Title

gettoken — Low-level parsing

Syntax

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

where *pchars* are the parsing characters, *lmacname* is a local macro name, and *emname* is described in the following table:

emname is ...	Refers to a ...
macroname	local macro
`(local)` *macroname*	local macro
`(global)` *macroname*	global macro

Description

`gettoken` is a low-level parsing command designed for programmers who wish to parse input for themselves. The `syntax` command (see [P] **syntax**) is an easier-to-use, high-level parsing command.

`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.

Options

`parse("`*pchars*`")` specifies the parsing characters. If `parse()` is not specified, `parse(" ")` is assumed, meaning that tokens are identified by blanks.

`quotes` indicates that the outside quotes are not to be stripped in what is stored in *emname1*. This option has no effect on what is stored in *emname2* because it always retains outside quotes. `quotes` is a rarely specified option; usually you want the quotes stripped. You would not want the quotes stripped if you wanted to make a perfect copy of the contents of the original macro for parsing at a later time.

`qed(`*lmacname*`)` specifies a local macroname that is to be filled in with 1 or 0 according to whether the returned token was enclosed in quotes in the original string. `qed()` does not change how parsing is done; it merely returns more information.

`match(`*lmacname*`)` specifies that parentheses be matched in determining the token. The outer level of parentheses, if any, are removed before the token is stored in *emname1*. The local macro *lmacname* is set to “(” if parentheses were found; otherwise, it is set to an empty string.

`bind` specifies that expressions within parentheses and those within brackets are to be bound together, even when not parsing on `()` and `[]`.

Remarks and examples

stata.com

Often we apply gettoken to the macro ‘0’ (see [U] 18.4.6 Parsing nonstandard syntax), as in

```
gettoken first : 0
```

which obtains the first token (with spaces as token delimiters) from ‘0’ and leaves ‘0’ unchanged. Or, alternatively,

```
gettoken first 0 : 0
```

which obtains the first token from ‘0’ and saves the rest back in ‘0’.

▷ Example 1

Even though gettoken is typically used as a programming command, we demonstrate its use interactively:

```
. local str "cat+dog   mouse++horse"
. gettoken left : str
. display ‘"‘left’"’
cat+dog
. display ‘"‘str’"’
cat+dog   mouse++horse
. gettoken left str : str, parse(" +")
. display ‘"‘left’"’
cat
. display ‘"‘str’"’
+dog   mouse++horse
. gettoken next str : str, parse(" +")
. display ‘"‘next’"’
+
. display ‘"‘str’"’
dog   mouse++horse
```

Both global and local variables may be used with gettoken. Strings with nested quotes are also allowed, and the quotes option may be specified if desired. For more information on compound double quotes, see [U] 18.3.5 Double quotes.

```
. global weird ‘"‘""some" strings"’ are ‘"within "strings""’"’
. gettoken (local)left (global)right : (global)weird
. display ‘"‘left’"’
"some" strings
. display ‘"$right"’
 are ‘"within "strings""’
. gettoken left (global)right : (global)weird , quotes
. display ‘"‘left’"’
‘""some" strings"’
. display ‘"$right"’
 are ‘"within "strings""’
```

The match() option is illustrated below.

```
. local pstr "(a (b c)) ((d e f) g h)"
. gettoken left right : pstr
. display `"`left'"'
(a
. display `"`right'"'
 (b c)) ((d e f) g h)
. gettoken left right : pstr , match(parns)
. display `"`left'"'
a (b c)
. display `"`right'"'
 ((d e f) g h)
. display `"`parns'"'
(
```

◁

▷ Example 2

One use of `gettoken` is to process two-word commands. For example, `mycmd list` does one thing and `mycmd generate` does another. We wish to obtain the word following `mycmd`, examine it, and call the appropriate subroutine with a perfect copy of what followed.

```
program mycmd
        version 13
        gettoken subcmd 0 : 0
        if "`subcmd'" == "list" {
                mycmd_l `0'
        }
        else if "`subcmd'" == "generate" {
                mycmd_g `0'
        }
        else    error 199
end

program mycmd_l
        ...
end

program mycmd_g
        ...
end
```

◁

▷ Example 3

Suppose that we wish to create a general prefix command with the syntax

`newcmd ... :` *stata_command*

where ... represents some possibly complicated syntax. We want to split this entire command line at the colon, making a perfect copy of what precedes the colon, which will be parsed by our program, and what follows the colon, which will be passed along to *stata_command*.

```
program newcmd
        version 13
        gettoken part 0 : 0, parse(" :") quotes
        while `"`part'"' != ":" & `"`part'"' != "" {
                local left `"`left' `part'"'
                gettoken part 0 : 0, parse(" :") quotes
        }

        (`left' now contains what followed newcmd up to the colon)
        (`0' now contains what followed the colon)

        ...
end
```

Notice the use of the `quotes` option. We also used compound double quotes when accessing `` `part' `` and `` `left' `` because these macros might contain embedded quotation marks.

◁

❑ Technical note

We strongly encourage you to specify space as one of your parsing characters. For instance, with the last example, you may have been tempted to use `gettoken` but to parse only on colon instead of on colon and space, as in

```
gettoken left 0 : 0, parse(":") quotes
gettoken colon 0 : 0, parse(":")
```

and thereby avoid the `while` loop. This is not guaranteed to work for two reasons. First, if the length of the string up to the colon is large, then you run the risk of having it truncated. Second, if `` `left' `` begins with a quotation mark, then the result will not be what you expect.

Our recommendation is always to specify a space as one of your parsing characters and to grow your desired macro as demonstrated in our last example.

❑

❑ Technical note

If one of the parsing characters specified is the equal sign, for example, `parse("= ")`, then not only is the equal sign treated as one token, but so is Stata's equality operator, `==`. For instance, parsing "`y=x if z==3`" results in the tokens "`y`", "`=`", "`x`", "`if`", "`z`", "`==`", and "`3`".

❑

Also see

[P] **syntax** — Parse Stata syntax

[P] **tokenize** — Divide strings into tokens

[P] **while** — Looping

[U] 18 Programming Stata