Title

stata.com

syntax — Mata language grammar and syntax

Description Syntax Remarks and examples Reference Also see

Description

Mata is a C-like compiled-into-pseudocode language with matrix extensions and run-time linking.

Syntax

The basic language syntax is

istmt

where

istmt :=
- *stmt*
- `function` *name*`(`*farglist*`)` *fstmt*
- *ftype name*`(`*farglist*`)` *fstmt*
- *ftype* `function` *name*`(`*farglist*`)` *fstmt*

stmt :=
- *nothing*
- `;` (meaning *nothing*)
- `version` *number*
- `{` *stmt* ... `}`
- *exp*
- `pragma` *pstmt*
- `if (`*exp*`)` *stmt*
- `if (`*exp*`)` *stmt* `else` *stmt*
- `for (`*exp*`;`*exp*`;`*exp*`)` *stmt*
- `while (`*exp*`)` *stmt*
- `do` *stmt* `while (`*exp*`)`
- `break`
- `continue`
- *label*`:`
- `goto` *label*
- `return`
- `return(`*exp*`)`

fstmt :=
- *stmt*
- *type arglist*
- `external` *type arglist*

arglist :=
- *name*
- *name*`()`
- *name*`,` *arglist*
- *name*`(),` *arglist*

farglist :=
- *nothing*
- *efarglist*

efarglist :=
felement
felement, *efarglist*
| *felement*
| *felement*, *efarglist*

felement :=
name
type name
name()
type name()

ftype :=
type
`void`

type :=
eltype
orgtype
eltype orgtype

eltype :=
`transmorphic`
`string`
`numeric`
`real`
`complex`
`pointer`
`pointer(`*ptrtype*`)`

orgtype :=
`matrix`
`vector`
`rowvector`
`colvector`
`scalar`

ptrtype :=
nothing
type
type `function`
`function`

pstmt :=
`unset` *name*
`unused` *name*

name :=
identifier up to 32 characters long

label :=
identifier up to 8 characters long

exp :=
expression as defined in [M-2] **exp**

Remarks and examples

stata.com

Remarks are presented under the following headings:

After reading [M-2] **syntax**, see [M-2] **intro** for a list of entries that give more explanation of what is discussed here.

Treatment of semicolons

Semicolon (;) is treated as a line separator. It is not required, but it may be used to place two statements on the same physical line:

```
x = 1 ; y = 2 ;
```

The last semicolon in the above example is unnecessary but allowed.

Single statements may continue onto more than one line if the continuation is obvious. Take "obvious" to mean that there is a hanging open parenthesis or a hanging dyadic operator; for example,

```
x = (
       3)
x = x +
         2
```

See [M-2] **semicolons** for more information.

Types and declarations

The *type* of a variable or function is described by

eltype *orgtype*

where *eltype* and *orgtype* are each one of

eltype	*orgtype*
transmorphic	matrix
numeric	vector
real	rowvector
complex	colvector
string	scalar
pointer	

For example, a variable might be `real scalar`, or `complex matrix`, or `string vector`.

Mata also has structures—the *eltype* is `struct` *name*—but these are not discussed here. For a discussion of structures, see [M-2] **struct**.

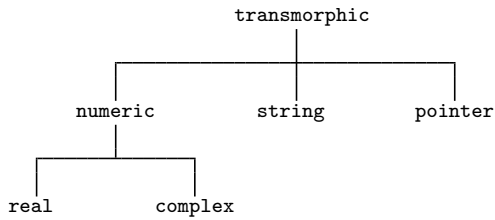
Mata also has classes—the *eltype* is `class` *name*—but these are not discussed here. For a discussion of classes, see [M-2] **class**.

Declarations are optional. When the *type* of a variable or function is not declared, it is assumed to be a `transmorphic matrix`. In particular:

1. *eltype* specifies the type of the elements. When *eltype* is not specified, `transmorphic` is assumed.
2. *orgtype* specifies the organization of the elements. When *orgtype* is not specified, `matrix` is assumed.

All *types* are special cases of `transmorphic matrix`.

The nesting of *eltypes* is



orgtypes amount to nothing more than a constraint on the number of rows and columns of a matrix:

orgtype	Constraint
`matrix`	$r \geq 0$ and $c \geq 0$
`vector`	$r = 1$ and $c \geq 0$ or $r \geq 0$ and $c = 1$
`rowvector`	$r = 1$ and $c \geq 0$
`colvector`	$r \geq 0$ and $c = 1$
`scalar`	$r = 1$ and $c = 1$

See [M-2] **declarations**.

Void matrices

A matrix (vector, row vector, or column vector) that is 0×0, $r \times 0$, or $0 \times c$ is said to be void; see [M-2] **void**.

The function `J(`*r*`,` *c*`,` *val*`)` returns an $r \times c$ matrix with each element containing *val*; see [M-5] **J()**.

`J()` can be used to create void matrices.

See [M-2] **void**.

Void functions

Rather than *eltype orgtype*, a function can be declared to return nothing by being declared to return `void`:

```
void function example(matrix A)
{
     real scalar i

     for (i=1; i<=rows(A); i++) A[i,i] = 1
}
```

A function that returns nothing (does not include a `return(`*exp*`)` statement), in fact returns `J(0, 0, .)`, and the above function could equally well be coded as

```
void function example(matrix A)
{
     real scalar   i
     for (i=1; i<=rows(A); i++) A[i,i] = 1
     return(J(0, 0, .))
}
```

or

```
void function example(matrix A)
{
     real scalar   i
     for (i=1; i<=rows(A); i++) A[i,i] = 1
     return(J(0,0,.))
}
```

Therefore, `void` also is a special case of `transmorphic matrix` (it is in fact a 0×0 `real matrix`). Since declarations are optional (but recommended both for reasons of style and for reasons of efficiency), the above function could also be coded as

```
function example(A)
{
     for (i=1; i<=rows(A); i++) A[i,i] = 1
}
```

See [M-2] **declarations**.

Operators

Mata provides the usual assortment of operators; see [M-2] **exp**.

The monadic prefix operators are

```
-   !   ++   --   &   *
```

Prefix operators `&` and `*` have to do with pointers; see [M-2] **pointers**.

The monadic postfix operators are

```
'    ++    --
```

Note the inclusion of postfix operator ′ for transposition. Also, for Z complex, Z' returns the conjugate transpose. If you want the transposition without conjugation, see [M-5] **transposeonly( )**.

The dyadic operators are

```
=    ?    \    ::    ,    ..    |    &    ==    >=    <=    <    >
!=    +    -    *    #    ^
```

Also, `&&` and `||` are included as synonyms for `&` and `|`.

The operators `==` and `!=` do not require conformability, nor do they require that the matrices be of the same type. In such cases, the matrices are unequal (`==` is false and `!=` is true). For complex arguments, `<`, `<=`, `>`, and `>=` refer to length of the complex vector. `==` and `!=`, however, refer not to length but to actual components. See [M-2] **op_logical**.

The operators `,` and `\` are the row-join and column-join operators. `(1,2,3)` constructs the row vector (1,2,3). `(1\2\3)` constructs the column vector $(1,2,3)'$. `(1,2\3,4)` constructs the matrix with first row (1,2) and second row (3,4). a`,`b joins two scalars, vectors, or matrices rowwise. a`\`b joins two scalars, vectors, or matrices columnwise. See [M-2] **op_join**.

`..` and `::` refer to the row-to and column-to operators. `1..5` is (1,2,3,4,5). `1::5` is (1\2\3\4\5). `5..1` is (5,4,3,2,1). `5::1` is (5\4\3\2\1). See [M-2] **op_range**.

For `|`, `&`, `==`, `>=`, `<=`, `<`, `>`, `!=`, `+`, `-`, `*`, `/`, and `^`, there is *:op* at precedence just below *op*. These operators perform the elementwise operation. For instance, A`*`B refers to matrix multiplication; A`:*`B refers to elementwise multiplication. Moreover, elementwise is generalized to cases where A and B do not have the same number of rows and the same number of columns. For instance, if A is a $1 \times c$ row vector and B is a $r \times c$ matrix, then $||C_{ij}|| = ||A_j|| * ||B_{ij}||$ is returned. See [M-2] **op_colon**.

Subscripts

A`[`i`,`j`]` returns the i,j element of A.

A`[`k`]` returns A`[1,`k`]` if A is $1 \times c$ and A`[`k`,1]` if A is $r \times 1$. That is, in addition to declared vectors, any $1 \times c$ matrix or $r \times 1$ matrix may be subscripted by one index. Similarly, any vector can be subscripted by two indices.

i, j, and k may be vectors as well as scalars. For instance, A`[(3\4\5), 4]` returns a 3×1 column vector containing rows 3 to 5 of the 4th column.

i, j, and k may be missing value. A`[., 4]` returns a column vector of the 4th column of A.

The above subscripts are called list-style subscripts. Mata provides a second format called range-style subscripts that is especially useful for selecting submatrices. A`[|3,3\5,5|]` returns the 3×3 submatrix of A starting at A`[3,3]`.

See [M-2] **subscripts**.

Implied input tokens

Before interpreting and compiling a line, Mata makes the following substitutions to what it sees:

Input sequence	Interpretation
′*name*	′**name*
`[,`	`[.,`
`,]`	`,.]`

Hence, coding `X′Z` is equivalent to coding `X′*Z`, and coding `x = z[1,]` is equivalent to coding `x = z[1,.]`.

Function argument-passing convention

Arguments are passed to functions by address, also known as by name or by reference. They are not passed by value. When you code

```
... f(A) ...
```

it is the address of `A` that is passed to `f()`, not a copy of the values in `A`. `f()` can modify `A`.

Most functions do not modify their arguments, but some do. `lud(`*A*`, `*L*`, `*U*`, `*p*`)`, for instance, calculates the LU decomposition of *A*. The function replaces the contents of *L*, *U*, and *p* with matrices such that $L[p,]*U = A$.

Oldtimers will have heard of the FORTRAN programmer who called a subroutine and passed to it a second argument of 1. Unbeknownst to him, the subroutine changed its second argument, with the result that the constant 1 was changed throughout the rest of his code. That cannot happen in Mata. When an expression is passed as an argument (and constants are expressions), a temporary variable containing the evaluation is passed to the function. Modifications to the temporary variable are irrelevant because the temporary variable is discarded once the function returns. Thus if `f()` modifies its second argument and you call it by coding `f(A,2)`, because 2 is copied to a temporary variable, the value of the literal 2 will remain unchanged on the next call.

If you call a function with an expression that includes the assignment operator, it is the left-hand side of the expression that is passed. That is, coding

```
f(a, b=c)
```

has the same result as coding

```
b = c
f(a, b)
```

If function `f()` changes its second argument, it will be `b` and not `c` that is modified.

Also, Mata attempts not to create unnecessary copies of matrices. For instance, consider

```
function changearg(x) x[1,1] = 1
```

`changearg(mymat)` changes the 1,1 element of `mymat` to 1. Now let us define

```
function cp(x) return(x)
```

Coding changearg(cp(mymat)) would still change mymat because cp() returned x itself. On the other hand, if we defined cp() as

```
function cp(x)
{
     matrix t

     t = x
     return(t)
}
```

then coding changearg(cp(mymat)) would not change mymat. It would change a temporary matrix which would be discarded once changearg() returned.

Passing functions to functions

One function may receive another function as an argument using pointers. One codes

```
function myfunc(pointer(function) f, a, b)
{
     ... (*f)(a) ... (*f)(b) ...
}
```

although the pointer(function) declaration, like all declarations, is optional. To call myfunc() and tell it to use function prima() for f(), and 2 and 3 for a and b, one codes

```
myfunc(&prima(), 2, 3)
```

See [M-2] **ftof** and [M-2] **pointers**.

Optional arguments

Functions may be coded to allow receiving a variable number of arguments. This is done by placing a vertical or bar (|) in front of the first argument that is optional. For instance,

```
function mynorm(matrix A, |scalar power)
{
     ...
}
```

The above function may be called with one matrix or with a matrix followed by a scalar.

The function args() (see [M-5] **args( )**) can be used to determine the number of arguments received and to set defaults:

```
function mynorm(matrix A, |scalar power)
{
     ...
      (args()==1) power = 2
     ...
}
```

See [M-2] **optargs**.

Reference

Gould, W. W. 2005. Mata Matters: Translating Fortran. *Stata Journal* 5: 421–441.

Also see

[M-2] **intro** — Language definition