SUGUK 2005 invited lecture: *A little bit of Stata programming goes a long way...*

Christopher F Baum[1]
Boston College
baum@bc.edu
http://ideas.repec.org/e/pba1.html

June 8, 2005

[1]I am deeply indebted to Nicholas J. Cox for his lucid treatment of many of these topics in his *Speaking Stata* columns in Vol. 1–3 of the *Stata Journal*. I am grateful to Petia Petrova, William Gould and Roberto Gutierrez for useful comments.

Abstract

This tutorial will discuss a number of elementary Stata programming constructs and discuss how they may be used to automate and robustify common data manipulation, estimation and graphics tasks. Those used to the syntax of other statistical packages or programming languages must adopt a different mindset when working with Stata to take full advantage of its capabilities. Some of Stata's most useful commands for handling repetitive tasks: `forvalues, foreach, egen, local, scalar, estimates` and `matrix` are commonly underutilized by users unacquainted with their power and ease of use. While relatively few users may develop ado-files for circulation to the user community, nearly all will benefit from learning the rudiments of use of the `program, syntax and return` statements when they are faced with the need to perform repetitive analyses. Worked examples making use of these commands will be presented and discussed in the tutorial.

Overview

Users of Stata often do more work—in some cases much more work—than necessary. Those familiar with other statistical packages' syntax or with computing in a procedural language in which loops and subscripts play a key role will often transfer their understanding to Stata. In most (but not all) cases, you *can* do it that way in Stata: but it is far from efficient in terms of computer time, and above all is wasteful of both the user's time and gray matter. If the program can do the housekeeping work for you, and permit you to specify merely the pattern of the necessary effort, why not let it do so? I have titled this talk *A little bit of Stata programming goes a long way...* to emphasize the sizable return on investment to human capital of this nature. Learn how to use Stata effectively, and you will save yourself time and avoid the grief of retracing many steps through a less well-constructed research procedure.

First, an important caveat. What I have to say here will be of little use to those who rely on Stata's menu system and the point-and-click interface, unless it convinces them that they should learn to use do-files. There are many good things to be said for a menu-driven interface when one is learning to use a package: it is perhaps easier to examine a command's many options in a dialog rather than trying to assimilate the on-line help. But ultimately a

menu-driven interface is a far slower way to get your work done if you know what you're doing. It also runs counter to the basic principle here: every time you use Stata to perform a non-trivial task, you should be building up your stock of human capital in terms of knowing how to use the package most effectively. If you eschew the menu-driven interface and even the interactive command line in favor of the do-file approach to Stata programming, you will construct a script to solve today's problem. When you next need to solve that problem, no need to reinvent the wheel: just haul out the script and tweak it a bit. Eventually, after you alter the script a few times, you may recognize that it would be easier to turn it into a very simple program to perform the variations in that research task. And voila! Soon you will be saying "yesterday, I couldn't spell programmer, and today I are one."

A second caveat: many tasks with Stata are made all the easier with the `by:` prefix. In Stata 8, considerable effort could be saved with the use of [R] **statsby**. In Stata 9, this notion has been extended with the [R] **rolling** command. The particulars of `by:` usage, particularly in a panel context, has been exposited in Cox (2001), and for brevity will not be discussed here.

The outline of the talk: we will first discuss the use of the `local macro`, its cousin `global`, `scalar` and their interaction with Stata's up-to-date constructs for repetitive work: `foreach` and `forvalues`. Stata 9's `Mata` notwithstanding, we will consider how Stata's existing `matrix` constructs may be used not only for linear algebra, but rather to organize results for analysis and tabular presentation. The use of `estimates` to minimize retyping of estimated results, coupled with Ben Jann's `estout` suite, provide further labor-saving and error-minimizing opportunities. Finally, for those who might take the plunge and package their sequences of commands in a slightly more general form: that of the Stata `program`, or `ado`-file, we will consider rudimentary use of that command and its sisters `syntax` and `return`.

Local and global macros

Those familiar with lower-level programming languages such as FORTRAN, C or Pascal may find Stata's terminology for various objects rather confusing. In those languages, one refers to a *variable* with statements such as `x = 2`. Although one may have to declare `x` before its use (for instance, as `integer` or `float`), the notion of a "variable" in those languages refers to an entity which can take on a single value (numeric or string). In contrast, the Stata *variable* refers to one column of the data matrix, and it may take

on `maxobs` values, one per observation.

So what corresponds to a FORTRAN or C *variable* in Stata's lingo? Either a Stata *macro* or a *scalar* (to be discussed below).[1] But that correspondence is not one-to-one, since a Stata *macro* may contain multiple elements: in fact, it may contain any combination of alphanumeric characters. The Stata macro is really an *alias* which has both a *name* and a *value*: when its name is *dereferenced*, it returns its value. That operation may be carried out at any time, or its value may be modified by an additional command. For an example of the first concept:

Exhibit 1

```
. local country US UK DE FR
. local ctycode 111 112 136 134
. display "‘country’"
US UK DE FR
. display "‘ctycode’"
111 112 136 134
```

The Stata command to define the macro is `local` (see [P] **macro**). A macro may be either *local* or *global*, which refers to its *scope*: where its existence will be recognized. A *local macro* is created in a do-file (or ado-file) and ceases to exist when that do-file terminates (normally or abnormally). A *global macro* exists for the duration of the Stata program or interactive session. There are good reasons to use global macros, but like any global definition, they may have unintended consequences. Consequently we will discuss local macros in most of the examples below.

Notice that the `local` command names the macro—as `country`—and defines its value to be the list of four biliteral country codes. The following statement does the same for macro `ctycode`. To bring forth the contents of the macro, we *dereference* it: the expression `‘macroname’` refers to the *value* of the macro. Notice that the macro's name is preceded by the *left tick* character (`‘`) and followed by the apostrophe (`’`). Most errors in the use of macros are caused by failure to adhere to this rule: to dereference the macro, the correct punctuation is vital. In the [P] **display** statement, we must wrap the dereferenced macro in double quotes, since `display` expects a

[1] Stata's scalars were purely numeric through version 8.0, as described in [P] **scalar**. The ability to store strings in scalars was added in the executable update of 1 July 2004.

string argument or the value of a scalar expression (e.g., `display log(14)`).

Notice that in both cases the `local` statement is written without an equals sign (=). It is acceptable syntax to use an equals sign following the macro's name, but it is a very bad idea to get in the habit of using it unless it is required. Why? Because the equals sign causes the remainder of the expression to be *evaluated*, rather than merely aliased to the macro's name. This is a common cause of head-scratching, where a user will complain "the do-file worked when I had eight regressors, but not when I had nine..." Defining a macro with an equals sign will cause *evaluation* of the remainder of the command as a character string, and in Intercooled Stata a character string (i.e., the contents of a string variable) cannot have more than 80 characters. In evaluating `local mybadstring = "This is an example of a string that will not all end up in the macro that it was intended to populate"` (with the quotation marks now required) the string will be truncated at 80 characters without error or warning.

When is it appropriate to use an equals sign in a `local` statement? When we require the macro's value to be evaluated. In this example, we show a macro used as a counter which fails to do exactly what we had in mind:

Exhibit 2

```
. local count 0
. local country US UK DE FR
. foreach c of local country {
  2.    local count `count'+1
  3.    display "Country `count' : `c'"
  4.    }
Country 0+1 : US
Country 0+1+1 : UK
Country 0+1+1+1 : DE
Country 0+1+1+1+1 : FR
```

In this case, we *must* use the equals sign to request evaluation (rather than concatenation):

Exhibit 3

```
. local count 0
. local country US UK DE FR
. foreach c of local country {
  2.    local count = `count'+1
  3.    display "Country `count' : `c'"
  4.    }
Country 1 : US
```

```
Country 2 : UK
Country 3 : DE
Country 4 : FR
```

Notice that the `local` statement contains the name of the macro twice: without punctuation, which defines its name, and on the right hand side of the equals, with its current value dereferenced via `‘count’`. It is crucial to understand why the statement is written this way: (re-)defining the object in the first instance, and referencing its current value in the second.

As nonsensical as the flawed example above might be, there are instances where we wish to construct a macro within a loop (i.e., repeatedly redefining its value), and decidedly *must* avoid the equals sign:

Exhibit 4

```
. local count 0
. local country US UK DE FR
. foreach c of local country {
  2.    local count = ‘count’+1
  3.    local newlist "‘newlist’ ‘count’ ‘c’"
  4.    }
. display "‘newlist’"
 1 US 2 UK 3 DE 4 FR
```

The `local newlist` statement is curious: it defines the local macro `newlist` as a string containing its own current contents, space, *value-of-count*, space, *value-of-c*. Without fully explaining the [P] **foreach** statement at this point, understand that it creates the local macro `c` with the value of each biliteral country code in turn. The first time through the loop, `newlist` does not exist: so how may we refer to its current value? Easily: every Stata macro has a null value unless it has explicitly been given a non-null value. Thus, it takes on the string `" 1 US"` the first time, and then the second time through concatenates that string with the new string `" 2 UK"`, and so on. In this example, use of the equals sign would be inappropriate in the `local newlist` statement, since it would cause truncation of `newlist` at 80 characters: a problem with a longer list of countries.

From these examples, we might conclude that Stata's macros are useful in constructing lists, or as counters and loop indices. They are that: but they play a much larger role in Stata do-files and ado-files, and indeed in the return values of virtually all Stata commands, as we discuss below. Macros

are the one of the key elements which allow the Stata user to avoid repetitive commands and the retyping of computed results. For instance, the macro defined by `local country US UK DE FR` may be used to generate a set of graphs with country-specific content and labels:

Exhibit 5

```
. local country US UK DE FR
. foreach c of local country {
  2.         tsline gdp if cty=="`c'", title("GDP for `c'")
  3.         }
```

or even to produce a single graph with panels for each country:

Exhibit 6

```
. local country US UK DE FR
. foreach c of local country {
  2.         tsline gdp if cty=="`c'", title("GDP for `c'") ///
> 
  3.         }
. graph combine `country', ti("Gross Domestic Product, 1971Q1-1995Q4")
```

A particular benefit of this use of macros is the ability to change the performance of the do-file by merely altering the contents of the local macro. To produce these graphs for a different set of countries, we need merely to alter one command: the list of codes. In this manner, our do-file can be made quite general, and that set of Stata commands may be reused or adapted for use in similar tasks with a minimum of effort.

Global macros

Global macros are distinguished from local macros by their manner of creation (via the `global` statement) and means of reference. We may refer to global macro `george` as `$george`, with the dollar sign taking the place of the punctuation surrounding the local macro's name when it is dereferenced. Global macros are often used to store items parametric to a program, such as a character string with today's date that is to be embedded in all filenames created by the program, or the name of a default directory.

Generally, unless there is an explicit need for a global macro (that is, a symbol with *global scope*), it is preferable to use a local macro. It is easy to forget that a particular symbol was defined in do-file *A*, and having that

definition still operative in do-file G or H may wreak havoc—and be quite difficult to track down. The same good programming practices that authors of FORTRAN or C programs are exhorted to follow: "keep definitions local unless they must be visible outside the module" is good advice for Stata programmers as well.

Extended macro functions and list functions

Stata contains a versatile library of functions that may be applied to macros: the *extended functions* (`help extended_fcn`, or [P] **macro**). These functions allow the contents of macros to be readily manipulated:

Exhibit 7

```
. local country US UK DE FR
. local wds: word count `country'
. display "There are `wds' countries:"
There are 4 countries:
. forvalues i = 1/`wds' {
  2.         local wd: word `i' of `country'
  3.         display "Country `i' is `wd'"
  4.         }
Country 1 is US
Country 2 is UK
Country 3 is DE
Country 4 is FR
```

In this example, we use the `word count` and `word # of` extended functions, both of which operate on strings. Note that we do not enclose the macro's value (`` `country' ``) in double quotes, for it then would be considered a single "word".

A wide variety of extended functions perform useful tasks such as extracting the variable label or value label from a variable, or determining its data type or display format; extracting the row or column names from a Stata matrix; or generating a list of the files in a particular directory that match a particular pattern (e.g., `*.dta`). The handy `subinstr` function allows a particular pattern to be substituted in a macro, either the first time it is encountered or in all instances.

Another very useful set of functions support the manipulation of lists held in local macros. These functions, described in `help macrolists` or [P] **macro lists**, may be used to identify the unique elements of a list, or the duplicate entries; to sort a list; and to combine lists with Boolean operators

such as AND, OR. A set of handy list functions allow one list's contents to be "subtracted" from another, identifying the elements of list A that are not duplicated in list B, and to test lists for "equality" (defined for lists as containing the identical elements in the same order; an alternate form tests for "weak equality", which does not consider ordering). A list function (`posof`) may be used to determine whether a particular entry exists in a list. An excellent discussion of many of these issues may be found in Cox (2003).

Scalars

The distinction between macros and Stata's *scalars* is no longer numeric content, since both macros and scalars may now contain string values. However, the length of a string scalar is limited to the length of a string variable (80 in Intercooled Stata, 244 in Stata/SE: [R] **limits**), whereas a macro's length is for most purposes unlimited (actually a finite 67,784 characters in Intercooled Stata, and over one million in Stata/SE). For most purposes, the real relevance of Stata's scalars is in their use in a numeric context. When a numeric quantity is stored in a macro, it must be converted from its internal (binary) representation into a printable form. That conversion is done with maximum accuracy, but incurs an overhead, particularly if the numeric quantity is non-integer. By storing the result of a computation (e.g., the mean or standard deviation of a variable) in a scalar, no conversion of its value need take place, and the result is held in Stata's full numeric precision. For this reason, most of Stata's statistical and estimation commands return various numeric results as scalars (as we discuss below). A scalar may be referred to in any Stata command by its name:

Exhibit 8

```
. scalar root2 = sqrt(2.0)
. gen rootGDP = gdp*root2
```

Thus, the distinction between a macro and a scalar appears when it is referenced: the macro must be dereferenced to refer to its value, while the scalar is merely named.[2] However, a scalar can only appear in an expression where a Stata variable could be used. For instance, one cannot specify a scalar as part of an `in` *range* qualifier, since its value will not be extracted. It may

[2]Stata is capable of working with scalars of the same name as Stata variables. As the manual suggests, Stata will not become confused, but you well may. Avoid using the same names for both entities!

be used in an `if` *exp* qualifier, since that contains a numeric expression.

Stata's scalars may play a useful role in a complicated do-file. By defining scalars at the beginning of the program and referring to them throughout the code, one may make the program parametric, and avoid the difficulties of changing various constants in the program's statements in each of the lines in which they appear. A researcher often needs to repeat a complex data generation task for a different category: e.g. 18-24 year old subjects rather than 25-39 year old subjects, with the qualifiers for minimum and maximum age appearing throughout the program. By defining those age limits as scalars at the program's outset, altering its function is greatly simplified, and reliability is enhanced.

Loop constructs

One of Stata's most powerful features is the ability to write a versatile Stata program without a large number of repetitive statements. Many Stata commands contribute to this parsimony of expression: for instance, the features of [R] **egen**, particularly with a `by` option, make it possible to avoid many explicit statements such as (in pseudo-code) `compute mean of age for race==1; compute mean of age for race==2;` etc. But two of Stata's most useful commands are not to be found in the *Reference Manual*: [P] **forvalues** and [P] **foreach**. Even in the Stata 9 documentation, these commands' full descriptions are to be found in the *Programming Manual*, so if you're going to make heavy use of the constructs discussed in this talk, that manual is well worth acquiring. These versatile tools have essentially supplanted other mechanisms in Stata for looping. One may use [P] **while** to construct a loop, but you must furnish the counter yourself (presumably with a local macro). The command `for` is now deprecated and is no longer described in the manuals: for good reason, as `for` only allowed a single command to be included in a loop structure (or multiple commands with a tortured syntax) and rendered nested loops almost impossible.

In contrast, the [P] **forvalues** and [P] **foreach** commands use a syntax familiar to users of C or other modern programming languages: the command is followed by a left brace ({), one or more following command lines, and a terminating line containing only a right brace (}). In Stata 8 and 9, that separation of the braces from the "body" of the loop is mandatory. You may place as many lines in the "loop body" as are needed. A simple numeric loop may thus be constructed as:

Exhibit 9

```
. forvalues i = 1/4 {
  2.         gen double lngdp‘i’ = log(gdp‘i’)
  3.         summ lngdp‘i’
  4.         }
```

Variable	Obs	Mean	Std. Dev.	Min	Max
lngdp1	400	7.931661	.59451	5.794211	8.768936

Variable	Obs	Mean	Std. Dev.	Min	Max
lngdp2	400	7.942132	.5828793	4.892062	8.760156

Variable	Obs	Mean	Std. Dev.	Min	Max
lngdp3	400	7.987095	.537941	6.327221	8.736859

Variable	Obs	Mean	Std. Dev.	Min	Max
lngdp4	400	7.886774	.5983831	5.665983	8.729272

In this example, we define the local macro `i` as the loop index; following an equals sign, we give the `range` of values that `i` is to take on. A range may be as simple as `1/4`, or `10(5)50`, indicating 10 to 50 in steps of 5; or `100(-10)20`, from 100 to 20 counting down by tens. Other syntaxes for the range are available; see [P] **forvalues** for details.

This example provides one of the most important uses of [P] **forvalues**: looping over variables where the variables have been given names with an integer component, which avoids the need for separate statements for each of the variables. The integer component need not be a suffix; we could loop over variables named `cty`*N*`gdp` just as readily given our understanding of local macros. Or, say that we have variable names with more than one integer component:

Exhibit 10

```
. forvalues y = 1995(2)1999 {
  2.         forvalues i = 1/4 {
  3.                 summ gdp‘i’_‘y’
  4.                 }
  5.         }
```

Variable	Obs	Mean	Std. Dev.	Min	Max
gdp1_1995	400	3226.703	1532.497	328.393	6431.328

Variable	Obs	Mean	Std. Dev.	Min	Max
gdp2_1995	400	3242.162	1525.788	133.2281	6375.105

Variable	Obs	Mean	Std. Dev.	Min	Max
gdp3_1995	400	3328.577	1457.716	559.5993	6228.302

Variable	Obs	Mean	Std. Dev.	Min	Max
gdp4_1995	400	3093.778	1490.646	288.8719	6181.229
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp1_1997	400	3597.038	1686.571	438.5756	7083.191
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp2_1997	400	3616.478	1677.353	153.0657	7053.826
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp3_1997	400	3710.242	1603.25	667.2679	6948.194
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp4_1997	400	3454.322	1639.356	348.2078	6825.981
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp1_1999	400	3388.038	1609.122	344.8127	6752.894
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp2_1999	400	3404.27	1602.077	139.8895	6693.86
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp3_1999	400	3495.006	1530.602	587.5793	6539.717
Variable	Obs	Mean	Std. Dev.	Min	Max
gdp4_1999	400	3248.467	1565.178	303.3155	6490.291

As we see here, a nested loop is readily constructed with two [P] **forvalues** statements.

foreach

As useful as [P] **forvalues** may be, the [P] **foreach** statement is even more generally useful in constructing efficient do-files. This command interacts perfectly with some of Stata's most common constructs: the macro, the *varlist* and the *numlist*. Like [P] **forvalues**, a local macro is defined as the loop index, but rather than cycling through a set of numeric values, [P] **foreach** specifies that the loop index iterates through the elements of a local (or global) macro, or the variable names of a *varlist*, or the elements of a *numlist*. The list can also be an arbitrary list of elements on the command line or a *newvarlist*, the elements of which must be valid names for variables not present in the data set.

This syntax allows [P] **foreach** to be used in a very flexible manner with any set of items, irregardless of pattern. In several of the examples above, we employed [P] **foreach** with the elements of a local macro defining the list. We

illustrate its use with a *varlist* with the `lifeexp` *Reference Manual* data set, computing summary statistics, correlations with `popgrowth` and generating scatterplots for each element of a *varlist* versus `popgrowth`:

Exhibit 11

```
. foreach v of varlist lexp-safewater {
  2.         summ `v'
  3.         correlate popgrowth `v'
  4.         scatter popgrowth `v'
  5.         }
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
        lexp |        68    72.27941    4.715315         54         79
(obs=68)
             | popgro~h     lexp
-------------+------------------
  popgrowth  |   1.0000
       lexp  |  -0.4360   1.0000
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
       gnppc |        63    8674.857    10634.68        370      39980
(obs=63)
             | popgro~h    gnppc
-------------+------------------
  popgrowth  |   1.0000
      gnppc  |  -0.3580   1.0000
    Variable |       Obs        Mean    Std. Dev.       Min        Max
-------------+--------------------------------------------------------
  safewater  |        40        76.1    17.89112         28        100
(obs=40)
             | popgro~h safewa~r
-------------+------------------
  popgrowth  |   1.0000
  safewater  |  -0.4280   1.0000
```

In the following example, we automate the construction of a [R] **recode** statement. The resulting statement could just be typed out for four elements, but imagine its construction if we had 180 country codes! Note the use of `local ++i`, a shorthand way of incrementing the counter variable within the loop.[3]

[3]The really serious Stata programmer would avoid that line and write the following line as `local rc "`rc' (`=`++i''=`c')"`, but the rest of us may have some trouble with that.

Exhibit 12

```
. local ctycode 111 112 136 134
. local i 0
. foreach c of local ctycode {
  2.         local ++i
  3.         local rc "`rc' (`i'=`c')"
  4.         }
. display "`rc'"
 (1=111) (2=112) (3=136) (4=134)
. recode cc `rc', gen(newcc)
(400 differences between cc and newcc)
. tab newcc
```

RECODE of cc	Freq.	Percent	Cum.
111	100	25.00	25.00
112	100	25.00	50.00
134	100	25.00	75.00
136	100	25.00	100.00
Total	400	100.00	

The [P] **foreach** statement can also be used to advantage with nested loops. One may combine [P] **foreach** and [P] **forvalues** in a nested loop structure, as illustrated here:

Exhibit 13

```
. local country US UK DE FR
. local yrlist 1995 1999
. forvalues i = 1/4 {
  2.         local cname: word `i' of `country'
  3.         foreach y of local yrlist {
  4.                 rename gdp`i'_`y' gdp`cname'_`y'
  5.                 }
  6.         }
. summ gdpUS*
```

Variable	Obs	Mean	Std. Dev.	Min	Max
gdpUS_1995	400	3226.703	1532.497	328.393	6431.328
gdpUS_1999	400	3388.038	1609.122	344.8127	6752.894

It is a good idea to use indentation (either spaces or tabs) to align the loop body statements as shown here; although Stata does not care (as long as the braces appear as required) it makes the do-file much more readable and easier to revise at a later date.

In summary, the [P] **foreach** and [P] **forvalues** statements are essential

components of any do-file writer's toolkit. Whenever you see a set of repetitive statements in a Stata do-file, it is likely to mean that its author did not understand how one of these loop constructs could have made the program (and its upkeep) simpler. An excellent discussion of the loop commands is to be found in Cox (2002).

Matrices

Stata has contained a full-featured matrix language, for the last several versions, and supports a broad range of matrix operations on real matrices, as described in [P] **matrix**. The big news with Stata version 9 is the addition of **Mata**, a matrix programming language which puts Stata on a par (or better!) with matrix languages such as MATLAB, GAUSS or Ox in terms of both capabilities and speed. Since most of us have yet to experience *The Joy of Mata*, I will confine my remarks to Stata 8 matrix functionality, which remains available in Stata version 9.

For those Stata users who are not developing their own ado-files, Stata matrices are likely to be useful in two particular contexts: that of saved results, and as a way of organizing information for presentation. Most of Stata's statistical and estimation commands generate one or more matrices "behind the scenes", For instance, [R] **regress** (like *all* Stata estimation commands) produces matrices `e(b)` and `e(V)` as the row vector of estimated coefficients (a $1 \times k$ matrix) and the estimated variance-covariance matrix of the coefficients (a $k \times k$ matrix), respectively. One may examine those matrices with the `matrix list` command or copy them for use in your program with the `matrix` statement: e.g., `matrix beta = e(b)` will create a matrix `beta` in your program as a copy of the last estimation command's coefficient vector. Stata matrices are rather unique in that their elements may be addressed both conventionally, with row and column numbers (counting from 1, not 0) and by their row and column names: `mat vv = v["gdp2","gdp3"]` will extract the estimated covariance of the coefficients on `gdp2` and `gdp3`. Note that references to matrix elements appear in square brackets. All Stata matrices have two subscripts; there is no vector data type, so that both subscripts must be given in any reference. A range of rows (columns) may be specified in an expression; see [P] **matrix** for details.

Stata's matrices are often useful devices for housekeeping purposes: for instance, for the accumulation of results that are to be presented in tabular

form.[4] Just as [R] **tabstat** may generate descriptive statistics for a set of by-groups, `statsmat` (Cox and Baum, available via `ssc`) can be used to generate a matrix of descriptive statistics for a set of variables, or a single variable over by-groups. Stata matrices have row and column labels, and those labels may be manipulated by `matrix rownames`, `matrix colnames` and several macro extended functions (described previously). This allows control of the row and column headings on tabular output in a quite flexible way. Stata's matrix operators make it possible to assemble a matrix from several submatrices: for instance, one matrix for each country in a multi-country sample. In summary, judicious use of Stata's matrices ease the burden of many housekeeping tasks, and make it feasible to update material in tabular form without retyping.

`return` and `ereturn`

Each of Stata's commands reports its results: sometimes noisily, as when a non-zero return code is accompanied by an error message (`help _rc`), but usually quite silently. The user may be unaware of the breadth and usefulness of the results made available for further use by Stata commands. Raising one's awareness of this facility can greatly simplify one's work with Stata, since a do-file may be constructed to use the results of a previous statement in a computation, title, graph label, or even in a conditional statement.

We must distinguish between *r-class* and *e-class* commands. Each Stata command is classified in a class, which may be r, e, or (less commonly) s. This applies to both those commands which are *built-in* (like [R] **summarize** or [R] **regress**) and to the 80% of official Stata commands that are implemented in the ado-file language.[5] The e-class commands are Estimation commands, all of which must return `e(b)` and `e(V)` (the estimated parameter vector and its variance-covariance matrix, respectively) to the calling program, as well as other information (`help ereturn`). Almost all other official Stata commands are r-class commands which return Results to the calling program (`help return`). Let us deal first with the simpler case of r-class commands.

Virtually every command—including those which you might not think of as generating results—places items in the *return list*, which may be displayed by the command of the same name.[6] For instance, consider [R] **describe**:

[4] A good discussion of matrices for housekeeping is presented in Watson (2005).

[5] If this distinction interests you, [R] **which** will either report that a command is built-in (i.e., compiled code) or located in a particular ado-file on your disk.

[6] Significant exceptions: [R] **generate** and [R] **egen**.

Exhibit 14

```
. webuse abdata,clear
. describe
Contains data from http://www.stata-press.com/data/r8/abdata.dta
  obs:         1,031
 vars:            30                          3 Sep 2002 12:25
 size:       105,162 (99.8% of memory free)
```

variable name	storage type	display format	value label	variable label
c1	str9	%9s		
ind	float	%9.0g		
year	float	%9.0g		
emp	float	%9.0g		
wage	float	%9.0g		
cap	float	%9.0g		
indoutpt	float	%9.0g		
n	float	%9.0g		
w	float	%9.0g		
k	float	%9.0g		
ys	float	%9.0g		
rec	float	%9.0g		
yearm1	float	%9.0g		
id	float	%9.0g		
nL1	float	%9.0g		
nL2	float	%9.0g		
wL1	float	%9.0g		
kL1	float	%9.0g		
kL2	float	%9.0g		
ysL1	float	%9.0g		
ysL2	float	%9.0g		
yr1976	byte	%8.0g		year== 1976.0000
yr1977	byte	%8.0g		year== 1977.0000
yr1978	byte	%8.0g		year== 1978.0000
yr1979	byte	%8.0g		year== 1979.0000
yr1980	byte	%8.0g		year== 1980.0000
yr1981	byte	%8.0g		year== 1981.0000
yr1982	byte	%8.0g		year== 1982.0000
yr1983	byte	%8.0g		year== 1983.0000
yr1984	byte	%8.0g		year== 1984.0000

```
Sorted by:  id  year
. return list
scalars:
                 r(N) =  1031
                 r(k) =  30
             r(width) =  98
             r(N_max) =  494610
             r(k_max) =  5000
          r(widthmax) =  50848
           r(changed) =  0
. local sb: sortedby
. di "dataset sorted by : `sb'"
dataset sorted by : id year
```

The return list contains items of a single type: *scalars*. Note that `r(N)` and `r(k)` provide the number of observations and variables present in the data set in memory while `r(changed)` is an indicator variable that will be set to 1 as soon as a change is made in the data set. We also demonstrate here how information about the data set's *sort order* may be retrieved by one of the extended macro functions discussed earlier. Any of the scalars defined in the return list may be used in a following statement: note that the return list need not be displayed, as every [R] **describe** command will define this set of scalars. A subsequent r-class command will replace the contents of the return list with its return values, so that if one wants to use these items, they should be saved to local macros or named scalars. For a more practical example, consider [R] **summarize**:

Exhibit 15

```
. summarize emp, detail
                            emp
-------------------------------------------------------------
      Percentiles      Smallest
 1%         .142           .104
 5%         .431           .122
10%         .665           .123       Obs                1031
25%         1.18           .125       Sum of Wgt.        1031

50%        2.287                      Mean           7.891677
                        Largest       Std. Dev.      15.93492
75%        7.036         101.04
90%       17.919         103.129      Variance       253.9217
95%         32.4         106.565      Skewness       3.922732
99%         89.2         108.562      Kurtosis       19.46982

. return list

scalars:
                 r(N) =  1031
             r(sum_w) =  1031
              r(mean) =  7.891677013539667
               r(Var) =  253.9217371514514
                r(sd) =  15.93492193741317
          r(skewness) =  3.922731923543386
          r(kurtosis) =  19.46982480250623
               r(sum) =  8136.319000959396
               r(min) =  .1040000021457672
               r(max) =  108.5619964599609
                r(p1) =  .1420000046491623
                r(p5) =  .4309999942779541
               r(p10) =  .6650000214576721
               r(p25) =  1.179999947547913
               r(p50) =  2.286999940872192
               r(p75) =  7.035999774932861
               r(p90) =  17.91900062561035
               r(p95) =  32.40000152587891
               r(p99) =  89.19999694824219

. scalar iqr = r(p75) - r(p25)
```

```
. di "IQR = " iqr
IQR = 5.8559998
. scalar semean = r(sd)/sqrt(r(N))
. di "Mean = " r(mean) " S.E. = " semean
Mean = 7.891677 S.E. = .49627295
```

We invoke the `detail` option to display the full range of results available (in this case, all in the form of scalars) after the [R] **summarize** command. We compute the inter-quartile range of the variable and the standard error of the mean and display those quantities. We very often need the mean of a variable for further computations, but do not wish to display the results of [R] **summarize**; in this case, the `meanonly` option of [R] **summarize** both suppresses output and does not calculate the variance or standard deviation of the series, which are more computationally demanding than calculating the mean. With this option, the scalars `r(N), r(mean), r(min), r(max)` are still available.

When working with time series or panel data, it is often useful to know whether the data have been [TS] **tsset**, and if so, what variable is serving as the calendar variable and (if present) the panel variable. For instance:

Exhibit 16

```
. tsset
       panel variable:  id, 1 to 140
        time variable:  year, 1976 to 1984
. return list
scalars:
              r(tmax) =  1984
              r(tmin) =  1976
              r(imax) =  140
              r(imin) =  1
macros:
          r(panelvar) : "id"
           r(timevar) : "year"
             r(unit1) : "."
             r(tsfmt) : "%9.0g"
             r(tmaxs) : "1984"
             r(tmins) : "1976"
```

In this example, we may note that the returned scalars include the first and last time periods in this panel data set and the range of the `id` variable, which is designated as `r(panelvar)`. The macros also include the time series calendar variable `r(timevar)` and the range of that variable in a form that can be readily manipulated (e.g., for graph titles).

A number of statistical commands are *r-class*, since they are not viewed as estimating a model. For instance, [R] **correlate** will return *one* estimated correlation coefficient, irrespective of the number of variables in the command's *varlist*: the correlation of the last and penultimate variables.[7] The [R] **ttest** command is also r-class, so that we may access its return list to retrieve all of the quantities it computes:

Exhibit 17

```
. g lowind = (ind<6)
. ttest emp, by(lowind)
Two-sample t test with equal variances
```

Group	Obs	Mean	Std. Err.	Std. Dev.	[95% Conf.	Interval]
0	434	8.955942	.9540405	19.87521	7.080816	10.83107
1	597	7.11799	.5019414	12.26423	6.132201	8.103779
combined	1031	7.891677	.496273	15.93492	6.917856	8.865498
diff		1.837952	1.004043		-.1322525	3.808157

```
Degrees of freedom: 1029
                     Ho: mean(0) - mean(1) = diff = 0
     Ha: diff < 0                 Ha: diff != 0                 Ha: diff > 0
       t =   1.8306               t =   1.8306                  t =   1.8306
   P < t =   0.9663           P > |t| =   0.0675          P > t =   0.0337
. return list
scalars:
                r(sd) =  15.93492193741317
              r(sd_2) =  12.26422618476487
              r(sd_1) =  19.87520847697869
               r(se) =  1.004042693732077
              r(p_u) =  .0337282628926325
              r(p_l) =  .9662717371073675
                r(p) =  .067456525785265
                r(t) =  1.83055206312211
             r(df_t) =  1029
             r(mu_2) =  7.117989959978378
              r(N_2) =  597
             r(mu_1) =  8.955942384452314
              r(N_1) =  434
```

ereturn list

An even broader array of information is provided after any e-class (Estimation) command, displayable via `ereturn list`. Most e-class commands

[7] If a whole set of correlations are required for further use, use `mat accum C =` *varlist*, `dev nocons` followed by `mat Corr = corr(C)`.

will return four types of Stata objects: scalars such as `e(N)`, summarizing the estimation process; macros, providing such information as the name of the response variable (`e(depvar)` and the estimation method (`e(model)`); matrices `e(b), e(V)` as described above; and a Stata *function*, `e(sample)`, which will return 1 for each observation included in the estimation sample, zero otherwise. For example, consider a simple regression:

Exhibit 18

```
. regress emp wage cap
      Source |       SS       df       MS              Number of obs =    1031
-------------+------------------------------           F(  2,  1028) = 1160.71
       Model |  181268.08     2    90634.04            Prob > F      =  0.0000
    Residual |  80271.3092  1028  78.0849311            R-squared     =  0.6931
-------------+------------------------------           Adj R-squared =  0.6925
       Total |  261539.389  1030  253.921737            Root MSE      =  8.8366

------------------------------------------------------------------------------
         emp |      Coef.   Std. Err.      t    P>|t|     [95% Conf. Interval]
-------------+----------------------------------------------------------------
        wage |  -.3238453   .0487472    -6.64   0.000    -.4195008   -.2281899
         cap |   2.104883   .0440642    47.77   0.000     2.018417    2.191349
       _cons |   10.35982   1.202309     8.62   0.000     8.000557    12.71908
------------------------------------------------------------------------------

. ereturn list
scalars:
                  e(N) =  1031
               e(df_m) =  2
               e(df_r) =  1028
                  e(F) =  1160.711019312048
                 e(r2) =  .6930813769821942
               e(rmse) =  8.83656783747737
                e(mss) =  181268.0800475577
                e(rss) =  80271.30921843699
               e(r2_a) =  .6924842590385798
                 e(ll) =  -3707.867843699609
               e(ll_0) =  -4316.762338658647

macros:
             e(depvar) : "emp"
                e(cmd) : "regress"
            e(predict) : "regres_p"
              e(model) : "ols"

matrices:
                  e(b) :  1 x 3
                  e(V) :  3 x 3

functions:
             e(sample)

. local regressors: colnames e(b)

. di "Regressors: `regressors'"
Regressors: wage cap _cons
```

Two particularly useful scalars on this list are `e(df_m), e(df_r)`: the Model

and Residual degrees of freedom, respectively (the numerator and denominator d.f. for `e(F)`). The `e(rmse)` allows retrieval of the `Root MSE` of the equation. Two of the scalars do not appear in the printed output: `e(ll)`, `e(ll_0)`, the likelihood function evaluated for the estimated model and for the "null model", respectively.[8] Although the name of the response variable is available in macro `e(depvar)`, the names of the regressors are not shown here. They may be retrieved from the matrix `e(b)`, as illustrated in the example. Since the estimated parameters are returned in a $1 \times k$ row vector, the variable names are column names of that matrix.

Many official Stata commands—as well as many user-written routines—make use of the information available via `ereturn list`. How can a [R] **regression diagnostics** command like `ovtest` compute the necessary quantities after [R] **regress**? Because it can retrieve all relevant information: the names of the regressors, dependent variable, and the net effect of all `if` *exp* or `in` *range* conditions (via `e(sample)`) from the results left behind as e-class scalars, macros, matrices or functions by the e-class command. Any do-file you write can perform the same magic: just use `ereturn list` to find the names of each quantity left behind for your use, and store the results you need in local macros or scalars immediately after the e-class command. As noted above, retaining scalars as scalars is preferable from the point of view of precision.

`estimates` and `estout`

We should also note that e-class commands, producing estimates, may be followed by any of the `estimates` suite of commands: estimates may be saved in sets, manipulated, and combined in tabular form. The [R] **estimates** command that addresses the need to organize several equations' estimates into a tabular form for scrutiny and, perhaps, publication is `estimates table`. One specifies that a table is to be produced containing several sets' results, and Stata automatically handles alignment of the coefficients into the appropriate rows of a table. Options allow the addition of estimated standard errors (`se`), t-values (`t`), p-values (`p`) or significance stars (`star`). Each of these quantities may be given its own display [R] **format** if the default is not appropriate, so that the coefficients, standard errors, t- and p-values need not be rounded by hand. The order of coefficients in the table may be controlled

[8] In this case of OLS regression, the "null model" is that considered by the ANOVA F: the intercept-only model with all slope coefficients set equal to zero.

by the `keep()` option (rather than relying on the order in which they appear in the list of estimates' contents), and certain variables may be removed from the coefficient table with `drop()`. Published results often omit certain regressors, such as sets of indicator variables; this may be achieved with this option. Any result left in `e()` (see [P] **ereturn**) may be added to the table with the `stat()` option, as well as several additional criteria such as the AIC and BIC model selection criteria. As an example, using several specifications from the housing price model:

Exhibit 19

```
. generate rooms2 = rooms^2
. qui reg lprice rooms
. est store model1
. qui reg lprice rooms rooms2 ldist
. est store model2
. qui reg lprice ldist stratio lnox
. est store model3
. qui reg lprice lnox ldist rooms stratio
. est store model4
. est table model1 model2 model3 model4, stat(r2_a rmse) ///
>  b(%7.3g) se(%6.3g) p(%4.3f)
```

Variable	model1	model2	model3	model4
rooms	.369	-.821		.255
	.0201	.183		.0185
	0.000	0.000		0.000
rooms2		.0889		
		.014		
		0.000		
ldist		.237	-.157	-.134
		.0255	.0505	.0431
		0.000	0.002	0.002
stratio			-.0775	-.0525
			.0066	.0059
			0.000	0.000
lnox			-1.22	-.954
			.135	.117
			0.000	0.000
_cons	7.62	11.3	13.6	11.1
	.127	.584	.304	.318
	0.000	0.000	0.000	0.000
r2_a	.399	.5	.424	.581
rmse	.317	.289	.311	.265

legend: b/se/p

In this example, we estimate four different models of median housing price

and use `estimates table` to present the coefficients, estimated standard errors and p-values in tabular form. The `stats` option is employed to add summary statistics from the `e()` results.

After a final set of results are chosen, we will want to move these tables to a different output format: preferably without requiring manual intervention in, say, a spreadsheet. A full-featured solution to preparing publication-quality tables in various output formats has recently been made available to the Stata user community by Ben Jann: `estout`. This routine, which he describes as a wrapper for `estimates table`, allows the reformatting of stored estimates in a variety of formats, the combination of summary statistics from model estimation, and output to several formats, including tab-delimited (for word processors or spreadsheets, LaTeX, and HTML. A companion program, `estadd`, allows the addition of user-specified statistics to the `e()` arrays accessible by [R] **estimates**. These useful programs are available via [R] **ssc**.

As an example, we format the four models of median housing price for inclusion in a LaTeX document. This rather involved example of the use of `estout` places the LaTeX headers and footers in the file and ensures that all items are in proper format for that typesetting language (e.g., the use of `_cons` would cause a formatting error unless modified).

Exhibit 20

```
.
. estout model1 model2 model3 model4 using ch3.19b_est.tex, ///
> style(tex) replace title("Models of median housing price")  ///
> prehead(\\begin{table}[htbp]\\caption{{\sc @title}}\\centering\\medskip ///
> \begin{tabular}{l*{@M}{r}}) ///
> posthead("\hline") prefoot("\hline") ///
> varlabels(rooms2 "rooms$^2$" _cons "constant") legend ///
> stats(N F r2_a rmse, fmt(%6.0f %6.0f %8.3f %6.3f) ///
> labels("N" "F" "$\bar{R}^2$" "RMS error")) ///
> cells(b(fmt(%8.3f)) se(par format(%6.3f))) ///
> postfoot(\hline\end{tabular}\end{table}) notype
```

The LaTeX fragment produced by this command may now be inserted directly in a research paper. Virtually every detail of the table may be modified by `estout` directives. Since LaTeX is a markup language (like HTML), formatting changes may be programmed: a flexibility lacking from other `estout` output options such as tab-delimited text for inclusion in a word processing document.

Table 1: Models of median housing price

	model1 b/se	model2 b/se	model3 b/se	model4 b/se
rooms	0.369	-0.821		0.255
	(0.020)	(0.183)		(0.019)
rooms^2		0.089		
		(0.014)		
ldist		0.237	-0.157	-0.134
		(0.026)	(0.050)	(0.043)
stratio			-0.077	-0.052
			(0.007)	(0.006)
lnox			-1.215	-0.954
			(0.135)	(0.117)
constant	7.624	11.263	13.614	11.084
	(0.127)	(0.584)	(0.304)	(0.318)
N	506	506	506	506
F	337	169	125	176
$\bar{R}^2$	0.399	0.500	0.424	0.581
RMS error	0.317	0.289	0.311	0.265

The `program` and `syntax` statements

In this last part of the talk we discuss the rudiments of a more ambitious task: writing one's own *ado-file*, or Stata command. The distinction between our prior examples of do-file construction and an ado-file is precisely that: if you have written `myprog.do`, you may execute it with the Stata command `do myprog`. If you have written `myrealprog.ado`, on the other hand, you may invoke it as the Stata command `myrealprog` as long as it is defined on the [R] **adopath**. A nice illustration of transforming a do-file into an ado-file is presented in Watson (2005).

There are more profound differences, to be sure; ado-file programs may accept *arguments* in the form of a *varlist*, `if` *exp* or `in` *range* conditions, or options. Nevertheless, one need not go very far beyond the do-file examples we display above in order to generate a Stata command. Consider our discovery that the [R] **summarize** command does not compute the standard error of the mean. We might have need for that quantity for a number of variables, and despite other ways of computing it with existing commands, let us write a program to do so. In this example, we define the program in a do-file. In practice, we would place the program in its own file, `semean.ado`:

Exhibit 21

```
. capture program drop semean
. *! semean  v1.0.0  CFBaum  16dec2004
. program define semean, rclass
  1. version 8.2
  2. syntax varlist(max=1 numeric)
  3. quietly summarize `varlist'
  4. scalar semean = r(sd)/sqrt(r(N))
  5. di _n "Mean of `varlist' = " r(mean) " S.E. = " semean
  6. return scalar semean = semean
  7. return scalar mean = r(mean)
  8. return local var `varlist'
  9. end
. semean emp
Mean of emp = 7.891677 S.E. = .49627295
. return list
scalars:
              r(mean) =  7.891677013539667
            r(semean) =  .4962729540865196
macros:
               r(var) : "emp"
```

We start with a `capture program drop` *progname* command, since once a program has been loaded into Stata's memory, it is retained for the duration

of the session. As we may be repeatedly defining our program during its development, we want to ensure that the latest version is retained. The following comment line, starting with `*!`, is a special form that will show up in the [R] **which** command. It is always a good idea to document an ado-file with a sequence number, author name, and date. The [P] **program** statement identifies the program name: in this case `semean`, which we have ascertained via [R] **findit** is not the name of an existing Stata command, and defines the program as `rclass`. Unless a program is defined as `rclass` or `eclass`, it may not return values. The following [P] **version** line states that the ado-file requires Stata 8.2 (or better), and ensures that the program will obey Stata 8.2 syntax when executed by Stata 9 or Stata 10.

The following line, [P] **syntax**, provides the ability for a Stata program to parse its command line and extract the program's arguments for use within the program. In this simple example, we only use one element of [P] **syntax**: specifying that the program has a mandatory *varlist* with a maximum of one element. Stata will enforce the constraint that a single name appears on the command line, and that that name refers to an existing numeric variable. The following lines echo those of the do-file example above, computing the scalar `semean` as the standard error of the mean. The next lines then use [P] **return** to place two scalars, `semean` and `mean`, and one macro (the variable name) in the return array. As the example demonstrates, invoking the program generates one line of output, and `return list` displays the three items returned from the program for future use.

All well and good, but a statistical command should accept `if` *exp* and `in` *range* qualifiers if it is to be useful. We might also want to use this program as a calculator, without printed output; we could always invoke it [R] **quietly**, but an option to suppress output would be useful. It turns out that very little work is needed to add these useful features to our program. The definition of `if` *exp* and `in` *range* qualifiers and program options is all handled by the [P] **syntax** statement. In the improved program, `[if]` and `[in]` denote that each of these qualifiers may be used; the square brackets `[ ]` in [P] **syntax** signify an optional component of the command. The `[, noPRInt]` indicates that the command has a "noprint" option, and that it is truly optional (paradoxically one may have non-optional or "required" options on a Stata command). We then illustrate the revised program:

Exhibit 22

```
. capture program drop semean
. *! semean  v1.0.1  CFBaum  16dec2004
. program define semean, rclass
  1. version 8.2
  2. syntax varlist(max=1 numeric) [if] [in] [, noPRInt]
  3. marksample touse
  4. quietly summarize `varlist' if `touse'
  5. scalar semean = r(sd)/sqrt(r(N))
  6. if ("`print'" ~= "noprint") {
  7.         di _n "Mean of `varlist' = " r(mean) " S.E. = " semean
  8.         }
  9. return scalar semean = semean
 10. return scalar mean = r(mean)
 11. return scalar N = r(N)
 12. return local var `varlist'
 13. end
. semean emp if year < 1982, noprint
. return list
scalars:
                  r(N) =  778
              r(mean) =  8.579679950573757
            r(semean) =  .6023535944792725
macros:
               r(var) : "emp"
```

Since with an `if` *exp* or `in` *range* qualifier something less than the full sample will be analyzed, we have returned `r(N)` to indicate the sample size used in the computations. The `marksample touse` command makes the `if` *exp* or `in` *range* qualifier operative if one was given on the command line; the command "marks" those observations which are to enter the computations in an indicator variable `touse`. This variable is a *tempvar*, or temporary variable, which (like a local macro) will not survive beyond the program's scope. One may explicitly create these temporary variables with the [P] **tempvar** command, and when a variable is needed within a program, that is the preferred style to avoid possible conflicts with the contents of the data set. Since the variable is temporary, we refer to it as we would a local macro, as `` `touse' ``, which is an alias to its internal (arbitrary) name. We must add `if `touse'` to each statement in the program which works with the input *varlist*: in this case, only the [R] **summarize** statement.

Two additional features would be useful in the context of this program. First, we would like it to be *byable*: to permit its use with a `by` *varlist:* prefix. Since we are not creating any new variables with this program, that may be accomplished by merely adding `byable(recall)` to the [P] **program**

statement (see [P] **byable** for details). We also might like to use time series operators (`L., D., F.`) with our program; adding the `ts` option to the *varlist* will enable that. To see these changes in practice:

Exhibit 23

```
. capture program drop semean
. *! semean  v1.0.2  CFBaum  16dec2004
. program define semean, rclass byable(recall)
  1. version 8.2
  2. syntax varlist(max=1 ts numeric) [if] [in] [, noPRInt]
  3. marksample touse
  4. quietly summarize `varlist' if `touse'
  5. scalar semean = r(sd)/sqrt(r(N))
  6. if ("`print'" ~= "noprint") {
  7.         di _n "Mean of `varlist' = " r(mean) " S.E. = " semean
  8.         }
  9. return scalar semean = semean
 10. return scalar mean = r(mean)
 11. return scalar N = r(N)
 12. return local var `varlist'
 13. end
. semean D.emp if year == 1982
Mean of D.emp = -.79091424 S.E. = .17187137
. bysort year: semean emp
-------------------------------------------------------------------------------
-> year = 1976
Mean of emp = 9.8449251 S.E. = 2.1021706
-------------------------------------------------------------------------------
-> year = 1977
Mean of emp = 8.5351159 S.E. = 1.393463
-------------------------------------------------------------------------------
-> year = 1978
Mean of emp = 8.6443428 S.E. = 1.3930028
-------------------------------------------------------------------------------
-> year = 1979
Mean of emp = 8.7162357 S.E. = 1.4311206
-------------------------------------------------------------------------------
-> year = 1980
Mean of emp = 8.5576715 S.E. = 1.4611882
-------------------------------------------------------------------------------
-> year = 1981
Mean of emp = 7.7214 S.E. = 1.3467025
-------------------------------------------------------------------------------
-> year = 1982
Mean of emp = 6.9304857 S.E. = 1.2245105
-------------------------------------------------------------------------------
-> year = 1983
Mean of emp = 5.2992564 S.E. = 1.3286027
-------------------------------------------------------------------------------
-> year = 1984
```

```
Mean of emp = 2.2205143 S.E. = .48380791
```

Finally, for pedagogical purposes, we demonstrate the addition of an interesting capability to the program: its ability to operate on a transformation of the *varlist* without first generating that variable. We make use of the [P] **tempvar** statement to allocate a temporary variable `target` which will be equated to the *varlist*, in the absence of the `function( )` argument, or that function of *varlist* if specified. The local macro `tgt` is used to store the "target" of the command, and used later to display the variable of interest (and the returned local macro `r(var)`). We place the `if 'touse'` qualifier on the [R] **generate** statement, and [P] **capture** the result of that statement to catch any errors: for instance, an attempt to use a function not known to Stata. The _rc (return code) is tested for a non-zero value, which will signify an error in the [R] **generate** command.

Exhibit 24

```
. capture program drop semean
. *! semean  v1.1.0  CFBaum  16dec2004
. program define semean, rclass byable(recall)
  1. version 8.2
  2. syntax varlist(max=1 ts numeric) [if] [in] [, noPRInt FUNCtion(string)]
  3. marksample touse
  4. tempvar target
  5. if "`function'" == "" {
  6.         local tgt "`varlist'"
  7.         }
  8. else {
  9.         local tgt "`function'(`varlist')"
 10.         }
 11. capture tsset
 12. capture gen double `target' = `tgt' if `touse'
 13. if _rc > 0 {
 14.         di as err "Error: bad function `tgt'"
 15.         error 198
 16.         }
 17. quietly summarize `target'
 18. scalar semean = r(sd)/sqrt(r(N))
 19. if ("`print'" ~= "noprint") {
 20.         di _n "Mean of `tgt' = " r(mean) " S.E. = " semean
 21.         }
 22. return scalar semean = semean
 23. return scalar mean = r(mean)
 24. return scalar N = r(N)
 25. return local var `tgt'
 26. end
. semean emp
Mean of emp = 7.891677 S.E. = .49627295
. semean emp, func(sqrt)
Mean of sqrt(emp) = 2.1652401 S.E. = .05576835
```

```
. semean emp if year==1982, func(log)
Mean of log(emp) = .92474464 S.E. = .11333991
. return list
scalars:
                  r(N) =  140
               r(mean) =  .9247446421128256
            r(semean) =  .1133399069800714
macros:
               r(var) : "log(emp)"
. semean D.emp if year==1982, func(log)
Mean of log(D.emp) = -2.7743942 S.E. = .39944652
. return list
scalars:
                  r(N) =  22
               r(mean) =  -2.774394169773632
            r(semean) =  .3994465211383764
macros:
               r(var) : "log(D.emp)"
```

As the example demonstrates, the program operates properly in the case of applying a transformation that reduces the sample size: the log of `D.emp` is only defined for positive changes in employment, and most of the 140 firms in this sample suffered declines in employment in 1982.

The program now is capable of emuiating many of the features of an official Stata command, and remains a very brief chunk of ado-file code. We have only scratched the surface of what may be done in your own ado-file: for instance, many user-written programs will generate new variables, or perform computations based on the values of options which may have their own default values. User-written programs may also be used to define additional [R] **egen** functions, in which case their name (and the file in which they reside) will start with `_g`: that is, `_gzorch.ado` will define the `zorch()` function to [R] **egen**. You may also find yourself writing ado-file programs in order to use [R] **ml**, [R] **nl** or [R] **simulate**.

Although many researchers may become very efficient users of Stata without ever writing an ado-file program, others will find that "quick-and-dirty" code that gets the job done today must be rewritten incessantly, with minor variations, to perform a similar task tomorrow. With that epiphany, the knowledgeable Stata user will recognize that it is a short leap to becoming more productive by learning how to write their own ado-files, whether or not those programs are of general use or meant to be shareable with other Stata users.

Final thoughts

I hope that you may be convinced that gaining familiarity with the constructs I have described can reduce the likelihood of errors, provide a mechanism for retracing any part of your research strategy, and simplify the sequence of commands needed to achieve a given task. This adds generality to your use of Stata even without invoking the added power of ado-file programming. Furthermore, if like myself you are called upon to transfer some of the technology you have developed to students, co-workers or Statalist posters, you will have the worked examples to give them at your fingertips. Truly, then, *a little bit of Stata programming goes a long way!*

0.1 References

Cox, N. J. 2001. Speaking Stata: How to repeat yourself without going mad. *Stata Journal* 1(1): 86–97.

—. 2002. Speaking Stata: How to face lists with fortitude. *Stata Journal* 2(2): 202–222.

—. 2003. Speaking Stata: Problems with lists. *Stata Journal* 3(2): 185–202.

Watson, I. 2005. Further processing of estimation results: Basic programming with matrices. *Stata Journal* 5(1): 83–91.