Title

stata.com

tokenget( ) — Advanced parsing

Description

These functions provide advanced parsing. If you simply wish to convert strings into row vectors by separating on blanks, converting `"mpg weight displ"` into `("mpg", "weight", "displ")`, see [M-5] **tokens( )**.

Syntax

t = `tokeninit(`[*wchars* [, *pchars* [, *qchars* [, *allownum* [, *allowhex*]]]]] `)`

t = `tokeninitstata()`

void `tokenset(`*t*`,` *string scalar s*`)`

string rowvector `tokengetall(`*t*`)`

string scalar `tokenget(`*t*`)`

string scalar `tokenpeek(`*t*`)`

string scalar `tokenrest(`*t*`)`

real scalar `tokenoffset(`*t*`)`

void `tokenoffset(`*t*`,` *real scalar offset*`)`

string scalar `tokenwchars(`*t*`)`

void `tokenwchars(`*t*`,` *string scalar wchars*`)`

string rowvector `tokenpchars(`*t*`)`

void `tokenpchars(`*t*`,` *string rowvector pchars*`)`

string rowvector `tokenqchars(`*t*`)`

void `tokenqchars(`*t*`,` *string rowvector qchars*`)`

real scalar `tokenallownum(`*t*`)`

void `tokenallownum(`*t*`,` *real scalar allownum*`)`

real scalar `tokenallowhex(`*t*`)`

void `tokenallowhex(`*t*`,` *real scalar allowhex*`)`

where

t is *transmorphic* and contains the parsing environment information. You obtain a *t* from `tokeninit()` or `tokeninitstata()` and then pass *t* to the other functions.

wchars is a *string scalar* containing the characters to be treated as whitespace, such as `" "`, `(" "+char(9))`, or `""`.

pchars is a *string rowvector* containing the strings to be treated as parsing characters, such as `""` and `(">", "<", ">=", "<=")`. `""` and `J(1,0,"")` are given the same interpretation: there are no parsing characters.

qchars is a *string rowvector* containing the character pairs to be treated as quote characters. `""` (that is, empty string) is given the same interpretation as `J(1,0,"")`; there are no quote characters. *qchars* = `(`‘`""""`’`)` (that is, the two-character string quote indicates that `"` is to be treated as open quote and `"` is to be treated as close quote. *qchars* = `(`‘`""""`’`,` ‘`""`‘`""`’`"`’`)` indicates that, in addition, ‘`"` is to be treated as open quote and `"`’ as close quote. In a syntax that did not use < and > as parsing characters, *qchars* = `("<>")` would indicate that < is to be treated as open quote and > as close quote.

allownum is a *string scalar* containing 0 or 1. *allownum* = 1 indicates that numbers such as 12.23 and 1.52e+02 are to be returned as single tokens even in violation of other parsing rules.

allowhex is a *string scalar* containing 0 or 1. *allowhex* = 1 indicates that numbers such as 1.921fb54442d18X+001 and 1.0x+a are to be returned as single tokens even in violation of other parsing rules.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Concepts

Parsing refers to splitting a string into pieces, which we will call tokens. Parsing as implemented by the `token*()` functions is defined by (1) the whitespace characters *wchars*, (2) the parsing characters *pchars*, and (3) the quote characters *qchars*.

White-space characters

Consider the string `"this that what"`. If there are no whitespace characters, no parsing characters, and no quote characters, that is, if *wchars* = *pchars* = *qchars* = `""`, then the result of parsing `"this that what"` would be one token that would be the string just as it is: `"this that what"`.

If *wchars* were instead `" "`, then parsing `"this that what"` results in `("this", "that", "what")`. Parsing `"this that  what"` (note the multiple blanks) would result in the same thing. White-space characters separate one token from the next but are not otherwise significant.

Parsing characters

If we instead left *wchars* = `""` and set *pchars* = `" "`, `"this that what"` parses into `("this", " ", "that", " ", "what")` and parsing `"this that  what"` results in `("this", " ", "that", " ", " ", " ", "what")`.

pchars are like *wchars* except that they are themselves significant.

pchars do not usually contain space. A more reasonable definition of *pchars* is `("+", "-")`. Then parsing `"x+y"` results in `("x", "+", "y")`. Also, the parsing characters can be character combinations. If *pchars* = `("+", "-", "++", "--")`, then parsing `"x+y++"` results in `("x", "+", "y", "++")` and parsing `"x+++y"` results in `("x", "++", "+", "y")`. Longer *pchars* are matched before shorter ones regardless of the order in which they appear in the *pchars* vector.

Quote characters

qchars specifies the quote characters. Pieces of the string being parsed that are surrounded by quotes are returned as one token, ignoring the separation that would usually occur because of the *wchars* and *pchars* definitions. Consider the string

```
mystr= "x = y"
```

Let *wchars* = `" "` and *pchars* include `"="`. That by itself would result in the above string parsing into the five tokens

mystr	=	"x	=	y"

Now let *qchars* = `(‘""’)`; that is, *qchars* is the two-character string `""`. Parsing then results in the three tokens

mystr	=	"x = y"

Each element of *qchars* contains a character pair: the open character followed by the close character. We defined those two characters as `"` and `"` above, that is, as being the same. The two characters can differ. We might define the first as ‘ and the second as ’. When the characters are different, quotations can nest. The quotation `"he said "hello""` makes no sense because that parses into `("he said ", hello, "")`. The quotation ‘he said ‘hello’’, however, makes perfect sense and results in the single token ‘he said ‘hello’’.

The quote characters can themselves be multiple characters. You can define open quote as ‘" and close as "’: *qchars* = `(‘"‘""’"’)`. Or you can define multiple sets of quotation characters, such as *qchars* = `(‘"’, ‘"‘""’"’)`.

The quote characters do not even have to be quotes at all. In some context you might find it convenient to specify them as `("()")`. With that definition, "$(2 \times (3 + 2))$" would parse into one token. Specifying them like this can be useful, but in general we recommend against it. It is usually better to write your code so that quote characters really are quote characters and to push the work of handling other kinds of nested expressions back onto the caller.

Overrides

The `token*()` functions provide two overrides: *allownum* and *allowhex*. These have to do with parsing numbers. First, consider life without overrides. You have set *wchars* = `" "` and *pchars* = `("=", "+", "-", "*", "/")`. You attempt to parse

```
y = x + 1e+13
```

The result is

y	=	x	+	1e	+	13

when what you wanted was

y	=	x	+	1e+13

Setting *allownum* = 1 will achieve the desired result. *allownum* specifies that, when a token could be interpreted as a number, the number interpretation is to be taken even in violation of the other parsing rules.

Setting *allownum* = 1 will not find numbers buried in the middle of strings, such as the `1e+3` in `"xis1e+3"`, but if the number occurs at the beginning of the token according to the parsing rules set by *wchars* and *pchars*, *allownum* = 1 will continue the token in violation of those rules if that results in a valid number.

The override *allowhex* is similar and Stata specific. Stata (and Mata) provide a unique and useful way of writing hexadecimal floating-point numbers in a printable, short, and precise way: π can be written 1.921fb54442d18X+001. Setting *allowhex* = 1 allows such numbers.

Setting the environment to parse on blanks with quote binding

Stata's default rule for parsing do-file arguments is "parse on blanks and bind on quotes". The settings for duplicating that behavior are

wchars = `" "`

pchars = `( "" )`

qchars = `( `"""'`, `"`""'"' )`

allownum = 0

allowhex = 0

This behavior can be obtained by coding

```
t = tokeninit(" ", "", (`""""', `"`""'"'), 0, 0)
```

or by coding

```
t = tokeninit()
```

because in `tokeninit()` the arguments are optional and "parse on blank with quote binding" is the default.

With those settings, parsing `` `"first second "third fourth" fifth"' `` results in `("first", "second", `""third fourth""', "fifth")`.

This result is a little different from that of Stata because the third token includes the quote binding characters. Assume that the parsed string was obtained by coding

```
res = tokengetall(t)
```

The following code will remove the open and close quotes, should that be desirable.

```
for (i=1; i<=cols(res); i++) {
        if (substr(res[i], 1, 1)==`""""') {
                res[i] = substr(res[i], 2, strlen(res[i])-2)
        }
        else if (substr(res[i], 1, 2)=="`" + `""""') {
                res[i] = substr(res[i], 3, strlen(res[i])-4)
        }
}
```

Setting the environment to parse full Stata syntax

To parse full Stata syntax, the settings are

```
   wchars =  " "
   pchars =  (  "\",   "~",    "!",    "=",    ":",    ";",    ",",
                "?",   "!",    "@",    "#",   "==",   "!=",   ">=",
               "<=",   "<",    ">",    "&",    "|",   "&&",   "||",
                "+",   "-",   "++",   "--",    "*",    "/",    "^",
                "(",   ")",    "[",    "]",    "{",    "}" )
   qchars =  ( `""""', `"`""'"', char(96)+char(39))
allownum =  1
 allowhex =  1
```

The above is a slight oversimplification. Stata is an interpretive language and Stata does not require users to type filenames in quotes, although Stata does allow it. Thus `"\"` is sometimes a parsing character and sometimes not, and the same is true of `"/"`. As Stata parses a line from left to right, it will change *pchars* between two `tokenget()` calls when the next token could be or is known to be a filename. Sometimes Stata peeks ahead to decide which way to parse. You can do the same by using the `tokenpchars()` and `tokenpeek()` functions.

To obtain the above environment, code

```
t = tokeninitstata()
```

Setting the environment to parse tab-delimited files

The `token*()` functions can be used to parse lines from tab-delimited files. A tab-delimited file contains lines of the form

⟨*field1*⟩⟨*tab*⟩⟨*field2*⟩⟨*tab*⟩⟨*field3*⟩

The parsing environment variables are

wchars = `""`
pchars = `( char(9) )` (i.e., *tab*)
qchars = `( "" )`
allownum = 0
allowhex = 0

To set this environment, code

```
t = tokeninit("", char(9), "", 0, 0)
```

Say that you then parse the line

Farber, William⟨*tab*⟩ 2201.00⟨*tab*⟩12

The results will be

```
("Farber, William", char(9), " 2201.00", char(9), "12")
```

If the line were

Farber, William⟨*tab*⟩⟨*tab*⟩12

the result would be

```
("Farber, William", char(9), char(9), "12")
```

The tab-delimited format is not well defined when the missing fields occur at the end of the line. A line with the last field missing might be recorded

Farber, William⟨*tab*⟩ 2201.00⟨*tab*⟩

or

Farber, William⟨*tab*⟩ 2201.00

A line with the last two fields missing might be recorded

Farber, William⟨*tab*⟩⟨*tab*⟩

or

Farber, William⟨*tab*⟩

or

Farber, William

The following program would correctly parse lines with missing fields regardless of how they are recorded:

```
real rowvector readtabbed(transmorphic t, real scalar n)
{
        real scalar       i
        string rowvector  res
        string scalar     token
        res = J(1, n, "")
        i = 1
        while ((token = tokenget(t))!="") {
                if (token==char(9)) i++
                else res[i] = token
        }
        return(res)
}
```

Function overview

The basic way to proceed is to initialize the parsing environment and store it in a variable,

```
t = tokeninit(...)
```

and then set the string `s` to be parsed,

```
tokenset(t, s)
```

and finally use `tokenget()` to obtain the tokens one at a time (`tokenget()` returns `""` when the end of the line is reached), or obtain all the tokens at once using `tokengetall(t)`. That is, either

```
while((token = tokenget(t)) != "") {
        ... process token ...
}
```

or

```
tokens = tokengetall(t)
for (i=1; i<=cols(tokens); i++) {
        ... process tokens[i] ...
}
```

After that, set the next string to be parsed,

```
tokenset(t, nextstring)
```

and repeat.

tokeninit() and tokeninitstata()

`tokeninit()` and `tokeninitstata()` are alternatives. `tokeninitstata()` is generally unnecessary unless you are writing a fairly complicated function.

Whichever function you use, code

```
t = tokeninit(...)
```

or

```
t = tokeninitstata()
```

If you declare *t*, declare it `transmorphic`. *t* is in fact a structure containing all the details of your parsing environment, but that is purposely hidden from you so that you cannot accidentally modify the environment.

`tokeninit()` allows up to five arguments:

```
t = tokeninit(wchars, pchars, qchars, allownum, allowhex)
```

You may omit arguments from the end. If omitted, the default values of the arguments are

```
 allowhex = 0
allownum = 0
   qchars = ( `""""', `"`""'"' )
   pchars = ( "" )
   wchars = " "
```

Notes

1. Concerning *wchars*:
    a. *wchars* is a *string scalar*. The whitespace characters appear one after the other in the string. The order in which the characters appear is irrelevant.
    b. Specify *wchars* as `" "` to treat blank as whitespace.
    c. Specify *wchars* as `" "+char(9)` to treat blank and *tab* as whitespace. Including *tab* is necessary only when strings to be parsed are obtained from a file; strings obtained from Stata already have the *tab* characters removed.
    d. Any character can be treated as a whitespace character, including letters.
    e. Specify *wchars* as `""` to specify that there are no whitespace characters.
2. Concerning *pchars*:
    a. *pchars* is a *string rowvector*. Each element of the vector is a separate parse character. The order in which the parse characters are specified is irrelevant.
    b. Specify *pchars* as `("+", "-")` to make `+` and `-` parse characters.
    c. Parse characters may be character combinations such as `++` or `>=`. Character combinations may be up to four characters long.
    d. Specify *pchars* as `""` or `J(1,0,"")` to specify that there are no parse characters. It makes no difference which you specify, but you will realize that `J(1,0,"")` is more logically consistent if you think about it.
3. Concerning *qchars*:
    a. *qchars* is a `string rowvector`. Each element of the vector contains the open followed by the close characters. The order in which sets of quote characters are specified is irrelevant.

b. Specify *qchars* as `(‘"""’)` to make " an open and close character.

c. Specify *qchars* as `(‘"""’, ‘"‘""’"’)` to make "" and ‘""’ quote characters.

d. Individual quote characters can be up to two characters long.

e. Specify *qchars* as `""` or `J(1,0,"")` to specify that there are no quote characters.

tokenset()

After `tokeninit()` or `tokeninitstata()`, you are not yet through with initialization. You must `tokenset(`*s*`)` to specify the string scalar you wish to parse. You `tokenset()` one line, parse it, and if you have more lines, you `tokenset()` again and repeat the process. Often you will need to parse only one line. Perhaps you wish to write a program to parse the argument of a complicated option in a Stata ado-file. The structure is

```
program ...
        ...
        syntax ... [, ... MYoption(string) ...]
        mata: parseoption(‘"‘myoption’"’)
        ...
end

mata:
void parseoption(string scalar option)
{
        transmorphic     t
        t = tokeninit(...)
        tokenset(t, option)
        ...
}
end
```

Notes

1. When you `tokenset(`*s*`)`, the contents of *s* are not stored. Instead, a pointer to *s* is stored. This approach saves memory and time, but it means that if you change *s* after setting it, you will change the subsequent behavior of the `token*()` functions.

2. Simply changing *s* is not sufficient to restart parsing. If you change *s*, you must `tokenset(`*s*`)` again.

tokengetall()

You have two alternatives in how to process the tokens. You can parse the entire line into a row vector containing all the individual tokens by using `tokengetall()`,

```
tokens = tokengetall(t)
```

or you can use `tokenget()` to process the tokens one at a time, which is discussed in the next section.

Using `tokengetall()`, `tokens[1]` will be the first token, `tokens[2]` the second, and so on. There are, in total, `cols(tokens)` tokens. If the line was empty or contained only whitespace characters, `cols(tokens)` will be 0.

tokenget(), tokenpeek(), and tokenrest()

`tokenget()` returns the tokens one at a time and returns `""` when the end of the line is reached. The basic loop for processing all the tokens in a line is

```
while ( (token = tokenget(t)) != "") {
                ...
}
```

`tokenpeek()` allows you to peek ahead at the next token without actually getting it, so whatever is returned will be returned again by the next call to `tokenget()`. `tokenpeek()` is suitable only for obtaining the next token after `tokenget()`. Calling `tokenpeek()` twice in a row will not return the next two tokens; it will return the next token twice. To obtain the next two tokens, code

```
...
current = tokenget(t)          // get the current token
...
t2 = t                         // copy parse environment
next_1 = tokenget(t2)          // peek at next token
next_1 = tokenget(t2)          // peek at token after that
...
current = tokenget(t)          // get next token
```

If you declare *t2*, declare it `transmorphic`.

`tokenrest()` returns the unparsed portion of the `tokenset()` string. Assume that you have just gotten the first token by using `tokenget()`. `tokenrest()` would return the rest of the original string, following the first token, unparsed. `tokenrest(`*t*`)` returns `substr(`*original_string*`, tokenoffset(`*t*`), .)`.

tokenoffset()

`tokenoffset()` is useful only when you are using the `tokenget()` rather than `tokengetall()` style of programming. Let the original string you `tokenset()` be "this is an example". Right after you have `tokenset()` this string, `tokenoffset()` is 1:

```
      this is an example
      |
tokenoffset() = 1
```

After getting the first token (say it is `"this"`), `tokenoffset()` is 5:

```
this is an example
    |
tokenoffset() = 5
```

`tokenoffset()` is always located on the first character following the last character parsed.

The syntax of `tokenoffset()` is

```
tokenoffset(t)
```

and

```
tokenoffset(t, newoffset)
```

The first returns the current offset value. The second resets the parser's location within the string.

tokenwchars(), tokenpchars(), and tokenqchars()

`tokenwchars()`, `tokenpchars()`, and `tokenqchars()` allow resetting the current *wchars*, *pchars*, and *qchars*. As with `tokenoffset()`, they come in two syntaxes.

With one argument, *t*, they return the current value of the setting. With two arguments, *t* and *newvalue*, they reset the value.

Resetting in the midst of parsing is an advanced issue. The most useful of these functions is `tokenpchars()`, since for interactive grammars, it is sometimes necessary to switch on and off a certain parsing character such as `/`, which in one context means division and in another is a file separator.

tokenallownum and tokenallowhex()

These two functions allow obtaining the current values of *allownum* and *allowhex* and resetting them.

Conformability

`tokeninit(`*wchars*`, `*pchars*`, `*qchars*`, `*allownum*`, `*allowhex*`)`:

wchars:	1×1	(optional)
pchars:	$1 \times c_p$	(optional)
qchars:	$1 \times c_q$	(optional)
allownum:	1×1	(optional)
allowhex:	1×1	(optional)
result:	*transmorphic*	

`tokeninitstata()`:

result:	*transmorphic*

`tokenset(`*t*`, `*s*`)`:

t:	*transmorphic*
s:	1×1
result:	*void*

`tokengetall(`*t*`)`:

t:	*transmorphic*
result:	$1 \times k$

`tokenget(`*t*`)`, `tokenpeek(`*t*`)`, `tokenrest(`*t*`)`:

t:	*transmorphic*
result:	1×1

`tokenoffset(`*t*`)`, `tokenwchars(`*t*`)`, `tokenallownum(`*t*`)`, `tokenallowhex(`*t*`)`:

t:	*transmorphic*
result:	1×1

`tokenoffset(`*t*`,` *newvalue*`)`, `tokenwchars(`*t*`,` *newvalue*`)`,
`tokenallownum(`*t*`,` *newvalue*`)`, `tokenallowhex(`*t*`,` *newvalue*`)`:

t:	*transmorphic*
newvalue:	1×1
result:	*void*

`tokenpchars(`*t*`)`, `tokenqchars(`*t*`)`:

t:	*transmorphic*
result:	$1 \times c$

`tokenpchars(`*t*`,` *newvalue*`)`, `tokenqchars(`*t*`,` *newvalue*`)`:

t:	*transmorphic*
newvalue:	$1 \times c$
result:	*void*

Diagnostics

None.

Also see

[M-5] **invtokens( )** — Concatenate string rowvector into string scalar

[M-5] **tokens( )** — Obtain tokens from string

[M-5] **ustrword( )** — Obtain Unicode word from Unicode string

[M-4] **programming** — Programming functions

[M-4] **string** — String manipulation functions

[P] **gettoken** — Low-level parsing

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