STATA Resources, Unit 4, Section 5.1
Programs and `gettoken`
Kyle Puetz • 4/2/13

This guide provides information on the `gettoken` function in STATA and in particular how it can be used in conjunction with the `program` function to manipulate your data.

Though we have covered the `program` function in previous guides, it is probably worthwhile to return it to it briefly.

`program` allows the analyst or coder to generate a new command, which itself consists of a prescribed sequence of other commands. This new command may then be used in conjunction with different variables or even be used to generate new local macros. In essence, new variables or elements can be chosen after the researcher has defined the program — that is, we may use the newly generated program with different elements even without altering any part of the program itself.

Consequently, `program` is a very powerful way of automating your work in STATA. Programs are capable of allowing you to perform very intricate tasks in a very parsimonious way. The amount of coding to allow you to perform some tasks outside of a program can be burdensome, and using `program` also makes it less likely that you will unintentionally introduce errors into your data manipulations.

Here is one example of how to code with program, provided in a previous guide in this series:

```
1 program list_args
2     display "The whole argument you typed is: `0'"
3     display "The 1st argument you typed is: `1'"
4     display "The 2nd argument you typed is: `2'"
5     display "The 3rd argument you typed is: `3'"
6     display "The 4th argument you typed is: `4'"
7 end
```

In this example, the `program` command indicates that we are about to code a new program. The lines that follow, which here are indented, show the sequence of commands that the program, if used, will run. Finally, the `end` command indicates to STATA to stop learning the new program. If you forget the `end` command, STATA will in effect continue to recognize all arguments and commands as part of the program.

It is important to explain the numbers in the example above because they are immediately relevant to `gettoken` as well. Let's provide an example. What if I were to enter `list_args here is an example` into STATA?

```
. list_args here is an example
The whole argument you typed is: here is an example
The 1st argument you typed is: here
The 2nd argument you typed is: is
The 3rd argument you typed is: an
The 4th argument you typed is: example
```

Now it's become clearer that the `` `0' `` is a macro that refers to the whole argument `here is an example`. Similarly, each number that follows numerically refers to the corresponding element in the argument (1 = `here`; 2 = `is`; 3 = `an`; 4 = `example`).

Keep in mind that once a new command has been generated by the `program` function, STATA will deliver an error if we repeat the code that generates the program:

```
. program list_args
list_args already defined
r(110);
```

For this reason, it is essential while writing do-files to drop programs before writing them anew (or changing the associated sequence of commands). In this instance, this can be achieved simply by writing `program drop list_args`. Unfortunately, whenever we use the `program drop` command and the program does not yet exist, this generates a new error:

```
. program drop list_args
list_args not found
r(111);
```

The solution is to use preface the `program drop` command with the `capture` command: `capture program drop`. This allows the rest of the do-file to run even in instances that we have not previously produced this program.

Now that we have reacquainted ourselves with how to program in STATA, it is now time to ask: How does the `gettoken` function work? `gettoken` is similar to the `pop` function in Python and other programming languages in that it allows us to very efficiently remove the initial element from an argument while leaving the rest of the argument intact.

The syntax for `gettoken`, described [in the manual for STATA 13](), is:

```
gettoken emname1 [emname2] : emname3 [, parse("pchars") quotes
    qed(lmacname) match(lmacname) bind]
```

The manual goes on to describe how `gettoken` works:

> "`gettoken` obtains the next token from the macro *emname3* and stores it in the macro *emname1*. If macro *emname2* is specified, the rest of the string from *emname3* is stored in the *emname2* macro. *emname1* and *emname3*, or *emname2* and *emname3*, may be the same name. The first token is determined based on the parsing characters *pchars*, which default to a space if not specified."

An example:

```
1 capture program drop list_args2
2 program list_args2
3     gettoken first 0 : 0
4     display "The 1st argument you typed is `first'
5     display "Here is what was left:  `0'"
6 end
```

```
. list_args2 here is an example
The 1st argument you typed is here
Here is what was left:   is an example
```

In this example, we have used `first` to represent the first element of the argument that we want to remove from the rest of the argument (though, as the description above indicates, we could use other names as well). The `gettoken` command has removed the initial element from the `` `0' `` and ultimately has placed the remaining elements the argument (those which we have not parsed out) back in the `` `0' `` macro.

`gettoken` becomes particularly helpful when we would like to subject every remaining element to the same procedure in STATA. For example:

```
capture program drop list_args2
program list_args2
    gettoken first 0 : 0
    display "The 1st argument you typed is: `first'"
    foreach rest of local 0 {
        display "The rest of what you typed is: `rest'"
    }
end
```

Below, we see the results of the code we have written:

```
. list_args2 this is an example
The 1st argument you typed is: this
The rest of what you typed is: is
The rest of what you typed is: an
The rest of what you typed is: example
```

In short, each individual element remaining in macro `0' has been independently subjected to the code we have written. Though this example isn't particularly useful, it is not difficult to imagine how using `display` in conjunction with loops can help with the automation of data manipulation. (Like the `first' macro, we do not have to name `rest' rest but could feasibly give it a more appropriate name, depending upon what variables or values it is representing.)

At this point, it may be helpful to note that we don't necessarily need to place the remaining argument back in the initially constructed macro `0':

```
capture program drop list_args2
program list_args2
    gettoken first 1 : 0
    display "The 1st argument you typed is `first'
    display "Here is the full argument:  `0'"
    display "The remaining argument is: `1'"
end
```

That is, if it becomes confusing which of the initial elements have been placed back into macro `0', it may be worthwhile simply to generate a new macro and use that within your code, thus keeping the original `0' macro intact:

```
. list_args2 here is an example
The 1st argument you typed is here
Here is the full argument:  here is an example
The remaining argument is:  is an example
```

Now, to provide a more complex example. For this example, I am using the dynamics of collective action dataset, available here. Derived from newspaper accounts, this dataset contains information, among other things, on the states in which each act of protest occurred as well as the types of claim protest participants invoked. Let's consider a scenario in which we would want to create a new variable indicating that a particular protest event took place in a particular state and evinced one of several types of claim. How can we generate a new program that will allow us to very easily generate new sets of variables, each of which specifies a different state and set of claims as its object of interest?

(Note that, instead of naming the macro generated via the `gettoken` command `first`, as we have done previously, we have named it `state` in order to more accurately characterize the data. Likewise, rather than naming the elements in local `` `0' `` `rest`, we have instead named them `claims`.)

```
1  capture program drop claimscript
2  program define claimscript
3      gettoken state 0 : 0
4      display "The 1st argument you typed is `state'
5      display "Here is what was left:  `0'"
6      generate `state'_claims = 0
7      forvalues v = 1/4 {
8          foreach claim of local 0 {
9              replace `state'_claims=1 if state1=="`first'" & claim`v'==`claim'
10         }
11     }
12 end
```

Let's take this code line by line.

- **Line 1**: The `capture program drop` command drops the `claimscript` program if it already exists. If it does not exist, the `capture` command prevents the script from breaking.
- **Line 2**: Here, we indicate that the following sequence of lines will constitute the new `claimscript` command.
- **Line 3**: The `gettoken` command indicates that the first element should be excised from the full set of elements and generates a macro called `state` that will stand in for this element for the remainder of the program.
- **Lines 4-5**: This is simply a test: It will show us whether the set of elements has been parsed in the way that we expect.
- **Line 6**: Here, we create a dummy variable. Recall that we want to create a variable that indicates when a particular protest event took place in a particular state and evinced one of several types of claim. Created here, the dummy variable initially has 0 cases which fit. But that is about to change…
- **Line 7-11**: Here, we create a first loop. Remember, in the dataset we are using, there are four claims variables, and we want to check all of them for the values of interest to us.
- **Line 8-10**: Here is a second loop. We recognize that it is using the local `` `0' `` generated by the `gettoken` command (i.e., the macro that has been parsed). Also, note that we are identifying each individual element within local `` `0' `` as a `claim`. This loop is

identifying each case that takes place in the state that we've specified *and* whose protestors' claims are identified as one of the claims we are interested in. Used in conjunction with the previous loop, it does this for variables `claim1`, `claim2`, `claim3`, and `claim4`.

- **Line 12**: This lets STATA know that the `claimscript` program is complete.

What if we are interested in creating a variable that indicates that a certain kind of claim took place in California? Then we simply use our program and name the state of interest, followed by some (arbitrarily chosen) claim types of interest: `claimscript CA 1500 1501 1502`.

```
. claimscript CA 1500 1501 1502
The 1st argument you typed is CA
Here is what was left:   1500 1501 1502
(71 real changes made)
(31 real changes made)
(1 real change made)
(18 real changes made)
(3 real changes made)
(1 real change made)
(1 real change made)
(0 real changes made)
(0 real changes made)
(0 real changes made)
(0 real changes made)
(0 real changes made)
```

This command subjects the data to 12 rounds of possible changes, as it looks for the presence of claim type 1500, 1501, and 1502 in the `claim1`, `claim2`, `claim3`, and `claim4` variables. At this point, it might be helpful to note that we are really not limited in the number of claims we could specify, and we could easily expand the list of claims to a much larger number. We could specify one kind of claim; we could specify dozens of types of claim.

As always, it is worthwhile to check the accuracy of the results. The `claimscript` function is helpful in that we can provide crosstabs for `state1` versus our newly constructed variable. We expect that all values for our new variable should be limited to California, and, indeed, that is what we find:

```
. tab state1 CA_claims
           |        CA_claims
    state1 |         0          1 |     Total
-----------+----------------------+----------
        AK |        40          0 |        40
        AL |       570          0 |       570
        AR |        77          0 |        77
        AZ |        73          0 |        73
```

```
        CA |      1,272          126 |      1,398
        CO |        127            0 |        127
           |                    ...          ...
        WA |        145            0 |        145
        WI |        212            0 |        212
        WV |         58            0 |         58
        WY |         21            0 |         21
-----------+----------------------+-----------
     Total |     23,195          126 |     23,321
```

Finally, it is probably worthwhile to note that it is also possible to perform nested `gettokens`. That is, once you've used the `gettoken` function, there is nothing to prevent you from using it again immediately after. Essentially, this allows you to parse several elements from a single initial argument:

```
capture program drop list_args3
program list_args3
    gettoken first 0 : 0
        display "The 1st argument you typed is: `first'"
        display "Here is what was left:  `0'"
    gettoken second 0 : 0
        display "The 1st argument you typed is: `first'"
        display "The 2nd argument you typed is: `second'"
        display "Here is what was left:  `0'"
    gettoken third 0 : 0
        display "The 1st argument you typed is: `first'"
        display "The 2nd argument you typed is: `second'"
        display "The 3rd argument you typed is: `third'"
        display "Here is what was left:  `0'"
    gettoken fourth 0 : 0
        display "The 1st argument you typed is: `first'"
        display "The 2nd argument you typed is: `second'"
        display "The 3rd argument you typed is: `third'"
        display "The 4th argument you typed is: `fourth'"
        display "Here is what was left:  `0'"
    gettoken fifth 0 : 0
        display "The 1st argument you typed is: `first'"
        display "The 2nd argument you typed is: `second'"
        display "The 3rd argument you typed is: `third'"
        display "The 4th argument you typed is: `fourth'"
        display "The 5th argument you typed is: `fifth'"
        display "Here is what was left:  `0'"
end
```

Now, what happens when we run the following command: `here is a very long example`?

```
. list_args3 here is a long example
The 1st argument you typed is: here
Here is what was left:   is a long example
The 1st argument you typed is: here
The 2nd argument you typed is: is
Here is what was left:   a long example
The 1st argument you typed is: here
The 2nd argument you typed is: is
The 3rd argument you typed is: a
Here is what was left:   long example
The 1st argument you typed is: here
The 2nd argument you typed is: is
The 3rd argument you typed is: a
The 4th argument you typed is: long
Here is what was left:   example
The 1st argument you typed is: here
The 2nd argument you typed is: is
The 3rd argument you typed is: a
The 4th argument you typed is: long
The 5th argument you typed is: example
Here is what was left:
```

In essence, nested `gettoken`s can allow you to go through every element in an argument until you've exhausted all of them.