Lab 2 Supplement: Programming in Stata
Short Course on Poverty & Development for Nordic Ph.D. Students
University of Copenhagen
June 13-23, 2000

To take full advantage of the strengths of Stata, it's work taking some time to learn the basics of programming. In the labs, we will primarily use programming for looping through the data. But you can also write programs to estimate models or to create your own commands (see Lab 3).

1. Defining, running and dropping programs

Let's just start with a very simple program to see how they work. Suppose you type the following three lines in the Stata Command window (you can also submit them in a '.do' file),

```
program define hello
 display “Hello!”
end
```

and then type `hello` . Stata will respond with

```
. Hello!
```

There. You just wrote and ran a progam in Stata. If you type `hello` again, Stata again will respond with

```
. Hello!
```

So until you tell Stata to drop the program, or until you exit the program, Stata will keep it in memory (as it does with matrices, scalars, and saved results from built in commands).

Now type,

```
program define hello
 display “Greetings!”
end
```

and then type `hello` . Stata will respond with

```
. Hello!
```

The reason that it doesn't respond with `Greetings!` is that the previous version of the program is still in memory. Now if you type

```
program drop hello
program define hello
 display “Greetings!”
end
```

and then type `hello` . Stata will respond with

```
. Greetings!
```

Note that Stata does not execute commands like `display` when they appear in a program until the program is invoked. Sometimes this makes debugging programs (I don’t know anyone who is able to get all of his/her programs doing what he/she wants on the first try) difficult. It especially becomes difficult if you forget to drop the previously saved version of your program. A common way around this is to type

```
capture program drop progname
```

before you define your progam. This tells Stata to recognize new versions of the program and to drop the version in memory if another is defined with the same name. Now you will get the following

```
capture program drop hello
program define hello
 display “Hello!”
end

hello

. Hello!

capture program drop hello
program define hello
 display “Greetings!”
end

hello

. Greetings!
```

2. Program arguments and locals

Suppose that I want to write a program that will find the mean of a variable which I’ll identify when I run the program (this is already a built in program in Stata, but we’ll just use this redundant program as an example). There are a couple of things that we need to learn about to do this. First, we’ll look at a program that does this, and then we’ll see what’s going on (note: you can’t give your program the same name as existing programs because there's a hierarcy of how Stata looks for programs – and user-written ones are at the bottom of the list – in other words, Stata will apply the built in command)

```
capture program drop mysum
program define mysum
  local var `1'
  summarize `var'
end
```

```
mysum haz
```

you will get the following result (given a dataset that has the variable "haz" in it),

```
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
     haz |   68561   -1.472182   1.410618      -5.89       2.98
```

I can also type

```
mysum whz
```

and Stata will respond with

```
Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
     whz |   68560    -.458337   1.192863         -4          6
```

The first thing to note is that we defined a "local" which is a variable or scalar (we can also define local matrices) that exists solely within the program. Once the program is finished running, Stata drops all locals from memory. We can define locals by using the following syntax

```
local name targetname
```

for variables where `targetname` is the name of the variable who's values are being assigned to the local variable, or in the case of scalars,

```
local name = targetname
```

Note that in the fourth line when the local is being called, the name is put in paretheses (these are ` and ' , respectively)

```
`var'
```

This tells Stata to look for the local with the name `var`, and to ignore existing variables or scalars with the same name. Note that when you define a local, you do not include the ` and ' around it.

What then is the `` `1’ ``? This tells Stata to use the first argument of the program (here we only have one argument – e.g. `mysum whz`). Of course, we could have written the program as follows,

```
capture program drop mysum
program define mysum
  summarize `1’
end
```

but when your programs become more complicated, you will find that it is much easier to to keep track of what you are doing if you give names to your locals.

3. Do loops

Now suppose that we want to find the means of this variable for each value of another indicator variable, for example by region, then we can modify our program to include a loop (again you can do this by simply typing `by urban: sum haz`).

```
capture program drop mysum
program define mysum
   local var `1'
   local byvar `2'
   quietly sum `byvar'
   local max = r(max)
   local i = r(min)

    while `i' <= `max' {
      sum `var' if `byvar' == `i'
      local i = `i' + 1
                       }
end
```

Here is what it looks like in the Stata Results window:

```
. capture program drop mysum

. program define mysum
  1. local var `1'
  2. local byvar `2'
  3. quietly sum `byvar'
  4. local max = r(max)
  5. local i = r(min)
  6. while `i' <= `max' {
  7. sum `var' if `byvar' == `i'
  8. local i = `i' + 1
  9. }
 10. end
```

```
. mysum haz urban

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
     haz |   48805   -1.594447    1.41825      -5.84       2.89

Variable |     Obs        Mean   Std. Dev.       Min        Max
---------+-----------------------------------------------------
     haz |   19756   -1.170138   1.344784      -5.89       2.98
```

The novelty here is the following,

```
    while `i' <= `max' {
      ..commands..
      local i = `i' + 1
                       }
```

This tells Stata to carry out the commands within the curly brackets, { }, until the `while` condition is satisfied. Note that the local `` `i' `` is augmented in the following fashion

```
      local i = `i' + 1
```

This says to define the local `i` again, but now assign it's present value (i.e. `` `i’ ``) and add `1` to it. The following will result in errors

```
      local `i’ = `i' + 1
      local i = i + 1
```

(except for the second line if you have a non-local variable or scalar with the name `i` – which, of course, is not what you want anyway).

With these basic programming commands under your belt, you should have no problem understanding the programs that we'll encounter in Labs 2, 3 and 4.