Introduction to Stata- A. Chevalier

Lecture 5: Advance programming-

Content of Lecture 5:

-create an ado file
-temporary stuff
-parsing: syntax
-parsing: tokenise and gettoken

A] ado file

Remember in lecture 3, we created our fist ado file, and stored it in our current directory. So if you type hello, you should still be able to get that ado file running.

An ado file, is nothing more than a do file that you are planning to use for various analyses, so that rather than copying and pasting it to your do file every time you need it, you want it to be constantly in stata memory.

Here is an example.

```
*! version 1.0.1  24jan2001 AC

version 6.0

capture program drop misto_99
program define misto_99

syntax [varlist]              <- we will explain these 2 lines later
tokenize `varlist'

while "`1'"~="" {
                  capture confirm string var `1'
                  if  _rc~=0  {
                        quietly recode `1' .=-99
                  }
            mac shift
            }

end
```

Comment: We have seen capture and used it since lecture 3. However, capture does more than executing a command if possible and skipped it if impossible. Capture returns a code 0 in _rc if the command was executed and another code (it depends on the error code) if the command was skipped. This is what we are using here.

misto_99 is so useful, that we want to be able to use all the time To do so, we have to save it as an ado file and save it in the appropriate directory (your personal file).
To find out where your personal file is;

```
. sysdir
```

Do not forget to write a help file to go with your ado file. To do that, go to your editor, write the syntax of your file, and a few notes so that you remember what that do file is doing and save this ASCII file with the same name as your ado file and a .hlp extension. Then save it in your personal directory.

Now if you do help misto_99 the help file associated to the ado file will appear.

Also text put between ^^ will be highlighted and text between @@ will be an hyperlink.
So if you need to type a ^ or @ sign you need to double it.

B] Temporary stuff

The main difference between a do file and an ado file is that in the do file you know, as a programmer, the context in which the program is going to be used. So for example, you know the list of variables, you can create new variable, saving files and opening new ones, because you know their names (or that their names do not exist). This is not the case in an ado file. In section C] we will focus on how to enter lists of variables in an ado file, but here we will look at how to create variables, and files that are not going to already exist. Since there is no way to know that a name has not already been given, there is a class of variables and files only for programmers: temporary variables.

* temporary variables:

You are writing an ado file and at some points you need to create a variable 1/ln(x) in order to do some other calculation. The name of this variable is not important as you want to drop it later. But you cannot give it any name, because you do not know whether the user will already have a variable with this name…So you need to create a temporary variable:

```
tempvar invlogx
gen `invlogx'=1/(ln(`x')
....

drop `invlogx'
```

This is really similar to creating a local macro invlogx (this is in fact what stata does). In the above do file, we did not have to bother about dropping invlogx, as when the program ends, any temporary variables left in the data are automatically dropped.

* temporary data destruction

For example, in a middle of your program you want to use the collapse command to generate some means that you want to put back into your original data. So you could save the data, collapse, save the new data and append them to the old data. The problem is that the final users of your program may have a different setup that yours on his computer or some files with the name that you want to use… So there is a way to temporarily destroy some data (this does not completely sort our problem, see next for the complete answer…)

```
program define stuff

      preserve
      drop _all
      set obs 2
      gen x=1
      describe
end

stuff
describe
```

See that despite you dropping all the variables in the stuff program, they are still there afterwards.

In this case, the original data was restored after we close the program. Within a do file you can also use the preserve command, to return to your original data; use restore.

* Temporary files
This is pretty unusual, but if you need to create a temporary file, you can use;

```
tempfile newfile
…
save `newfile’
```

C] parsing: syntax

We want to create a program displaying median and iqr. Say we have written something like the following:

```
capture program drop iqr
program define iqr

quietly su `1', detail
display in green "`1'"                            /*
      */ _skip(10) "obs = " r(N)          /*
      */ _skip (10) "median = " r(p50)    /*
      */ _skip (10) "iqr = " r(p75)-r(p25)
end
```

which is OK when we want to display only 1 variable, adding a while loop allows us to run for more than one variable at a time.

It seems to work OK, but they are a few annoying bugs with it.

Try:
iqr – we would like to display all variables as is possible with other stata commands
iqr wei – we would like stata to display weigth rather than wei
iqr m* - we would like to display all variables starting with m.

However by adding 2 lines at the beginning of the program, we can cure all these drawbacks. Note also that I changed the display statement, so that all displays are perfectly in line.

```
capture program drop iqr
program define iqr

syntax [varlist]
tokenize "`varlist'"

while "`1'" ~="" {

      quietly su `1', detail
      display in green "`1'"                            /*
            */ _col(10) "obs = " r(N)           /*
            */ _col(25) "median = " r(p50)      /*
            */ _col(45) "iqr = " r(p75)-r(p25)

      mac shift
      }

end
```

Syntax defines what your program expects in terms of stata input (here you state that iqr will require a list of variables to work.
Tokenize reads the local varlist and puts each component (separated by blanks) in a local macro. So the first variable will be put in local 1, the second in local 2 and so on….

Comment: stata also creates a `0' local that echoes everything typed by the user after the command name.
*- By **not specifying** an element on the syntax line, you state that this element is **not allowed.***
- By specifying an element in [], you state that it is optional
- By specifying an element, you make it a requirement for the user to provide this element.

When you type iqr, the default is to copy all the variables in `0'. You can change this default by typing: syntax [varlist(default=none)]. In this case, typing iqr, will not produce any output but will not result in stata crashing. This is seldom used.
When you type iqr wei here is what happens:
-syntax interprets wei as weight and copy it into the local varlist
when you type iqr m*:
-syntax interprets m* as all variables starting with m, find that mpg and make follow this criteria and copy them into varlist.
When nothing is typed, in stata language this is equivalent to all variables. This only works because we have written `syntax [varlist].` If we had typed `syntax varlist` without square brackets then the program will return an error message as at least one variable would be expected (see comments above).

Syntax allows other types of input: varlist, varname, newvarlist or newvarname, which can be followed by restrictions in ();

min=#	you specify the minimum number of input
max=#	you specify the maximum number of input
numeric	the variables must be of the numeric type
string	the variables must be of the string type
ts	time series variables are allowed
generate	for use when newvarlist (varname) is specified

more specifiers can be added:

if	define whether if is allowed
in	define whether in is allowed
=exp	
using	
fw aw pw iw	defined which weights are allowed

Explain the difference between the various following uses of syntax:

syntax varlist
syntax [varlist]
syntax varlist (min=2)
syntax varlist (min=2 max=4 numeric)

If we were to write the syntax of some stata commands,

Summarize:	syntax [varlist] [aw] [fw] [if] [in]
Tabulate:	syntax varlist (max=2) [fw] [if] [in]
Generate:	syntax newvarname = exp [if] [in]
Regress	syntax varlist [aw] [fw] [iw] [if] [in]

for example, at the moment if we type;

```
. iqr mpg if foreign
if not allowed
r(101);
```

by changing to : `syntax [varlist] [if]`
we can now get a result when the user specifies or do not specify if (you always want to have if in [ ]), in this case: `iqr mpg if foreign`, the local if contains “if foreign” , however, we have only made the use of if available but we have not change the core of the program to take care of this option. There are two methods to do so.

* Change program to allow for if statement.

Here it is relatively straightforward, as the analysis is performed in a single line, so if we change the line to:

```
      quietly su `1' `if’, detail
```

when stata reads this line, the local if will be either empty or equal to if foreign, which is what we want.

Adding `if to each line works fine, when the analysis is conducted on a few lines and the lines do not include if statement themselves, when the do file gets longer, it is easy to forget to alter one line and get the wrong results.

* Difficulty 1

The user can sometimes use “” in his input but this may create problem to your syntax:
The user has typed if sex==”male” rather than if sex==1. In that case, the line
`display “tt_1 value = 15 `if’ ”` may read like:

```
      display “tt_1 value = 15 if sex==”male””
```

This may look obvious to you , but stata has a simple logic where the first “ open and the second one closes” , therefore the line reads like:

```
            display “tt_1 value = 15 if sex==”
                  male
                  ””
```

which will return an error message…
The way to get around that problem is to state which “ is an opening one and which “ is a closing one. This is done by adding ` and ‘. So your program should have been:

```
display `“tt_1 value = 15 `if’ ”’
```

Comment: This is utterly confusing, but if you write a program for yourself, you don’t need to ever use it, as long as you stick to the rule of not referring to values by their labels.

* Difficulty 2:

within your program you have lines like:

```
reg `yvar’ `xvar’ if `zvar’~=`.’
```

You also want the users to have the freedom to specify if for the all do file, so you want your syntax to be of the form:

```
syntax varlist [if]
…
reg `yvar’ `xvar’ if `zvar’~=`.’ &`if’
```

However, this is does not work. Say that the user wants to restrict the analysis to women only, so he has typed `myprog <varlist> if gender==0`.
Syntax then put if gender== 0 in the macro `if’. Then when your program reaches the line

```
reg `yvar’ `xvar’ if `zvar’~=`.’ & if gender==0
```

will generate an error because you cannot have two if statement in the same line. You could use preserve and restore, or use any of the following two approaches.

Solution 1:

Change the syntax line to:

```
syntax varlist [if/]
```

/ works like everything before / does not get into the local, so if your syntax is [if/] then, `if’ contains gender==0. All you need to do, is to modify, the reg line to:

```
reg `yvar’ `xvar’ if `zvar’~=`.’ & (`if’)
```

The () are important, cause the if statement may be complicated, consider something like if educ==1 | educ==2

However, we have now created an extra problem. If the user does not state if, we are left with the statement:

```
reg `yvar’ `xvar’ if `zvar’~=`.’ & ()
```

which is a syntax error.

The final version of this program should therefore look like:

```
syntax varlist [if/]

if “`if’” ~=”” {
     local andif “& (`if’)”
     }
```

```
reg `yvar’ `xvar’ if `zvar’~=`.’ `andif’
```

Solution 2:
Solution 1 was a lot of fiddling around with the program, this is somehow more direct, but maybe less clear at first.

All we do here is to create a temporary variable touse that is equal to 1 for the sample we want and 0 otherwise. The program looks like:

```
syntax varlist [if]

tempvar touse
quietly gen `touse’ =0
quietly replace `touse’ =1 `if’

reg `yvar’ `xvar’ if `zvar’~=`.’ & `touse’
```

In fact this is so useful, that statacorp has simplified this procedure to:

```
syntax varlist [if]

tempvar touse
mark `touse’ `if’ `in’        <- in case you also want to take care of in

reg `yvar’ `xvar’ if `zvar’~=`.’ & `touse’
```

This does not appear to be much shorter, but the mark command is much faster than generate and replace.

Even better and simpler, since the introduction of stata 6, all you have to do is to mark your sample just after your syntax line with a special command, so that your program is:

```
syntax varlist [if]

marksample `touse’
reg `yvar’ `xvar’ if `zvar’~=`.’ & `touse’
```

Can’t get simpler than that?

* alternatively: use preserve and restore;

The second solution is to keep the core of the program as it originally was, but limit the analysis to the subpopulation defined by the if statement.

So after the tokenise statement you type something like:

```
if "`if'"~="" {
      preserve
      quietly keep `if'
      }
```

You may or may not type restore at the end of you do file to restore the original data.
So now try: iqr mpg if mpg>20

Comments, in an ado file that may be used by other users, you may alter the syntax to: `if `"`if'"~=""' {` *. The single quote outside the "" ensure that if the users type something like if foreign==”foreign” stata interprets the syntax correctly.*

* Parsing options

Numerous stata commands have options (anything after the comma) like details for summarize…., here are some explanations on how to parse options.

Say we want to write the syntax of summarize where the option detail is available:
syntax [varlist] [aw] [fw] [if] [in] [,detail]

Comments: Note that the comma is inside [], the users do not have to specify the detail option.
You can have more than one option:
syntax [varlist] [aw] [fw] [if] [in] [,detail beta]

after parsing the macro `detail’ will contain “detail” or “”, the `beta’ will contain “beta” or “”. 0, 1 or the 2 options may be specified

Then typically part of your code is going to look like:

```
if “`detail”~=”” {
      do some stuff
      }
```

You can specify the minimum abbreviation of the options by capitalisation:
syntax [varlist] [aw] [fw] [if] [in] [,DEtail BEta]

* Parsing arguments

sometimes commands need argument (like in qreg), 6 different arguments are possible:

```
syntax [varlist] [,one (integer 1) two(real 3.14) three(string) four(varlist) /*
*/                 five (numlist) six(passthru)
```

option one(). If one is specified, its containt has to be an integer, if one is not specified then by default it will be set to 1.
You do not have to specify a default for options, one(integer) is perfectly valid

Option two(), is similar to one, but the value entered can be a real.

Option three() allows a string argument. So anything goes: numbers and variable names or any other type of information will be copied into `three’.

Option four() is similar to three but allows only a varlist

Option five() is similar to three but only numbers are allowed (possibly a list of numbers separated by space)

Option six() allows the use of names and ().
For example to parse myprog ….. , saving(mygph , replace) , the option saving needs to be define as passthru.

You can also use *, whatever option is typed by the user will be put in `option'. This is useful when the command you are creating has to allow for options available to some other commands that you need in you program
For example, if your program uses the graph command, you want to allow the users to specify options associated with graph. Rather than typing them all, you can use the * in your syntax line. In your program, you will have a line like: graph ….., `option'

D] Tokenize and gettoken

*Tokenize

Tokenize brakes down an input, and creates local macros containing part of this input. This is called parsing. I previously told you that the parsing was based on blanks, this is the default option in stata, it is however possible to modify it.

`tokenize “<whatever>” , parse(“+”)`   <- parsing is based on +

so if whatever contains “this is”   `1’ contains `this is’
if whatever contains “2+3*6   `1’ contains 2 and `2’ contains 3*6

your parsing can be done on more than one character

`tokenize “<whatever>” , parse(“  +,”)`   <- parsing is based on space + and ,

if whatever is “this is,2+3 *6”   `1’ is this, `2’ is, `3’ 2, `4’ 3 and `5’ *6

* Gettoken

gettoken is used only when your parsing is non-standard and cannot be dealt with by tokenize. Gettoken helps you to break the parsing line into sub-blocks that can be dealt with tokenize.
For example: mycmd 3 earnings age if sex==”female”, follow
Here you are in trouble because the parsing line starts with a number.

The syntax of gettoken is the following:

gettoken <newmac>   : <oldmac>   [,<options>]
or
gettoken <newmac>   <oldmac>   : <oldmac>   [,<options>]

where <oldmac> and <newmac> are macro names.

Gettoken looks at <oldmac> obtain the next element (token) from it and stores it in <newmac>. In the second syntax, gettoken also takes what is left of <oldmac> and uses it to replace <oldmac> whether in the first syntax, the contant of <oldmac> remain unchanged.

So lets got through that with some details:

Consider that `0’ contains 3 earnings age if sex==”female”, follow

By coding:

```
gettoken number      : 0
```

you obtain:

in `0’: 3 earnings age if sex==”female”, follow
in `number’: 3

By coding:

```
gettoken number    0  : 0
```

you obtain:

in `0’: earnings age if sex==”female”, follow
in `number’: 3

By default, tokens are identified by space, but the parse option can change that (as in tokenize).
Gettoken differs from tokenize in that is splits off only the first token (not the entire string) and gettoken puts the results where you specify it rather than in (`1’, `2’,….).
gettoken is mostly used to get rid off the non standard element of the parsing, then you can use syntax and tokenize.

For example, let’s pretend you want to parse something like:

Prog2 # [#] [varlist] [if] [,Constant]
Where # refers to number.

Basically, you want to parse separately the first number, check whether there is a second number and then the rest of the parsing is classic and can be deal with by syntax.

```
gettoken num1 0 : 0

capture confirm number `num1’
if _rc==0 {
        gettoken num2 varlist:0
}

tokenize `varlist’
```

so if the user type something like 3 2 wage education, this is place into `0’

after the first gettoken
`num1’ contains 3
`0’ is now 2 wage education

after the second gettoken
`num2’ contains 2
`varlist’ contains wage education

`varlist’ is now standard so you can parse it with tokenize

Congratulations, you are now a stata expert….