Statistics/Data Analysis

copycode

Title

copycode -- produce modular self-written ado files

Syntax

copycode *projectname* , **inputfile(***inputfilename***)**
[**{targetfile(***targetfilename***)** | **nocopy }** *other_options*]

main_options	Description
projectname	Name of project to be processed
inputfile(*inputfilename***)**	Name/path of input file that contains the file listings
targetfile(*targetfilename***)**	Name/path of output file
nocopy	Show command output without performing the copy operation

other_options	Description
simplemode	Process project under simple mode (as opposed to ado mode)
replace	Replace target file if it already exists
force	Do not confine copying of lines to copy regions
starbang(*stb_mode***)**	Determine whether starbang lines should be copied
noprogdrop	Do not automatically drop program corresponding to *targetfilename* from memory

Description

copycode copies different source code files into one file. A typical application would be an ado file whose final version is supposed to contain subroutines and/or Mata routines that you are accustomed to use and re-use in your ados.

copycode is thought to facilitate code production, code maintenance and code distribution for Stata programmers who write Stata/Mata code on a large scale. You can write small programs and functions, each one in a separate file, test them thoroughly, and then include them in any ado file using **copycode**. When you have to make changes to these general purpose functions the re-make of the ado files is completely automated.

If you have written many ado files with many interdependencies (i.e. many of your self-written ados call other self-written ados), **copycode** enables you to organize your code in such a way that each final ado file is "modular", i.e. does not depend on other user-written code files. This makes the distribution of your files easier.

If you do not want to use **copycode** to produce your final ado files, you can still use it to shed light on the dependency structure of your files (see the nocopy subsection below).

Another feature of **copycode** is that it allows you to add "private" comments to your source code file. Your final published file will not include these lines.

There is also a wrapper command for **copycode** available called fastcc ("fast copycode") that will allow you to work with **copycode** more efficiently when using it interactively.

For an alternative description of **copycode** see Schneider (2012).

Defintions used in this help entry

The definitions of the following table are repeated in the help text as terms appear. The table is thought to serve as a reference for a more thorough reading of this help entry.

modular ado file	A (user-written) ado file whose functioning does not require the presence/installation of other user-written code files.
project	Identifies a set of code files that belong together.
main adv	File that is listed first for a project and whose name corresponds to the project name and that has an .adv extension. It contains the main program of a (ado) project.
project component file	Each file entry under a project in the input file.
dependency	Program or function that is called by another program or function. There can be direct/first-order and indirect/higher-order dependencies.
copy region limits	Strings '!copycodebeg=>' (beginning of a copy region) and '!copycodeend||' (end of a copy region).
copy region	Lines of a text file between the strings '!copycodebeg=>' and '!copycodeend||'.

Options

projectname Name of project in the input file that pins down the file listing.

inputfile(inputfilename) specifies the file path and name of the input file. Relative file paths are accepted. The absolute part of the path is assumed to be the working directory.

targetfile(targetfilename) determines the file path and name of the output file. Relative file paths are accepted. The absolute part of the path is assumed to be the working directory.

nocopy can be used to specify that no targetfile should be created. If this option is used, **copycode** performs fewer consistency checks of what you supply as input. It still performs a consistency check of the input file. It does not, however, check for the existence of files, nor does it check for the consistency of copy region limits within code files.

When using option **nocopy**, options **replace**, **force** and **starbang** are ignored. Option **nocopy** does observe the usual rules that apply in determining whether **copycode** is run in simple mode or in ado mode.

simplemode will force simple mode. Ado mode is entered when the first file for the project to be processed is a "main adv" file. Option **simplemode** overrides this. Then, only the files listed under *projectname* (direct dependencies) will be copied together, without checking for dependencies of adv files that belong to the project (see section Simple mode and ado mode below).

replace will overwrite the outputfile if it already exists.

force tells **copycode** to perform copy operations irrespective of copy regions of the code files. Under option **force** all lines of all files will be copied except for starbang lines. What happens to starbang lines is determined by option **starbang(***stb_mode***)**.

starbang(stb_mode) specifies whether starbang lines should be copied to the output file. The contents of starbang lines are displayed by the which command. There are three modes, i.e. *stb_mode* may assume three values: "skip" will not copy any starbang lines. "first" will only copy starbang lines of the main adv file. "all" will copy all starbang lines.

Typically, when producing an ado file you only want to have the starbang line(s) of the main adv displayed by **which** and not the ones of subordinate advs or subroutines, which is why *stb_mode* defaults to "first" in ado mode. In simple mode, however, *stb_mode* defaults to "all".

noprogdrop will prevent **copycode** from automatically issuing a
. capture program drop *targetbasename*
statement, where *targetbasename* is the file name "base" of *targetfilename*. For example, if you specify **targetfile(mypath/mytarget.ado)**, copycode normally issues a
. capture program drop mytarget
to ensure that after the re-make of the project no old version of the project resides in memory. Option **noprogdrop** prevents this statement from being executed.

Usage of option **nocopy** will also prevent the "program drop" statement.

Remarks

Remarks are presented under the following headings:

Overview

copycode compiles a list of code files (text files) that accounts for code dependencies of a project and copies code from these files to *targetfile*.

A dependency is a user-written program or function that is called by another user-written program or function. There can be direct/first-order and indirect/higher-order dependencies. If a.ado calls b.ado and b.ado calls c.ado, b.ado is a direct dependency of a.ado, c.ado is a direct dependency of b.ado, and c.ado is an indirect dependency of a.ado. With many ado files, these dependency structures can become complex.

copycode will figure out dependency structures if you supply it with an input file that lists, for each project, the direct dependencies. This input file must have the following structure:

- Each line must contain exactly two tokens.
- The first token specifies the project.
- The second token specifies the path and name to a file.

An example of an input file is given in the Input file structure section. Once all dependencies have been determined, **copycode** copies all relevant code into *targetfile* which then does not depend on any other user-written files. The targetfile must always be specified explicitly. This is to prevent accidental overwriting of existing files.

While copying, **copycode** takes so-called "copy regions" into account. It copies only lines that you have marked to be copied.

Backup of files

copycode concerns itself mainly with writing ado files to disk. If you use or try out **copycode**, 1) make a backup of your self-written files and 2) keep the backup in a safe place. Otherwise there is the danger of accidentially overwriting and losing self-written code files.

Simple mode and ado mode

copycode compiles file lists differently, depending whether it is run in simple mode or in ado mode. Simple mode can be invoked e.g. by using option **simplemode**. Then the files to be copied correspond exactly to the lines of the input file whose first-column entry equals *projectname*.

In ado mode, **copycode** works differently. Ado mode is related to the following problem: When you write an ado file and your ado file depends on other ados you have written it becomes difficult to distribute your code - especially if you have many ados with many interdependencies.

One solution is to use **copycode** to assemble the new ado file that contains the code of all ados that are necessary to make the new ado file run. For example, if new.ado calls your user-written function existing.ado, you can (sort of) simply copy the code from existing.ado into new.ado, that way making it a modular routine. Roughly speaking, this is the approach taken by **copycode**. What you have to supply to the command is an input file that lists the direct dependencies of your (ado) projects.

Ado mode is entered when the first file of a project is an adv file (see the next section on file extensions) whose file name is identical to the project name. This file is then called the "main adv" of the project.

Most of what follows is related to using **copycode** in ado mode.

File extensions

copycode handles the following file extensions in a special way:

.adv	"ado development file", contains the newly written code necessary for your new ado project. In ado mode, when encountering an adv file that is listed as a dependency in the input file, **copycode** will search the input file for a correspondingly named project. This project must exist, otherwise **copycode** will error out. Any files listed under this project (which may just consist of one adv file) are included in the list of files to be copied and in turn searched for further dependencies.
.ado	Existing modular user-written ado files to be used as subroutines (appended) to the new ado project
.stp	"Stata program file", contains modular Stata code to be appended to the code in the adv file. stp files are thought to contain short and simple Stata programs that do not depend on other programs. You may ignore stp file types and put all of your Stata programs into adv files.
.mata	Mata source code file, contains modular Mata code to be appended to the code in the adv file

These files contain Stata programs and Mata functions. You should always name your files according to the names of the programs, e.g. sub1.stp should contain a Stata program called "sub1", and func2.mata should contain a Mata function called "func2".

copycode will copy the code from the files indicated in your input file in the order given in the table above: First it copies your new code from your new adv, then (possibly) codes from other adv files, then code from modular (i.e. non-dependent) ado files. Then it copies code from stp files, then code from mata files, and finally code from files with any other extension.

In ado mode, within each extension group, files are copied in the order of appearance in the input file. In simple mode files are copied exactly in the order of appearance in the input file.

copycode does not check for dependencies of stp files because they should only contain small programs that do not call other programs. Checking for the depenencies among mata files would be desirable, but this has not yet been implemented. To summarize, only adv files are checked for dependencies.

Input file structure

copycode can detect complex dependency structures among user-written files. What it needs as an input, and what you have to provide, is a file that lists all (ado) projects and their direct dependencies. This is enough information to create a list of dependencies of all orders, for any project in the input file.

Here is an example of such an input file:

```
myinput.txt -------------------------------------
//        Optional project A description goes here
//        note that comments are allowed in the input file
proj_a    c:\ado\personal\proj_a.adv
proj_a    c:\ado\personal\proj_b.adv
proj_a    c:\ado\personal\modular.ado
proj_a    c:\ado\personal\sub1.stp
proj_a    c:\ado\personal\sub2.stp
proj_a    c:\ado\personal\func1.mata
proj_a    c:\ado\personal\func2.mata

//        Optional project B description goes here
proj_b    c:\ado\personal\proj_b.adv
proj_b    c:\ado\personal\proj_c.adv
proj_b    c:\ado\personal\func3.mata
proj_b    c:\ado\personal\sub2.stp
proj_b    c:\ado\personal\sub3.stp
proj_b    c:\ado\personal\func2.mata
proj_b    c:\ado\personal\remarks.txt

//        Optional project C description goes here
proj_c    c:\ado\personal\proj_c.adv
-----------------------------------------------
```

The first column of your input file specifies the project name. The second column specifies the path to the files whose code goes into the output file.

Suppose you want to create a new ado file proj_a.ado that you want to generate using **copycode**, in ado mode. Then you create a project "proj_a.ado" in your input file. The first entry of that project references a file "proj_a.adv". This is the "main adv" of proj_a and contains the main routine of the new ado file. Then you add entries for all direct dependencies that occur in the code contained in proj_a.adv.

Project names are not case-sensitive and may not contain blanks. File paths may contain blanks but be sure to surround the entries by double quotes or compound double quotes in this case. File names may not contain blanks. File paths must be supplied as absolute paths (e.g. c:\ado\personal\mydir\myfile.adv). Relative file paths (e.g. mydir\myfile.adv) are not allowed in the input file. File paths may not exceed 244 characters.

Tab characters on comment lines and in between input tokens are allowed and are interpreted as blanks.

As can be seen from the above example, the input file may contain empty lines and comments. Comments must be indicated by "//" which must be the first non-blank character sequence on a comment line. Comments must be on a separate line, i.e. they may not occur on the same line as an input entry.

When running **copycode** in ado mode, the above input file will produce ado files that contain code from the following files, in the order as given below:

```
Project A
          proj_a.adv
          proj_b.adv
          proj_c.adv
          modular.ado
          sub1.stp
          sub2.stp
          sub3.stp
          func1.mata
          func2.mata
          func3.mata
          remarks.txt
```

```
        Project B
                  proj_b.adv
                  proj_c.adv
                  sub2.stp
                  sub3.stp
                  func3.mata
                  func2.mata
                  remarks.txt

        Project C
                  proj_c.adv
```

When running **copycode** in simple mode, the output files will contain code from the file list recorded unter each project, in the same order:

```
        Project A
                  proj_a.adv
                  proj_b.adv
                  modular.ado
                  sub1.stp
                  sub2.stp
                  func1.mata
                  func2.mata

        Project B
                  proj_b.adv
                  proj_c.adv
                  func3.mata
                  sub2.stp
                  sub3.stp
                  func2.mata
                  remarks.txt

        Project C
                  proj_c.adv
```

The composition of the final ado files produced under ado mode is discussed in the next section.

Code dependencies

From the above example, note the following:

- The order of file types to be copied into the final ado file is: adv - ado - stp - mata - other.
- Apart from the main adv, you do not have to stick to any particular order in which you specifiy project component files. The only exception to this concerns Mata structures. As you can see from func3.mata of project B above which is somewhat hidden, it makes things more transparent to keep the order adv-ado-stp-mata-other in the input file.
- The main adv file of the project (the first project entry) will always be copied first.
- All subroutines from adv dependencies are copied: Files that are referenced by proj_b are copied into proj_a.ado since proj_a has proj_b.adv as dependency (sub3.stp and remarks.txt are used directly by proj_b only and not by proj_a). proj_a.ado requires all of its calls to its private program proj_b to work; hence, all project component files for proj_b have to be present in proj_a.ado.
- proj_c.adv is copied into the targetfile for proj_a because proj_a depends on proj_c.adv through proj_b.adv. proj_c.adv is a second-order dependency of proj_a.
- There is no duplicate copying of files: sub2.stp is copied only once into proj_a.ado, even though proj_a has proj_b.adv as dependency and both proj_a and proj_b require the usage of sub2.stp.
- In proj_a, modular.ado is copied without any attempt to detect its dependencies, i.e. if you specify an ado file it is presumed to be modular.
- If an adv file is specified as a project component file, a project with the same name must exist. In the above example, when processing project A or project B, copycode sooner or later stumbles upon the reference proj_c.adv which it needs to resolve. If it cannot determine whether proj_c.adv has dependencies (which is the case if project proj_c does not exist) it will

error out.

Circular references of adv files are not allowed: If proj_1.adv depends proj_2.adv, proj_2.adv must not depend on proj_1.adv. If you try to process a project that has such a structure **copycode** will issue an error. To detect circular references, a maximum of 1000 direct or indirect adv dependencies (in total) is set. You will necessarily hit this limit if your input file defines a circular reference. Note that a circular reference error will also be issued if these circular dependencies are not on the top level of dependencies. For example, consider the input file

```
proj_1    c:\ado\personal\proj_1.adv
proj_1    c:\ado\personal\proj_2.adv
proj_2    c:\ado\personal\proj_2.adv
proj_2    c:\ado\personal\proj_3.adv
proj_3    c:\ado\personal\proj_3.adv
proj_3    c:\ado\personal\proj_1.adv
```

An error is issued because proj_1 depends on proj_2 which depends on proj_3 which in turn depends on proj_1, so the references have a circular structure.

Copy regions

Before **copycode** starts copying the lines of a file, it searches the file for the strings '!copycodebeg=>' and '!copycodeend||'. Then it only copies the lines in between the two strings (excluding the lines containing these strings) to *targetfile*. Henceforth, the two strings are called "copy region limits" and the lines between the limits are called a "copy region". A file may contain multiple copy regions. Copy region limits must be present in each file, occur in the correct order, and reside on different lines. Otherwise an error is issued. You can prevent the error from being issued by using option **force**. In this case the entire contents of the files are copied, except for starbang lines. What happens to starbang lines is determined by option **starbang**. The usage or omission of option **force** applies to all files in the project listing.

The purpose of copy regions is that you can keep comments and auxiliary commands/functions in the original files and still have a clean composite file.

Avoiding file loss

The output files that you want to create with **copycode** normally are ado files. This means that the ado files that you are creating are fully overwritten whenever you make changes to component files and re-process a project. Even if you proceed carefully, it may happen that you accidentially overwrite code that you have newly written.

In particular, this may happen if some of your ado files have a complex dependency structure so you create these ado files using **copycode**, but other ado files are very simple, do not have dependencies and are outside of the **copycode** system, i.e. written in the usual way. Then you may get into the habit of sometimes modifying adv files (to modify ado files created by **copycode**), and sometimes ado files (to directly modify the small and simple ado files). You can then easily by accident change the ado file of one of the complex ado projects. The next time you re-process this project, this ado file will get overwritten and your work will be lost.

One save solution is to **make it a rule to never ever edit ado files**. Include all of your files in the **copycode** system, even the small and simple ones that do not have dependencies. This means giving the simple ado files an adv extension, adding one project line per adv file to the **copycode** input file, and producing the ado file using **copycode**. This may sound complicated but it is usually a change that can be implemented very quickly. Once you have done so, you should have a look at fastcc to make working with **copycode** more efficient.

Downsides

While **copycode** makes your life as a developer easier by reducing the amount of code that needs to be written and maintained, its major disadvantage is that it produces code bloat **for the user** of your ado routines. Including the code of general purpose functions or even full-blown ado routines in a new ado file means that larger parts of the code are probably not necessary to make the new ado work. If a user of your ados wants to look at your code, it may take longer to read through it.

You may alleviate this problem by including a comment at the top of each adv file, e.g.

```
// Some portions of the code for subroutines may not be necessary for the
// functionality of this ado file since it has been produced with
-copycode-.
// More information is available on SSC: type -ssc help copycode-
// A separate help file for the program below may be available on SSC.
```

If the adv code is used as a subroutine, the user may be able to just read through the help file if it is available, instead of having to work through the code itself.

Another downside is the limited ability to format the target file. Last but not least, it takes a little while to understand how the command works (as you may have noticed).

The 'nocopy' option

If you do not want to use **copycode** to generate your self-written final ado files, you can still put it to use to shed light on the dependency structure of your code files.

You still have to create an input file like the one above that matches the direct dependencies of the projects that you have written. The screen output of **copycode** and the results saved in **r()** will then tell you the details about the full dependency structure of your projects.

Be sure to give your self-written ado file dependencies in your input file an adv extension. These files may actually exist as ado files (not as adv files) on your hard drive. You must still give them an adv extension in the input file, because only then will **copycode** check for their dependencies.

Use option 'nocopy' then. Then you do not have to create a bogus target file. Also, **copycode** concerns itself only with the dependency structure that it can glean from the input file and does not check for the existence of files and the consistency of copy region limits within the code files.

Miscellaneous remarks

Relationship of ado mode / simple mode, option force, and option starbang. Running **copycode** in ado mode / simple mode, using option **force** and using option **starbang** are in principle three different things independent of each other.

- Choosing ado mode / simple mode determines whether higher-order dependencies of a project should be copied.
- Option **force** says that copy operations should not be confined to copy regions.
- Option **starbang** determines whether starbang lines should be included in the copy operations.

The only link between the above three features is that in ado mode *stb_mode* of option **starbang** defaults to "first". In simple mode, it defaults to "all".

Potential function name clashes. There is a potential naming conflict between private subroutines that exist in adv/ado files referenced in the input file and other programs that you are refering to in the input file. As an example, consider the input file

```
------------------------------------------------
proj_a    c:\ado\personal\proj_a.adv
proj_a    c:\ado\personal\modular.ado
proj_a    c:\ado\personal\sub1.stp
------------------------------------------------
```

If modular.ado has a private subroutine (program) called "sub1" the output file produced by **copycode** will have two programs named "sub1" and Stata will refuse to load the ado. One way to remedy this is to adopt appropriate naming conventions, e.g. give subroutines within adv/ado files prefixes according to the adv/ado file name, etc.

It is perfectly fine to have subroutines in your main adv or any other adv/ado file. If you have a special purpose program/function whose only application is within one particular adv/ado file, the only issue of adding this function to the adv/ado (project component) file is to pay attention to potential naming conflicts. Make sure to keep these program/functions within copy regions so they will always be copied to the output file.

If you have a subroutine that turns out to be useful in other contexts as well it is beneficial to include this program in a separate adv or stp file, and add corresponding stp entries to projects that make use of the subroutine.

Debugging and certification scripts. Each of the files that go into the final ado can be loaded separately into Stata for debugging purposes. If you have a look at func1.mata used in the examples section, you can see that the code file func1.mata is used in the copycode system but can be loaded separately into memory:

```
. do func1.mata
```

will define func1() as a Mata function. This works because the actual code that gets copied by **copycode** is confined to copy region limits.

You can even push it further and use the file func1.mata as a certification script by adding initialization statements at the top of the file (see cscript) and add certification assert and confirm statements at the bottom of the file. You will still be able to use this file within the **copycode** context since, by default, **copycode** only copies the code that is located within copy region limits. If all certification-related lines are located outside of copy regions, you can use func1.mata as both a source code file for the **copycode** system and as a certification script. The same holds true for adv and for stp files.

Using code from other users. If your project depends on an ado file written by some other user, you can include this ado file in your **copycode** project as a dependency in order to make your project modular. Your final ado file will then contain the code from the "third-party" user-written ado file. It goes without saying that in these cases you should ask for permission first and give proper acknowledgements in your help files when you distribute your code.

Version statements. Another potential conflict concerns version statements. If you plan to build a new ado, say under "version 11", and accidentially include a program via **copycode** that has a "version 12" statement, the ado file will not execute under Stata 11. A cursory look at the (top of the) ado code, however, would (misleadingly) suggest that. You must manually check for appropriate version control statements in your output ado files.

Exit statements. If you routinely include "exit" statements at the end of your main adv definition programs (after the closing "end" statement), make sure to exclude them from the copy region so they do not show up in the new composite ado file. Otherwise it will not work.

Mata structures. In ado mode, within each file extension group, the order of files copied is the order of appearance in the input file. This is important if you use Mata structures. In the Mata code, structures have to be defined before they can be used in subsequent code. If you use Mata structure definitions in separate files, it is the safest strategy to put these files right underneath the main adv file of your project.

Other uses. You may discover other uses of **copycode**, especially using option **simplemode**. For example, you can use it to produce nice estimation output reports. First, generate various smcl log files with estimation output and manually write other smcl files that comment on the output. You can then use **copycode** to copy the smcl files into one file in any order desired, that way producing a nice and clean report.

Reverting to non-modular ado files. All you have to do is to create a copy of your input file and comment out all adv files of a project except for the main adv. Using the two input files you can very easily switch between creating modular ado files and non-modular ado files.

Using do-files. If you have many ado files, you probably want to have a "master" do file that processes all of your projects. For the input file used above, this do file could read

```
copycode_runall.do:
  ------------------------------------------------------------
  // FILE PURPOSE: GENERATE ALL SELF-WRITTEN ADO FILES
  copycode proj_a, inp(c:\ipath\myinput.txt) target(c:\tpath\proj_a.ado)
  replace
  copycode proj_b, inp(c:\ipath\myinput.txt) target(c:\tpath\proj_b.ado)
  replace
  copycode proj_c, inp(c:\ipath\myinput.txt) target(c:\tpath\proj_c.ado)
  replace
------------------------------------------------------------------------------
  > -
```

Operating system. While **copycode** can be used on any operating system that Stata runs on, it has only been tested on Windows.

Examples

Consider the following input file

```
input.txt: ---------------------------------------
//        Files for project A
proj_a    c:\ado\personal\proj_a.adv
proj_a    c:\ado\personal\sub1.stp
proj_a    c:\ado\personal\func1.mata
//        Files for project B
proj_b    c:\ado\personal\proj_b.adv
proj_b    c:\ado\personal\proj_a.adv
proj_b    c:\ado\personal\sub1.stp
proj_b    c:\ado\personal\func2.mata
-----------------------------------------------
```

The files that go into the final ado could look like this:

```
proj_a.adv: ----------------------------------------
          // !copycodebeg=>
          *! version 1.0.1 12jun2011 dcs
          program define proj_a
           display "hello world says proj_a"
           proj_a_sub1
           sub1
           /* !copycodeend||
           TODO: improve sub1
           by switching to...
           (...)
           !copycodebeg=> */
           mata: func1()
          end

          program define proj_a_sub1
           (...)
          end

          // !copycodeend||
```

```
                /*
                TODO:
                - make the routine do something useful
                - (...)

                VERSION HISTORY
                0.0.1 12apr 2011 (...)
                */
        -----------------------------------------------


        sub1.stp: -----------------------------------
                capture program drop sub1
                // !copycodebeg=>
                program define sub1
                 (...)
                end

                // !copycodeend||

                /*
                TODO
                 (...)
                */
        -----------------------------------------------


        func1.mata: ---------------------------------
                capture mata mata drop func1()
                // !copycodebeg=>
                mata:
                void func1() {
                 (...)
                }
                end

                // !copycodeend||

                /*
                TODO
                 (...)
                */
        -----------------------------------------------
```

The statement

```
    . copycode proj_a, inputfile(c:\mypath\input.txt)
    targetfile(c:\mytargetdir\proj_a.ado)
```

would produce c:\mytargetdir\proj_a.ado which reads:

```
        proj_a.ado: ---------------------------------
                *! version 1.0.1 12jun2011 dcs
                program define proj_a
                 display "hello world says proj_a"
                 proj_a_sub1
                 sub1
                 mata: func1()
                end

                program define proj_a_sub1
                 (...)
                end

                program define sub1
                 (...)
                end

                mata:
                void func1() {
                 (...)
```

```
                }
                end
        ------------------------------------------------
```

From the above example, note the following:

- **copycode** found the first file of proj_a to be an adv file whose name is identical to the project. It therefore entered ado mode. However, since there are no other adv files specified, there are no dependencies to detect.
- By default, **copycode** copies only copy regions. The private comments in the source code files do not show up in the output file.
- In ado mode, the **starbang(***stb_mode***)** option defaults to "first", which means that the starbang lines from the main adv file are copied to the output file.
- The main adv file contains a subroutine. Prepending the name of the subroutine by the name of the adv file ("proj_a_sub1") prevented a function name clash.

If, in addition, we assume the following files:

proj_b.adv:
```
        ------------------------------------
                // !copycodebeg=>
                *! version 1.2.0 30dec2011 dcs
                program define proj_b
                 display "hello world again"
                 proj_a
                 sub1
                 mata: func2()
                end

                // !copycodeend||

                /*
                TODO:
                - (...)

                VERSION HISTORY
                - (...)
                */
        ------------------------------------------------
```

func2.mata:
```
        ------------------------------------
                capture mata mata drop func2()
                // !copycodebeg=>
                mata:
                void func2() {
                 (...)
                }
                end

                // !copycodeend||

                /*
                TODO
                 (...)
                */
        ------------------------------------------------
```

The statement

```
    . copycode proj_b, inputfile(c:\mypath\input.txt)
    targetfile(c:\mytargetdir\proj_b.ado)
```

would produce c:\mytargetdir\proj_b.ado which reads:

```
proj_b.ado: -------------------------------------
          *! version 1.2.0 30dec2011 dcs
          program define proj_b
           display "hello world again"
           proj_a
           sub1
           mata: func2()
          end

          program define proj_a
           display "hello world says proj_a"
           proj_a_sub1
           sub1
           mata: func1()
          end

          program define proj_a_sub1
           (...)
          end

          program define sub1
           (...)
          end

          mata:
          void func2() {
           (...)
          }
          end

          mata:
          void func1() {
           (...)
          }
          end
-----------------------------------------------
```

From the above example, note the following:

- **copycode** found the first file of proj_b to be an adv file whose name is identical to the project. It therefore entered ado mode. It found proj_a as an adv input to proj_b and checked for a proj_a project entry in the input file. It included the files found there in the output file.
- If proj_a would have had a component file proj_c.adv, **copycode** would have included the component files from proj_c as well, and so forth.
- Again, in ado mode, option **starbang(***stb_mode***)** option defaults to "first", which means that the starbang lines from the main adv file are copied to the output file. The starbang lines from proj_a have not been copied.

Saved results

copycode saves the following in **r()**:

Macros

r(project)	Name of project that has been processed
	in simple mode, the following is also saved:
r(dep_direct)	File names of direct dependencies of the project
r(dep_direct_path)	Full file paths of the direct dependencies
	in ado mode, the following is also saved:
r(dep_all)	File names of all dependencies of the project
r(dep_adv)	File names of adv dependencies
r(dep_ado)	File names of ado dependencies
r(dep_stp)	File names of stp dependencies
r(dep_mata)	File names of mata dependencies
r(dep_other)	File names of other dependencies
r(dep_all_path)	Full file paths corresponding to r(dep_all)
r(dep_adv_path)	Full file paths corresponding to r(dep_adv)
r(dep_ado_path)	Full file paths corresponding to r(dep_ado)
r(dep_stp_path)	Full file paths corresponding to r(dep_stp)

r(dep_mata_path) Full file paths corresponding to r(dep_mata)
r(dep_other_path) Full file paths corresponding to r(dep_other)

All returned values are lower case, irrespective of what the actual casing of the files on disk or in the input file is.

Author

Daniel C. Schneider, Goethe University Frankfurt, schneider_daniel@hotmail.com

Acknowledgements

I thank Kevin Crow from StataCorp and the participants of the German Stata Users Group Meeting 2012 for helpful comments.

DISCLAIMER

References

Schneider, D.C. (2012). Modular Programming in Stata. Presentation at the German Stata Users Group Meeting 2012, Berlin. Available at http://www.stata.com/meeting/germany12/abstracts/desug12_schneider.pdf.

Also see

Online: **[R] net**

User-written, if installed:

adolist -help- -install- -remote help-