/program you assign their contents, while you can access global macros "anywhere". However, to quote Stata help,

> *Global macros are rarely used, and when they are used, it is typically for communication between programs. You should never use a global macro where a local macro would suffice.*

6.1.2 Calling a Macro

Essentially, macros are strings that are stored in Stata's memory. When a macro is called, Stata looks at what is inside and replaces its name with its content.

If you assign the value of 1 into a local macro `x`,

```
. local x 1
```

you can then use it in a calculation or simply display its value:

```
. display ‘x’ // executes "display 1"
1
```

Notice how a local macro is called - with an opening single quote (located under the Esc key on American keyboards)[1] and with a closing single quote (also called an apostrophe).[2]

6.1.3 Assigning Macro Content

Generally, we can either assign any string into a macro, or you can use a “=” when assigning a macro. In such a case, the expression gets evaluated first and only then its result will be input into the contents of the created macro. Take the difference between the next two macros:

```
. local a 2+2

. local b = 2+2
```

If you write

```
. display ‘a’ // display 2+2
4

. display ‘b’ // display 4
4
```

there seems to be no difference, as 2+2 actually is 4 (in the case of *a*, display calculated this value, while in case of *b*, it was already calculated above). If you, however, write

```
. display "‘a’" // display "2+2"
2+2

. display "‘b’" // display "4"
4
```

you can see the difference between the two. Moreover, if you write

```
. display 2*‘a’ // display 2*2+2
6

. display 2*‘b’ // display 2*4
8
```

[1]As far as I know, the sign is not present on a Czech keyboard. It can be written by the *alt+96* combination.

[2]Since we do not work with global macros, the way how to call them has been left to this footnote: A global macro called *globalname* is called as `$globalname`.

you can see that the results may be very different. To take another example, let's fit a regression and store its R^2:

```
.  sysuse auto, clear
(1978 Automobile Data)

. reg mpg weight

      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(1, 72)        =    134.62
       Model |   1591.9902         1   1591.9902   Prob > F        =    0.0000
    Residual |  851.469256        72  11.8259619   R-squared       =    0.6515
-------------+----------------------------------   Adj R-squared   =    0.6467
       Total |  2443.45946        73  33.4720474   Root MSE        =    3.4389

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.0060087   .0005179   -11.60   0.000    -.0070411   -.0049763
       _cons |   39.44028   1.614003    24.44   0.000     36.22283    42.65774
------------------------------------------------------------------------------

. local rsqf e(r2) // stores only the string "e(r2)"

. local rsqv = e(r2) // evaluates and stores e(r2)

. display ‘rsqf’          // this has the current R-squared
.65153125

. display ‘rsqv’          // as does this
.65153125

. reg mpg weight foreign // run another regression

      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(2, 71)        =     69.75
       Model |   1619.2877         2  809.643849   Prob > F        =    0.0000
    Residual |  824.171761        71   11.608053   R-squared       =    0.6627
-------------+----------------------------------   Adj R-squared   =    0.6532
       Total |  2443.45946        73  33.4720474   Root MSE        =    3.4071

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
      weight |  -.0065879   .0006371   -10.34   0.000    -.0078583   -.0053175
     foreign |  -1.650029   1.075994    -1.53   0.130      -3.7955    .4954422
       _cons |    41.6797   2.165547    19.25   0.000     37.36172    45.99768
------------------------------------------------------------------------------

. display ‘rsqf’          // the formula has the new R-squared
.66270291
```

```
. display ‘rsqv’          // this guy has the old one
.65153125
```

In case you want to clear the macro of its contents, you can either assign empty quotes into it

```
.  local indepvars ""
```

or you can just write its name (in such case there is “nothing” that follows it, so the contents of the macro will be replaced with “nothing”)

```
.  local indepvars
```

6.1.4 Using Macros

If we assign a local macro called *indepvars* as

```
.  local indepvars price weight foreign
```

and then write

```
.  sum ‘indepvars’

    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       price |        74    6165.257    2949.496       3291      15906
      weight |        74    3019.459    777.1936       1760       4840
     foreign |        74    .2972973    .4601885          0          1

. reg mpg ‘indepvars’

      Source |       SS          df       MS      Number of obs   =        74
-------------+----------------------------------   F(3, 70)        =     45.93
       Model |  1620.30716         3  540.102388   Prob > F        =    0.0000
    Residual |  823.152295        70  11.7593185   R-squared       =    0.6631
-------------+----------------------------------   Adj R-squared   =    0.6487
       Total |  2443.45946        73  33.4720474   Root MSE        =    3.4292

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       price |   .0000566   .0001922     0.29   0.769    -.0003268      .00044
      weight |  -.0067758   .0009048    -7.49   0.000    -.0085805   -.0049712
     foreign |  -1.855891   1.289063    -1.44   0.154    -4.426846    .7150641
       _cons |   41.95948   2.377726    17.65   0.000     37.21725     46.7017
------------------------------------------------------------------------------
```

Stata reads exactly the same as the following two lines (output omitted):

```
. sum price weight foreign
. reg mpg price weight foreign
```

The advantage with the first method is that if you want to add *i.rep78* into the analysis, it is much easier to only make this change in the definition of the local macro:

```
.  local indepvars price weight foreign i.rep78

. sum ‘indepvars’

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
       price |         74    6165.257    2949.496       3291      15906
      weight |         74    3019.459    777.1936       1760       4840
     foreign |         74    .2972973    .4601885          0          1
             |
       rep78 |
         2   |         69     .115942    .3225009          0          1
-------------+---------------------------------------------------------
         3   |         69    .4347826    .4993602          0          1
         4   |         69    .2608696    .4423259          0          1
         5   |         69    .1594203    .3687494          0          1

. reg mpg ‘indepvars’

      Source |       SS           df       MS      Number of obs   =        69
-------------+----------------------------------   F(7, 61)        =     19.90
       Model |  1627.48579         7  232.49797   Prob > F        =    0.0000
    Residual |  712.717109        61  11.683887   R-squared       =    0.6954
-------------+----------------------------------   Adj R-squared   =    0.6605
       Total |   2340.2029        68  34.4147485   Root MSE        =    3.4182

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       price |   .0000604   .0002015     0.30   0.765    -.0003424    .0004633
      weight |  -.0064961   .0009727    -6.68   0.000    -.0084411   -.0045511
     foreign |   -3.13669    1.52857    -2.05   0.044    -6.193255    -.080126
             |
       rep78 |
         2   |  -.3114325   2.710206    -0.11   0.909    -5.730825     5.10796
         3   |  -.0729958   2.513312    -0.03   0.977    -5.098675    4.952683
         4   |   .6498104    2.62137     0.25   0.805    -4.591943    5.891564
         5   |   3.799259   2.800019     1.36   0.180    -1.799725    9.398244
             |
       _cons |     40.862   3.447222    11.85   0.000     33.96885    47.75514
------------------------------------------------------------------------------
```

6.1.5 The Importance of Calling the Right Macro Name

Notice that in case a non-existing macro is called, Stata evaluates its contents as empty and does not substitute anything. This can lead to a lot of confusions and generally incorrect results, as the following example demonstrates.

Suppose that we make a mistake and call a macro *depvars* instead of *indepvars*. In such case, Stata will read the commands above as is noted behind the double slashes below - it will first summarize all variables and then run a regression of mpg on a constant only (an output of `sum` and `reg` without an argument).

```
.  local indepvars price weight foreign i.rep78

. sum ‘depvars’ // sum

    Variable |        Obs        Mean    Std. Dev.       Min        Max
-------------+---------------------------------------------------------
        make |          0
       price |         74    6165.257    2949.496       3291      15906
         mpg |         74     21.2973    5.785503         12         41
       rep78 |         69    3.405797    .9899323          1          5
    headroom |         74    2.993243    .8459948        1.5          5
-------------+---------------------------------------------------------
       trunk |         74    13.75676    4.277404          5         23
      weight |         74    3019.459    777.1936       1760       4840
      length |         74    187.9324    22.26634        142        233
        turn |         74    39.64865    4.399354         31         51
displacement |         74    197.2973    91.83722         79        425
-------------+---------------------------------------------------------
  gear_ratio |         74    3.014865    .4562871       2.19       3.89
     foreign |         74    .2972973    .4601885          0          1

. reg mpg ‘depvars’ // reg mpg

      Source |       SS           df       MS      Number of obs   =        74
-------------+----------------------------------   F(0, 73)        =      0.00
       Model |           0         0           .   Prob > F        =         .
    Residual |  2443.45946        73  33.4720474   R-squared       =    0.0000
-------------+----------------------------------   Adj R-squared   =    0.0000
       Total |  2443.45946        73  33.4720474   Root MSE        =    5.7855

------------------------------------------------------------------------------
         mpg |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       _cons |    21.2973   .6725511    31.67   0.000      19.9569    22.63769
------------------------------------------------------------------------------
```

The code will run, but all the values from the regression will be erroneous. If we, however, decide to use a coefficient of *weight*, Stata will report an error:

```
.  display _b[weight]
[weight] not found
r(111);
```

6.1.6 Example: Automatic Processing of Log Files

One particularly useful application of macros is a creation of log files. Using the following code, we only need to change the desired log name in the beginning and the log will be opened in the beginning, closed at the end and translated into a text file of the particular name:

```
.  local logname Part6log

. log using ‘logname’, replace
------------------------------------------------------------------------------------------
      name:  <unnamed>
       log:  C:\Dropbox\CERGE\Teaching_IES_SFE\LN\Part6log.smcl
  log type:  smcl
 opened on:   9 Feb 2018, 12:28:31

. display "this is a sample log called ‘logname’"
this is a sample log called Part6log

. log close
      name:  <unnamed>
       log:  C:\Dropbox\CERGE\Teaching_IES_SFE\LN\Part6log.smcl
  log type:  smcl
 closed on:   9 Feb 2018, 12:28:32
------------------------------------------------------------------------------------------

. translate ‘logname’.smcl ‘logname’.txt, replace
(file Part6log.txt written in .txt format)
```

6.2 Loops

In cases where you need to use the same set of commands for a lot of variables or where you generally need to run the same piece of code more times, it may be appropriate to utilize loops.

6.2.1 `foreach` Loops

The first kind of loops which we will cover is a `foreach` loop. With this kind, you specify a name you will call within the loop, a type of a list you will use, and the list itself. Inside of the loop, you will then specify commands that you want to be repeated. As loops are best understood by means of an example, let’s write our first loop:

```
.  foreach color in red blue green {
.         display "‘color’"
. }
red
blue
green
```

This, rather simple loop takes the three colors (red, blue and green) and displays them on the screen.
Notes:

- Each loop has to finish with an opening brace after the list and has to then end with a single closing brace on a line that contains nothing else. Therefore, each loop needs to span at least three rows.
- The `display` command is indented - although not required, it is a good programming practice to always indent contents of what's inside of the loop.
- the `color` which specifies the name of each object in the list is referenced as a local macro.[3]
- It is also enclosed in the quotation marks, telling Stata to display a string. If you do not put the quotation marks in, Stata will report an error "red not found":

```
.  foreach color in red blue green {
.         display `color'
. }
red not found
r(111);
```

You can also use loops to generate variables. For illustration, let's now load an example dataset *months*:

```
.  use months, clear

. describe

Contains data from months.dta
  obs:            10
 vars:            13                           14 Apr 2010 15:24
 size:           280
--------------------------------------------------------------------------------------------------------
              storage   display    value
variable name   type    format     label      variable label
--------------------------------------------------------------------------------------------------------
id              float   %9.0g
incJan          int     %8.0g
incFeb          int     %8.0g
incMar          int     %8.0g
incApr          int     %8.0g
incMay          int     %8.0g
incJun          int     %8.0g
incJul          int     %8.0g
incAug          int     %8.0g
incSep          int     %8.0g
incOct          int     %8.0g
incNov          int     %8.0g
incDec          int     %8.0g
--------------------------------------------------------------------------------------------------------
Sorted by:
```

[3] More specifically, it is defined on the first line and then referenced three times on the second line.

We can see that the dataset contains 13 variables for 10 people, specifying their id and their income in each of the 12 months. Suppose now you want to create a variable *hadInc** which would contain information on whether they had any income in these months. This can be done in a loop:

```
.  foreach month in Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec {
.          gen hadInc`month'=(inc`month'>0) if !missing(inc`month')
.          label var hadInc`month' "1 if inc`month' > 0"
. }

. describe had* // check it's created correctly

              storage   display    value
variable name   type    format     label      variable label
---------------------------------------------------------------------------------------------------------
hadIncJan       float   %9.0g                 1 if incJan > 0
hadIncFeb       float   %9.0g                 1 if incFeb > 0
hadIncMar       float   %9.0g                 1 if incMar > 0
hadIncApr       float   %9.0g                 1 if incApr > 0
hadIncMay       float   %9.0g                 1 if incMay > 0
hadIncJun       float   %9.0g                 1 if incJun > 0
hadIncJul       float   %9.0g                 1 if incJul > 0
hadIncAug       float   %9.0g                 1 if incAug > 0
hadIncSep       float   %9.0g                 1 if incSep > 0
hadIncOct       float   %9.0g                 1 if incOct > 0
hadIncNov       float   %9.0g                 1 if incNov > 0
hadIncDec       float   %9.0g                 1 if incDec > 0
```

Of course, the same task could be dealt with by applying `reshape` two times, however, that procedure is not able to label the variables in such an informative manner.

The second example will utilize the *data1* data coming from Kohler & Kreuter (2012) which we also used in previous parts of the course.

```
.  use data1, clear
(SOEP 2009 (Kohler/Kreuter))
```

Suppose now that we want to create centered versions $(x_i - \mu_i)$ for variables *income*, *hhinc*, *size* and *rent*. We want to call these variables *c_var* and label them *var (centered)*:

```
.  foreach x of varlist income hhinc size rent {
.          qui sum `x' // quietly summarize the variable to create r(mean)
.          gen c_`x' = `x' - r(mean) // generate the centered version
.          label var c_`x' "`x' (centered)" // label the new variable
. }
(632 missing values generated)
(4 missing values generated)
(3,049 missing values generated)

. describe c_* // check that they were created

              storage   display    value
variable name   type    format     label      variable label
```

```
------------------------------------------------------------------------------------------------
c_income        float   %9.0g                 income (centered)
c_hhinc         float   %9.0g                 hhinc (centered)
c_size          float   %9.0g                 size (centered)
c_rent          float   %9.0g                 rent (centered)

. tabstat c_* // check that mean of the new variables is very close to 0

   stats |  c_income   c_hhinc    c_size    c_rent
---------+----------------------------------------
    mean |  .0002214   .000039  4.02e-06 -1.96e-06
--------------------------------------------------
```

The last type of a list commonly specified in a `foreach` loop is a general local macro, where its contents will be taken piece by piece separated by spaces. Let's define a macro *country* holding codes for four countries and a macro *count* holding a counter (starting from 0):

```
.  local count 0

. local country US UK DE FR

. display ‘count’ // specified correctly
0

. display "‘country’" // also correct (remember we need the double quotes here)
US UK DE FR
```

We can now list these countries together with their number on the screen using a loop with `display`:

```
.  foreach c of local country {
.        local count = ‘count’+1
.        display "Country ‘count’: ‘c’"
. }
Country 1: US
Country 2: UK
Country 3: DE
Country 4: FR
```

Notes:

- The existing macro *country* is called without single quotes after the list type specification `local`. If you call it as a macro somewhere else (that is, by typing `local ‘country’` instead of `local country`), Stata will substitute the contents of the macro and put them into the first command (it will read `foreach c of local US UK DE FR`), causing an error:[4]

```
.  foreach c of local ‘country’ {
.        local count = ‘count’+1
.        display "Country ‘count’: ‘c’"
```

[4]Explanation of the error: Stata expects { to appear, but instead, `U` from "UK" appears.

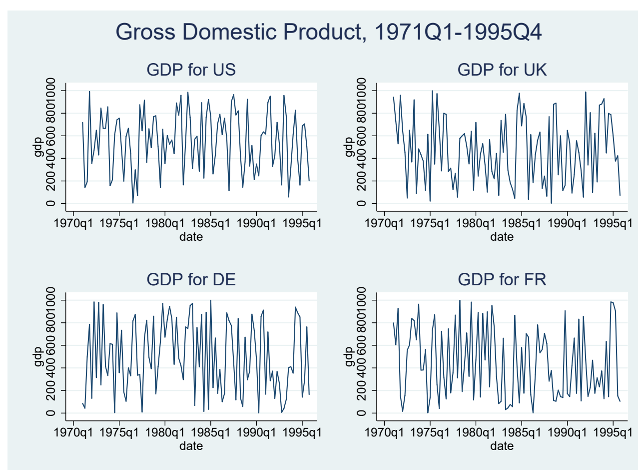
```
. }
{ required
r(100);
```

- The “=” sign in the second line is crucial in this context, as it causes the *count* macro to evaluate the expression and increment its value by 1 in each step. If we omit the sign, we will get the following:

```
.  local count 0

. local country US UK DE FR

. foreach c of local country {
.         local count ‘count’+1
.         display "Country ‘count’: ‘c’"
. }
Country 0+1: US
Country 0+1+1: UK
Country 0+1+1+1: DE
Country 0+1+1+1+1: FR
```

The last example of `foreach` shows us how to create a combined graph utilizing a macro:

```
.  use gdp4cty, clear

. local country US UK DE FR

. foreach c of local country {
.         line gdp date if cty=="‘c’", title("GDP for ‘c’") ///
                 nodraw name(‘c’,replace)

. }

. graph combine ‘country’, ti("Gross Domestic Product, 1971Q1-1995Q4")
```



Notice the option `nodraw` which generates the graph, but does not show it on the screen - this is a useful option if you need to get a lot of graphs and then combine them together.

6.2.2 `forvalues` Loops

The second type of loops that we will cover are `forvalues` loops. They are, essentially, very similar to loops of the previous type, except they process numbers instead of strings. We can thus use the `numlist` operators as `1/10` or `100(10)150`.

Take the following (very simple) loop:

```
.  forvalues i=1/5 {
.          display ‘i’
. }
1
2
3
4
5
```

or a very similar (and also very simple) loop:

```
.  forvalues i=2(2)10 {
.          display 10*‘i’
. }
20
40
60
80
100
```

Coming back to the income example, we will load a similar dataset containing data for income in given years:

```
.  use years, clear

. forvalues year=1990/2010 {
.          gen hadInc‘year’=(inc‘year’>0) if inc‘year’<.
.          label var hadInc‘year’ "inc‘year’ == 1"
. }

. describe had* // check it’s created correctly

               storage   display    value
variable name   type     format     label      variable label
------------------------------------------------------------------------------------------
hadInc1990      float    %9.0g                 inc1990 == 1
hadInc1991      float    %9.0g                 inc1991 == 1
hadInc1992      float    %9.0g                 inc1992 == 1
hadInc1993      float    %9.0g                 inc1993 == 1
hadInc1994      float    %9.0g                 inc1994 == 1
hadInc1995      float    %9.0g                 inc1995 == 1
```

```
hadInc1996      float   %9.0g                 inc1996 == 1
hadInc1997      float   %9.0g                 inc1997 == 1
hadInc1998      float   %9.0g                 inc1998 == 1
hadInc1999      float   %9.0g                 inc1999 == 1
hadInc2000      float   %9.0g                 inc2000 == 1
hadInc2001      float   %9.0g                 inc2001 == 1
hadInc2002      float   %9.0g                 inc2002 == 1
hadInc2003      float   %9.0g                 inc2003 == 1
hadInc2004      float   %9.0g                 inc2004 == 1
hadInc2005      float   %9.0g                 inc2005 == 1
hadInc2006      float   %9.0g                 inc2006 == 1
hadInc2007      float   %9.0g                 inc2007 == 1
hadInc2008      float   %9.0g                 inc2008 == 1
hadInc2009      float   %9.0g                 inc2009 == 1
hadInc2010      float   %9.0g                 inc2010 == 1
```

6.2.3 Nested Loops

Previous examples sufficed with one instance of a loop. In many applications, this will not be enough. For such cases, loops can be nested. Consider now a very simple example of generating all the possible combinations from throwing of two dice. We can generate this by typing

```
.  forval i=1/6 {
.         forval j=1/6 {
.                 display "`i',`j'"
.         }
. }
1,1
1,2
1,3
1,4
1,5
1,6
2,1
2,2
2,3
2,4
2,5
2,6
3,1
3,2
3,3
3,4
3,5
3,6
4,1
4,2
4,3
4,4
4,5
4,6
```

```
5,1
5,2
5,3
5,4
5,5
5,6
6,1
6,2
6,3
6,4
6,5
6,6
```

Note that this took much shorter than if we would have written

```
display "1,1"
display "1,2"
etc
```

while also leaving much less for possible errors and/or typos.

Consider now the following dataset with incomes that combines data from several years and months. If we want to generate the same dummies as in previous examples, we could (similarly as in the previous examples) use a two-way `reshape` or we can combine the two types of loops.

What `reshape` would not do, however, is to identify on its own (without using functions monthly, mofd, or similar) consecutive numbers of periods and identifying that January 1991 is one month after December 1990. We can do it with a loop in the following way:

```
.  use monthyear, clear

. local period 1

. forval year=1990/2010 {
.         foreach month in Jan Feb Mar Apr May Jun Jul Aug Sep Oct Nov Dec {
.                 gen hadInc`period' = (inc`month'`year' > 0) ///
                                 if !missing(inc`month'`year')

.                 label var hadInc`period' "inc`month'`year' == 1"
.                 local period=`period'+1
.         }
. }
```

6.3 If and Then Branching

In many programming applications, you will want to execute different pieces of code given some condition being true. Take e.g. the following example:

```
.  local start = date("28/12/2015","DMY")

. local end =  date("02/01/2016","DMY")
```

```
. forval date = `start'/`end' {
.         local mth = month(`date')
.         local day = day(`date')
.         local text = "Today is `mth'/`day'."
.         if `mth'==1 & `day'==1 {
.                 local text `text' Happy New Year!
.         }
.         else if `mth'==12 & `day'==31 {
.                 local text `text' Enjoy the party!
.         }
. display "`text'"
. }
Today is 12/28.
Today is 12/29.
Today is 12/30.
Today is 12/31. Enjoy the party!
Today is 1/1. Happy New Year!
Today is 1/2.
```

6.4 Programming Techniques and Regressions

Note: The material starting from now until the end of this file should arguably be included in an advanced rather than an introductory course. Nevertheless, applying techniques explained below will help you work more efficiently and could be helpful while working on the empirical project.

Let's now come back to the *data1* file from Kohler & Kreuter (2012) and go through two examples of how one can utilize programming skills while presenting regression results.

6.4.1 Drawing Predictions at Means of Other Regressors

Suppose we want to fit a regression of home size (*size*) that contains not only household size (*hhsize*) and household income (*hhinc*) but also a location variable for the difference between East and West Germany (states from the East are coded 11 and higher) and an ownership variable indicating owned and rented living space (ownership is coded as 1 in *renttype*).

Let's first prepare the dataset:

```
.  use data1, clear // load the dataset
(SOEP 2009 (Kohler/Kreuter))

. gen east = (state>=11) if !missing(state) // create dummy for east

. gen owner = (renttype==1) if !missing(renttype) // create ownership dummy
```

Second, we can specify the macros holding names of variables and run the model:

```
.  local depvar size

. local indepvar hhinc

. local controlvars hhsize east owner

. reg `depvar' `indepvar' `controlvars'
```

```
      Source |       SS           df       MS      Number of obs   =     5,407
-------------+----------------------------------   F(4, 5402)      =   1075.80
       Model |   507558816         4   126889704   Prob > F        =    0.0000
    Residual |   637158747     5,402  117948.676   R-squared       =    0.4434
-------------+----------------------------------   Adj R-squared   =    0.4430
       Total |  1.1447e+09     5,406  211749.457   Root MSE        =    343.44

------------------------------------------------------------------------------
        size |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
       hhinc |   .0046358   .0001956    23.70   0.000     .0042523    .0050193
      hhsize |   83.56691   4.396585    19.01   0.000     74.94783    92.18599
        east |  -106.6267   10.88597    -9.79   0.000    -127.9676   -85.28581
       owner |   366.1249   9.889078    37.02   0.000     346.7383    385.5115
       _cons |     550.58   12.39905    44.41   0.000     526.2729    574.8872
------------------------------------------------------------------------------
```

Notes:

- If you only run the regression command as a sub-selection of the do file, you will get an error - while running a part of the do file, contents of a local macro must be included in the batch of commands that are selected.
- I chose to specify explanatory variables by two macros rather than one. The motivation for doing so will hopefully become clear in the next piece of code where we calculate the means of other control variables in order to calculate the prediction.

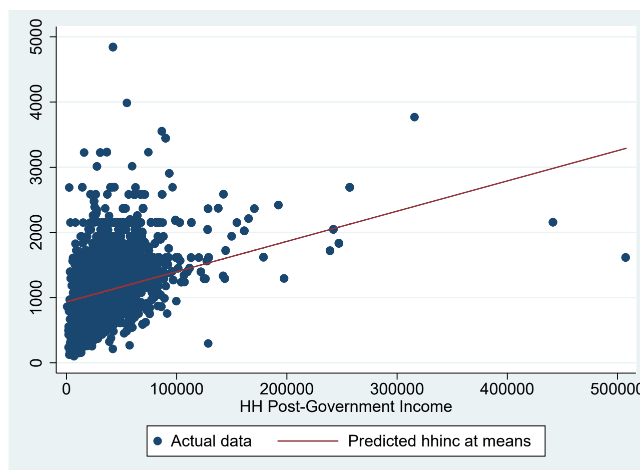
The following loop performs two tasks:

1. Calculates and stores means of control variables (which we want to be evaluated at means)
2. Prepares a local macro *atmeans* holding a string that is a part of the command for calculating the predicted values.

```
.  local atmeans _b[_cons]

. foreach x of local controlvars {
.         qui sum ‘x’ if e(sample)==1
.         scalar m_‘x’ = r(mean)
.         local atmeans ‘atmeans’ + _b[‘x’]*m_‘x’
. }

. disp "‘atmeans’" // check that atmeans was created correctly
_b[_cons] + _b[hhsize]*m_hhsize + _b[east]*m_east + _b[owner]*m_owner
```

We can now generate the graph as follows:

```
.  gen pred = ‘atmeans’ + _b[‘indepvar’]*‘indepvar’
(4 missing values generated)

. twoway (scatter ‘depvar’ ‘indepvar’) (line pred ‘indepvar’, sort(‘indepvar’)), ///
                 legend(order(1 "Actual data" 2 "Predicted ‘indepvar’ at means"))
```



6.4.2 State Specific Regressions

Suppose now that we want to fit a similar regression, however, this time we want to run a regression specifically for each state (note that it follows that we will not be including *east* into the regression).

Before we perform these regressions, let's first define the macros we will need. In doing so, we will use the `levelsof` command which takes all the levels of a variable and stores them into a local macro of a chosen name:

```
.  set more off

. local depvar size

. local indepvars hhinc hhsize owner

. levelsof state, local(states)
0 1 2 3 5 6 7 8 9 12 13 14 15 16
```

Next, we will create a matrix which will hold the regression results. We will utilize two new functions: firstly, `J()` is a function that creates a matrix with specified number of rows and columns and assigns them some value (in our case, a missing value). Secondly, `word count` is an example of an *extended macro function* which are briefly covered below.

```
.  local j : word count `states' // counts the number of states

. local k : word count `indepvars' // counts the number of regressors

. matrix res = J(`j',2*`k'+1,.) // creates the matrix

. local colnames state // initializes string for matrix column names

. foreach x of local indepvars {
.         local colnames `colnames' b_`x' se_`x'
. }

. display "`colnames'" // checks that the macro holds the correct string
state b_hhinc se_hhinc b_hhsize se_hhsize b_owner se_owner
```

```
. matrix colnames res = ‘colnames’ // assingns names to columns
```

Next, we can run the regression in a loop for each state. Inside will be a nested loop which will store the coefficients and their standard errors into the *res* matrix

```
.  local count // Initializes outer count as empty

. foreach i of local states {
.         local count = ‘count’ + 1 // count of how many times loop started
.         qui reg size ‘indepvars’ if state==‘i’ // quiet reg, 1 state only
.         matrix res[‘count’,1] = ‘i’
.         local colcount 1 // Initialize inner count as 1
.         foreach x of local indepvars {
.                 local colcount = ‘colcount’ + 1
.                 matrix res[‘count’,‘colcount’] = _b[‘x’]
.                 local colcount = ‘colcount’ + 1
.                 matrix res[‘count’,‘colcount’] = _se[‘x’]
.         }
. }
```

After the loop finishes, we can list the results on the screen:

```
.  matrix list res

res[14,7]
         state    b_hhinc   se_hhinc   b_hhsize  se_hhsize    b_owner   se_owner
 r1          0  .00441705  .00121758  66.091942  20.696169   331.7662   52.50203
 r2          1   .0066482  .00088314  70.020131  25.487531  301.69619  60.210123
 r3          2  .00734481  .00237307  1.3988229  46.495994  525.91083  100.56558
 r4          3  .00711573  .00086544  84.020702  21.912438  417.32862  50.126857
 r5          5  .00448759  .00043512  81.014695   9.179758  333.45781  20.385996
 r6          6  .00310583  .00046242  71.893248   17.23695  363.05938   35.84326
 r7          7  .00895363  .00113968  33.516251  15.413957  384.33273  38.271279
 r8          8  .00727424  .00059794  49.756483  12.878549  311.51124  30.801966
 r9          9  .00417299  .00062935  119.57979  12.561029  326.67745  28.951227
r10         12  .00140728  .00041321   146.9936  23.975915  407.75885  54.969585
r11         13   .0069607  .00149555  85.285381  23.892708  427.59943  42.713945
r12         14  .00920655  .00145176   44.69394  23.888967  377.43718  40.023816
r13         15  .00305029   .0010988  89.556137  16.213758  366.03756  33.610618
r14         16  .00595737   .0009317   81.16917  11.971161  228.84416  25.595114
```

6.5 Custom Programs

You can actually create your own programs in Stata using advanced techniques described above. The following is the simplest program and its purpose is only that you see a program in action. If you are interested in programming for Stata, chapter 12 of Kohler & Kreuter (2012) provides a good treatment of the topic.

If you type `hello` into Stata, you will get an error.:

```
.  hello
command hello is unrecognized
r(199);
```

However, if you define a program called `hello` with the following:

```
.  capture program drop hello

. program define hello
.         di "Hello World!"
. end
```

You can then use it:

```
.  hello
Hello World!
```

Notice that if you do not include the `capture` line there and try to run it again, Stata will complain *hello already defined*:

```
.  program define hello
program hello already defined
r(110);

.         di "Hello World!"
Hello World!

. end
command end is unrecognized
r(199);
```

Therefore, it is recommended to always include such line just before the program definition.

A perhaps more usable program might be one that prints current date and time on the screen. Following Cameron & Trivedi (2010), such program may look as follows:

```
.  capture program drop time

. program time
.         display c(current_time) c(current_date)
. end
```

With the program defined, whenever we want to print the current time and date, we can simply type

```
.  time
12:29:05 9 Feb 2018
```

6.6 Extended Macro Functions

Extended macro functions are such functions that operate on macros. You can identify them by a colon before the name of the function. However, as they arguably present the most advanced topic of the whole course, their purpose here is to conclude the course by means of a rather silly example. If you are interested in extended macro functions, see `help extended_fcn`.

```
.  local 1 "cat dog cow pig" // lists animals

. local 2 "meow woof moo oinkoink" // lists their sounds

. local n : word count ‘1’ // generates a number of animals

. forvalues i = 1/‘n’ {
.    local a : word ‘i’ of ‘1’ // takes particular animal
.    local b : word ‘i’ of ‘2’ // takes particular sound
.    di "‘a’ says ‘b’" // displays them together on screen.
.    }
cat says meow
dog says woof
cow says moo
pig says oinkoink
```

References

Cameron, A. & Trivedi, P. (2010). *Microeconometrics Using Stata, Revised Edition.* StataCorp LP.

Kohler, U. & Kreuter, F. (2012). *Data Analysis Using Stata, Third Edition.* Taylor & Francis.