Title

stata.com

syntax — Parse Stata syntax

Description Syntax Syntax, continued Remarks and examples Also see

Description

There are two ways that a Stata program can interpret what the user types:

1. positionally, meaning first argument, second argument, and so on, or
2. according to a grammar, such as standard Stata syntax.

`args` does the first. The first argument is assigned to *macroname1*, the second to *macroname2*, and so on. In the program, you later refer to the contents of the macros by enclosing their names in single quotes: `‘`*macroname1*`’`, `‘`*macroname2*`’`, . . . :

```
program myprog
        version 14.1
        args varname dof beta
        (the rest of the program would be coded in terms of ‘varname’, ‘dof’, and ‘beta’)
        ...
end
```

`syntax` does the second. You specify the new command's syntax on the `syntax` command; for instance, you might code

```
program myprog
        version 14.1
        syntax varlist [if] [in] [, DOF(integer 50) Beta(real 1.0)]
        (the rest of the program would be coded in terms of ‘varlist’, ‘if’, ‘in’, ‘dof’, and ‘beta’)
        ...
end
```

`syntax` examines what the user typed and attempts to match it to the syntax diagram. If it does not match, an error message is issued and the program is stopped (a nonzero return code is returned). If it does match, the individual components are stored in particular local macros where you can subsequently access them. In the example above, the result would be to define the local macros `‘varlist’`, `‘if’`, `‘in’`, `‘dof’`, and `‘beta’`.

For an introduction to Stata programming, see **[U] 18 Programming Stata** and especially **[U] 18.4 Program arguments**.

Standard Stata syntax is

cmd [*varlist* | *namelist* | *anything*]
[*if*]
[*in*]
[`using` *filename*]
[`=` *exp*]
[*weight*]
[`,` *options*]

Each of these building blocks, such as *varlist*, *namelist*, and `if`, is outlined below.

Syntax

Parse Stata syntax positionally

```
args macroname1 [ macroname2 [ macroname3 ... ] ]
```

Parse syntax according to a standard syntax grammar

```
syntax description_of_syntax
```

Syntax, continued

The *description_of_syntax* allowed by `syntax` includes

---

description_of_varlist:

type	*nothing*	
or		
optionally type	`[`	
then type one of	`varlist varname newvarlist newvarname`	
optionally type	(*varlist_specifiers*)	
type	`]`	(if you typed `[` at the start)

varlist_specifiers are

```
default=none  min=#  max=#  numeric
string  str#  strL  fv  ts
generate    (newvarlist and newvarname only)
```

Examples:

```
syntax varlist ...
syntax [varlist] ...
syntax varlist(min=2) ...
syntax varlist(max=4) ...
syntax varlist(min=2 max=4 numeric) ...
syntax varlist(default=none) ...

syntax newvarlist(max=1) ...

syntax varname ...
syntax [varname] ...
```

If you type nothing, the command does not allow a varlist.

Typing `[` and `]` means that the varlist is optional.

`default=` specifies how the varlist is to be filled in when the varlist is optional and the user does not specify it. The default is to fill it in with all the variables. If `default=none` is specified, it is left empty.

`min=` and `max=` specify the minimum and maximum number of variables that may be specified. Typing `varname` is equivalent to typing `varlist(max=1)`.

`numeric`, `string`, `str#`, and `strL` restrict the specified varlist to consist of entirely numeric, entirely string (meaning `str#` or `strL`), entirely `str#`, or entirely `strL` variables.

`fv` allows the varlist to contain factor variables.

`ts` allows the varlist to contain time-series operators.

`generate` specifies, for `newvarlist` or `newvarname`, that the new variables be created and filled in with missing values.

After the `syntax` command, the resulting varlist is returned in `‘varlist’`. If there are new variables (you coded `newvarname` or `newvarlist`), the macro `‘typlist’` is also defined, containing the storage type of each new variable, listed one after the other.

---

description_of_namelist:

type	*nothing*	
or		
optionally type	`[`	
then type one of	`namelist name`	
optionally type	(*namelist_specifiers*)	
type	`]`	(if you typed `[` at the start)

namelist_specifiers are `name=`*name* `id="`*text*`" local`
`min=#` (`namelist` only) `max=#` (`namelist` only)

Examples:

```
syntax namelist ...
syntax [namelist] ...
syntax name(id="equation name") ...
syntax [namelist(id="equation name")] ...
syntax namelist(name=eqlist id="equation list")...
syntax [name(name=eqname id="equation name")] ...
syntax namelist(min=2 max=2) ...
```

`namelist` is an alternative to `varlist`; it relaxes the restriction that the names the user specifies be of variables. `name` is a shorthand for `namelist(max=1)`.

`namelist` is for use when you want the command to have the nearly standard syntax of command name followed by a list of names (not necessarily variable names), followed by `if`, `in`, *options*, etc. For instance, perhaps the command is to be followed by a list of variable-label names.

If you type nothing, the command does not allow a namelist. Typing `[` and `]` means that the namelist is optional. After the `syntax` command, the resulting namelist is returned in `‘namelist’` unless `name=`*name* is specified, in which case the result is returned in `‘`*name*`’`.

`id=` specifies the name of namelist and is used in error messages. The default is `id=namelist`. If `namelist` were required and `id=` was not specified, and the user typed “`mycmd if...`” (omitting the namelist), the error message would be “namelist required”. If you specified `id="equation name"`, the error message would be “equation name required”.

`name=` specifies the name of the local macro to receive the namelist; not specifying the option is equivalent to specifying `name=namelist`.

`local` specifies that the names that the user specifies satisfy the naming convention for local macro names. If this option is not specified, standard naming convention is used (names may begin with a letter or underscore, may thereafter also include numbers, and must not be longer than 32 characters). If the user specifies an invalid name, an error message will be issued. If `local` is specified, specified names are allowed to begin with numbers but may not be longer than 31 characters.

description_of_anything:

type	*nothing*	
or		
optionally type	`[`	
type	`anything`	
optionally type	(*anything_specifiers*)	
type	`]`	(if you typed `[` at the start)

anything_specifiers are `name=`*name* `id="`*text*`"` `equalok` `everything`

Examples:

```
syntax anything ...
syntax [anything] ...
syntax anything(id="equation name") ...
syntax [anything(id="equation name")] ...
syntax anything(name=eqlist id="equation list") ...
syntax [anything(name=eqlist id="equation list")] ...
syntax anything(equalok) ...
syntax anything(everything) ...
syntax [anything(name=0 id=clist equalok)] ...
```

`anything` is for use when you want the command to have the nearly standard syntax of command name followed by something followed by `if`, `in`, *options*, etc. For instance, perhaps the command is to be followed by an expression or expressions or a list of numbers.

If you type nothing, the command does not allow an "anything". Typing `[` and `]` means the "anything" is optional. After the `syntax` command, the resulting "anything list" is returned in `‘anything’` unless `name=`*name* is specified, in which case the result is returned in `‘name’`.

`id=` specifies the name of "anything" and is used only in error messages. For instance, if `anything` were required and `id=` was not specified, and the user typed "`mycmd if...`" (omitting the "anything"), the error message would be "something required". If you specified `id="expression list"`, the error message would be "expression list required".

`name=` specifies the name of the local macro to receive the "anything"; not specifying the option is equivalent to specifying `name=anything`.

`equalok` specifies that `=` is not to be treated as part of `=`*exp* in subsequent standard syntax but instead as part of the `anything`.

`everything` specifies that `if`, `in`, and `using` are not to be treated as part of standard syntax but instead as part of the `anything`.

`varlist`, `varname`, `namelist`, `name`, and `anything` are alternatives; you may specify at most one.

description_of_if:

type	*nothing*	
or		
optionally type	`[`	
type	`if`	
optionally type	`/`	
type	`]`	(if you typed `[` at the start)

Examples:

```
syntax ... if ...
syntax ... [if] ...
syntax ... [if/] ...
syntax ... if/ ...
```

If you type nothing, the command does not allow an `if` *exp*.

Typing `[` and `]` means that the `if` *exp* varlist is optional.

After the `syntax` command, the resulting `if` *exp* is returned in `‘if’`. The macro contains `if` followed by the expression, unless you specified `/`, in which case the macro contains just the expression.

description_of_in:

type	*nothing*	
or		
optionally type	`[`	
type	`in`	
optionally type	`/`	
type	`]`	(if you typed `[` at the start)

Examples:
```
syntax ... in ...
syntax ... [in] ...
syntax ... [in/] ...
syntax ... in/ ...
```

If you type nothing, the command does not allow an `in` *range*.

Typing `[` and `]` means that the `in` *range* is optional.

After the `syntax` command, the resulting `in` *range* is returned in `‘in’`. The macro contains `in` followed by the range, unless you specified `/`, in which case the macro contains just the range.

---

description_of_using:

type	*nothing*	
or		
optionally type	`[`	
type	`using`	
optionally type	`/`	
type	`]`	(if you typed `[` at the start)

Examples:
```
syntax ... using ...
syntax ... [using] ...
syntax ... [using/] ...
syntax ... using/ ...
```

If you type nothing, the command does not allow `using` *filename*.

Typing `[` and `]` means that the `using` *filename* is optional.

After the `syntax` command, the resulting filename is returned in `‘using’`. The macro contains `using` followed by the filename in quotes, unless you specified `/`, in which case the macro contains just the filename without quotes.

---

description_of_=exp:

type	*nothing*	
or		
optionally type	`[`	
type	`=`	
optionally type	`/`	
type	`exp`	
type	`]`	(if you typed `[` at the start)

Examples:
```
syntax ... =exp ...
syntax ... [=exp] ...
syntax ... [=/exp] ...
syntax ... =/exp ...
```

If you type nothing, the command does not allow an `=`*exp*.

Typing `[` and `]` means that the `=`*exp* is optional.

After the `syntax` command, the resulting expression is returned in `‘exp’`. The macro contains `=`, a space, and the expression, unless you specified `/`, in which case the macro contains just the expression.

---

description_of_weights:

type	*nothing*	
or		
type	[	
type any of	fweight aweight pweight iweight	
optionally type	/	
type	]	

Examples:
```
syntax ... [fweight] ...
syntax ... [fweight pweight] ...
syntax ... [pweight fweight] ...
syntax ... [fweight pweight iweight/] ...
```

If you type nothing, the command does not allow weights. A command may not allow both a weight and =*exp*.

You must type [and]; they are not optional. Weights are always optional.

The first weight specified is the default weight type.

After the `syntax` command, the resulting weight and expression are returned in `‘weight’` and `‘exp’`. `‘weight’` contains the weight type or nothing if no weights were specified. `‘exp’` contains =, a space, and the expression, unless you specified /, in which case the macro contains just the expression.

description_of_options:

type	*nothing*	
or		
type	[,	
type	*option_descriptors*	(these options will be optional)
optionally type	*	
type	]	
or		
type	,	
type	*option_descriptors*	(these options will be required)
optionally type	[	
optionally type	*option_descriptors*	(these options will be optional)
optionally type	*	
optionally type	]	

Examples:
```
syntax ... [, MYopt Thisopt]
syntax ..., MYopt Thisopt
syntax ..., MYopt [Thisopt]
syntax ... [, MYopt Thisopt *]
```

If you type nothing, the command does not allow options.

The brackets distinguish optional from required options. All options can be optional, all options can be required, or some can be optional and others be required.

After the `syntax` command, options are returned to you in local macros based on the first 31 letters of each option's name. If you also specify *, any remaining options are collected and placed, one after the other, in `‘options’`. If you do not specify *, an error is returned if the user specifies any options that you do not list.

option_descriptors include the following; they are documented below.

optionally_on
optionally_off
optional_integer_value
optional_real_value
optional_confidence_interval
optional_credible_interval
optional_numlist
optional_varlist
optional_namelist
optional_string
optional_passthru

option_descriptor optionally_on:

type	*OPname*	(capitalization indicates minimal abbreviation)

Examples:
```
syntax ..., ... replace ...
syntax ..., ... REPLACE ...
syntax ..., ... detail ...
syntax ..., ... Detail ...
syntax ..., ... CONStant ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name. Thus option `replace` is returned in local macro `‘replace’`; option `detail`, in local macro `‘detail’`; and option `constant`, in local macro `‘constant’`.

The macro contains nothing if not specified, or else it contains the macro's name, fully spelled out.

Warning: Be careful if the first two letters of the option's name are `no`, such as the option called `notice`. You must capitalize at least the `N` in such cases.

option_descriptor optionally_off:

type	`no`	
type	*OPname*	(capitalization indicates minimal abbreviation)

Examples:
```
syntax ..., ... noreplace ...
syntax ..., ... noREPLACE ...
syntax ..., ... nodetail ...
syntax ..., ... noDetail ...
syntax ..., ... noCONStant ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name, excluding the `no`. Thus option `noreplace` is returned in local macro `‘replace’`; option `nodetail`, in local macro `‘detail’`; and option `noconstant`, in local macro `‘constant’`.

The macro contains nothing if not specified, or else it contains the macro's name, fully spelled out, with a `no` prefixed. That is, in the `noREPLACE` example above, macro `‘replace’` contains nothing, or it contains `noreplace`.

option_descriptor optional_integer_value:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(integer`	
type	*#* (unless the option is required)	(the default integer value)
type	`)`	

Examples:
```
syntax ..., ... Count(integer 3) ...
syntax ..., ... SEQuence(integer 1) ...
syntax ..., ... dof(integer -1) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the integer specified by the user, or else it contains the default value.

option_descriptor optional_real_value:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(real`	
type	*#* (unless the option is required)	(the default value)
type	`)`	

Examples:
```
syntax ..., ... Mean(real 2.5) ...
syntax ..., ... SD(real -1) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the real number specified by the user, or else it contains the default value.

option_descriptor optional_confidence_interval:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(cilevel)`	

Example: `syntax ..., ... Level(cilevel) ...`

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

If the user specifies a valid level for a confidence interval, the macro contains that value; see [R] **level**. If the user specifies an invalid level, an error message is issued, and the return code is 198.

If the user does not type this option, the macro contains the default level obtained from `c(level)`.

---

option_descriptor optional_credible_interval:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(crlevel)`	

Example: `syntax ..., ... CLEVel(crlevel) ...`

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

If the user specifies a valid level for a credible interval, the macro contains that value; see [BAYES] **set clevel**. If the user specifies an invalid level, an error message is issued, and the return code is 198.

If the user does not type this option, the macro contains the default level obtained from `c(clevel)`.

---

option_descriptor optional_numlist:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(numlist`	
type	`ascending` or `descending` or *nothing*	
optionally type	`integer`	
optionally type	`missingokay`	
optionally type	`min=#`	
optionally type	`max=#`	
optionally type	`>#` or `>=#` or *nothing*	
optionally type	`<#` or `<=#` or *nothing*	
optionally type	`sort`	
type	`)`	

Examples:
```
syntax ..., ... VALues(numlist) ...
syntax ..., ... VALues(numlist max=10 sort) ...
syntax ..., ... TIME(numlist >0) ...
syntax ..., ... FREQuency(numlist >0 integer) ...
syntax ..., ... OCCur(numlist missingokay >=0 <1e+9) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the values specified by the user, but listed out, one after the other. For instance, the user might specify `time(1(1)4,10)` so that the local macro `‘time’` would contain "`1 2 3 4 10`".

`min` and `max` specify the minimum and maximum number of elements that may be in the list.

`<`, `<=`, `>`, and `>=` specify the range of elements allowed in the list.

`integer` indicates that the user may specify integer values only.

`missingokay` indicates that the user may specify missing values as list elements.

`ascending` specifies that the user must give the list in ascending order without repeated values. `descending` specifies that the user must give the list in descending order without repeated values.

`sort` specifies that the list be sorted before being returned. Distinguish this from modifier `ascending`, which states that the user must type the list in ascending order. `sort` says that the user may type the list in any order but it is to be returned in ascending order. `ascending` states that the list may have no repeated elements. `sort` places no such restriction on the list.

option_descriptor optional_varlist:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(varlist or (varname`	
optionally type	`numeric or string`	
optionally type	`min=#`	
optionally type	`max=#`	
optionally type	`fv`	
optionally type	`ts`	
type	`)`	

Examples:

```
syntax ..., ... ROW(varname) ...
syntax ..., ... BY(varlist) ...
syntax ..., ... Counts(varname numeric) ...
syntax ..., ... TItlevar(varname string) ...
syntax ..., ... Sizes(varlist numeric min=2 max=10) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the names specified by the user, listed one after the other.

`min` indicates the minimum number of variables to be specified if the option is given. `min=1` is the default.

`max` indicates the maximum number of variables that may be specified if the option is given. `max=800` is the default for `varlist` (you may set it to be larger), and `max=1` is the default for `varname`.

`numeric` specifies that the variable list must consist entirely of numeric variables. `string` specifies that the variable list must consist entirely of string variables, meaning `str#` or `strL`. `str#` and `strL` specify that the variable list must consist entirely of `str#` or `strL` variables, respectively.

`fv` specifies that the variable list may contain factor variables.

`ts` specifies that the variable list may contain time-series operators.

option_descriptor optional_namelist:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(namelist or (name`	
optionally type	`min=#`	
optionally type	`max=#`	
optionally type	`local`	
type	`)`	

Examples:

```
syntax ..., ... GENerate(name) ...
syntax ..., ... MATrix(name) ...
syntax ..., ... REsults(namelist min=2 max=10) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the variables specified by the user, listed one after the other.

Do not confuse `namelist` with `varlist`. `varlist` is the appropriate way to specify an option that is to receive the names of existing variables. `namelist` is the appropriate way to collect names of other things—such as matrices—and `namelist` is sometimes used to obtain the name of a new variable to be created. It is then your responsibility to verify that the name specified does not already exist as a Stata variable.

`min` indicates the minimum number of names to be specified if the option is given. `min=1` is the default.

`max` indicates the maximum number of names that may be specified if the option is given. The default is `max=1` for `name`. For `namelist`, the default is the maximum number of variables allowed in Stata.

`local` specifies that the names the user specifies are to satisfy the naming convention for local macro names.

option_descriptor optional_string:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(`<u>`string`</u>	
optionally type	`asis`	
type	`)`	

Examples:
```
syntax ..., ... Title(string) ...
syntax ..., ... XTRAvars(string) ...
syntax ..., ... SAVing(string asis) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the string specified by the user, or else it contains nothing.

`asis` specifies that the option's arguments be returned just as the user typed them, with quotes (if specified) and with any leading and trailing blanks. `asis` should be specified if the option's arguments might contain suboptions or expressions that contain quoted strings. If you specify `asis`, be sure to use compound double quotes when referring to the macro.

option_descriptor optional_passthru:

type	*OPname*	(capitalization indicates minimal abbreviation)
type	`(passthru)`	

Examples:
```
syntax ..., ... Title(passthru) ...
syntax ..., ... SAVing(passthru) ...
```

The result of the option is returned in a macro name formed by the first 31 letters of the option's name.

The macro contains the full option—unabbreviated option name, parentheses, and argument—as specified by the user, or else it contains nothing. For instance, if the user typed `ti("My Title")`, the macro would contain `title("My Title")`.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Introduction

Stata is programmable, making it possible to implement new commands. This is done with the `program` definition statement:

```
program newcmd
        ...
end
```

The first duty of the program is to parse the arguments that it receives.

Programmers use positional argument passing for subroutines and for some new commands with exceedingly simple syntax. It is so easy to program. If program `myprog` is to receive a variable name (call it `varname`) and two numeric arguments (call them `dof` and `beta`), all they need to code is

```
program myprog
        args varname dof beta
        (the rest of the program would be coded in terms of `'varname'', `'dof'', and `'beta'')
        ...
end
```

The disadvantage of this is from the caller's side, because problems would occur if the caller got the arguments in the wrong order or did not spell out the variable name, etc.

The alternative is to use standard Stata syntax. `syntax` makes it easy to make new command `myprog` have the syntax

`myprog` *varname* [`, dof(#) beta(#)`]

and even to have defaults for `dof()` and `beta()`:

```
program myprog
        syntax varlist(max=1) [, Dof(integer 50) Beta(real 1.0)]
        (the rest of the program would be coded in terms of `varlist', `dof', and `beta')
        ...
end
```

The args command

`args` splits what the user typed into words and places the first word in the first macro specified; the second, in the second macro specified; and so on:

```
program myprog
        args arg1 arg2 arg3 ...
        do computations using local macros `arg1', `arg2', `arg3', ...
end
```

`args` never produces an error. If the user specified more arguments than the macros specified, the extra arguments are ignored. If the user specified fewer arguments, the extra macros are set to contain "".

A better version of this program would read

```
program myprog
        version 14.1                    ← new
        args arg1 arg2 arg3 ...
        do computations using local macros `arg1', `arg2', `arg3', ...
end
```

Placing `version 14.1` as the first line of the program ensures that the command will continue to work with future versions of Stata; see **[U] 16.1.1 Version** and **[P] version**. We will include the `version` line from now on.

▷ Example 1

The following command displays the three arguments it receives:

```
. program argdisp
  1.         version 14.1
  2.         args first second third
  3.         display "1st argument = `first'"
  4.         display "2nd argument = `second'"
  5.         display "3rd argument = `third'"
  6. end
. argdisp cat dog mouse
1st argument = cat
2nd argument = dog
3rd argument = mouse
. argdisp 3.456 2+5-12 X*3+cat
1st argument = 3.456
2nd argument = 2+5-12
3rd argument = X*3+cat
```

Arguments are defined by the spaces that separate them. "`X*3+cat`" is one argument, but if we had typed "`X*3 + cat`", that would have been three arguments.

If the user specifies fewer arguments than expected by `args`, the additional local macros are set as empty. By the same token, if the user specifies too many, they are ignored:

```
. argdisp cat dog
1st argument = cat
2nd argument = dog
3rd argument =
. argdisp cat dog mouse cow
1st argument = cat
2nd argument = dog
3rd argument = mouse
```

◁

❑ Technical note

When a program is invoked, exactly what the user typed is stored in the macro `‘0’`. Also the first word of that is stored in `‘1’`; the second, in `‘2’`; and so on. `args` merely copies the `‘1’`, `‘2’`, ... macros. Coding

```
args arg1 arg2 arg3
```

is no different from coding

```
local arg1 ‘"‘1’"’
local arg2 ‘"‘2’"’
local arg3 ‘"‘3’"’
```

❑

The syntax command

`syntax` is easy to use. `syntax` parses standard Stata syntax, which is

command varlist `if` *exp* `in` *range* [*weight*] `using` *filename*, *options*

Actually, standard syntax is a little more complicated than that because you can substitute other things for *varlist*. In any case, the basic idea is that you code a `syntax` command describing which parts of standard Stata syntax you expect to see. For instance, you might code

```
syntax varlist if in, title(string) adjust(real 1)
```

or

```
syntax [varlist] [if] [in] [, title(string) adjust(real 1)]
```

In the first example, you are saying that everything is required. In the second, everything is optional. You can make some elements required and others optional:

```
syntax varlist [if] [in], adjust(real) [title(string)]
```

or

```
syntax varlist [if] [in] [, adjust(real 1) title(string)]
```

or many other possibilities. Square brackets denote that something is optional. Put them around what you wish.

You code what you expect the user to type. `syntax` then compares that with what the user actually did type, and, if there is a mismatch, `syntax` issues an error message. Otherwise, `syntax` processes what the user typed and stores the pieces, split into categories, in macros. These macros are named the same as the syntactical piece:

The varlist specified	will go into	`'varlist'`
The `if` *exp*	will go into	`'if'`
The `in` *range*	will go into	`'in'`
The `adjust()` option's contents	will go into	`'adjust'`
The `title()` option's contents	will go into	`'title'`

Go back to the section *Syntax, continued*; where each element is stored is explicitly stated. When a piece is not specified by the user, the corresponding macro is cleared.

▷ Example 2

The following program simply displays the pieces:

```
. program myprog
  1. version 14.1
  2. syntax varlist [if] [in] [, adjust(real 1) title(string)]
  3. display "varlist contains |`varlist'|"
  4. display "     if contains |`if'|"
  5. display "     in contains |`in'|"
  6. display " adjust contains |`adjust'|"
  7. display "  title contains |`title'|"
  8. end
. myprog
varlist required
r(100);
```

Well, that should not surprise us; we said that the varlist was required in the `syntax` command, so when we tried `myprog` without explicitly specifying a varlist, Stata complained.

```
. myprog mpg weight
varlist contains |mpg weight|
     if contains ||
     in contains ||
 adjust contains |1|
  title contains ||
. myprog mpg weight if foreign
varlist contains |mpg weight|
     if contains |if foreign|
     in contains ||
 adjust contains |1|
  title contains ||
. myprog mpg weight in 1/20
varlist contains |mpg weight|
     if contains ||
     in contains |in 1/20|
 adjust contains |1|
  title contains ||
. myprog mpg weight in 1/20 if foreign
varlist contains |mpg weight|
     if contains |if foreign|
     in contains |in 1/20|
 adjust contains |1|
  title contains ||
```

```
. myprog mpg weight in 1/20 if foreign, title("My Results")
varlist contains |mpg weight|
     if contains |if foreign|
     in contains |in 1/20|
 adjust contains |1|
  title contains |My Results|
. myprog mpg weight in 1/20 if foreign, title("My Results") adjust(2.5)
varlist contains |mpg weight|
     if contains |if foreign|
     in contains |in 1/20|
 adjust contains |2.5|
  title contains |My Results|
```

That is all there is to it.

◁

▷ Example 3

After completing the last example, it would not be difficult to actually make `myprog` do something. For lack of a better example, we will change `myprog` to display the mean of each variable, with said mean multiplied by `adjust()`:

```
program myprog
        version 14.1
        syntax varlist [if] [in] [, adjust(real 1) title(string)]
        display
        if "`title'" != "" {
                display "`title':"
        }
        foreach var of local varlist {
                quietly summarize `var' `if' `in'
                display %9s "`var'" "  " %9.0g r(mean)*`adjust'
        }
end

. myprog mpg weight
      mpg    21.2973
   weight   3019.459
. myprog mpg weight if foreign==1
      mpg   24.77273
   weight   2315.909
. myprog mpg weight if foreign==1, title("My title")
My title:
      mpg   24.77273
   weight   2315.909
. myprog mpg weight if foreign==1, title("My title") adjust(2)
My title:
      mpg   49.54545
   weight   4631.818
```

◁

❑ Technical note

`myprog` is hardly deserving of any further work, given what little it does, but let's illustrate two ideas that use it.

First, we will learn about the `marksample` command; see [P] **mark**. A common mistake is to use one sample in one part of the program and a different sample in another part. The solution is to create at the outset a variable that contains 1 if the observation is to be used and 0 otherwise. `marksample` will do this correctly because `marksample` knows what `syntax` has just parsed:

```
program myprog
        version 14.1
        syntax varlist [if] [in] [, adjust(real 1) title(string)]
        marksample touse                                    ← new
        display
        if "‘title’" != "" {
                display "‘title’:"
        }
        foreach var of local varlist {
                quietly summarize ‘var’ if ‘touse’          ← changed
                display %9s "‘var’" "  " %9.0g r(mean)*‘adjust’
        }
end
```

Second, we will modify our program so that what is done with each variable is done by a subroutine. Pretend here that we are doing something more involved than calculating and displaying a mean.

We want to make this modification to show you the proper use of the `args` command. Passing arguments by position to subroutines is convenient, and there is no chance of error due to arguments being out of order (assuming that we wrote our program properly):

```
program myprog
        version 14.1
        syntax varlist [if] [in] [, adjust(real 1) title(string)]
        marksample touse
        display
        if "‘title’" != "" {
                display "‘title’:"
        }
        foreach var of local varlist {
                doavar ‘touse’ ‘var’ ‘adjust’
        }
end

program doavar
        version 14.1
        args touse name value
        qui summarize ‘name’ if ‘touse’
        display %9s "‘name’" "  " %9.0g r(mean)*‘value’
end
```

❑

Also see

[P] **gettoken** — Low-level parsing

[P] **mark** — Mark observations for inclusion

[P] **numlist** — Parse numeric lists

[P] **program** — Define and manipulate programs

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

[P] **unab** — Unabbreviate variable list

[TS] **tsrevar** — Time-series operator programming command

[U] 11 Language syntax

[U] 16.1.1 Version

[U] 18 Programming Stata

[U] 18.3.1 Local macros

[U] 18.3.5 Double quotes