Description Syntax Remarks and examples Reference Also see

Description

exp is used in syntax diagrams to mean "any valid expression may appear here". Expressions can range from being simple constants

```
2
"this"
3+2i
```

to being names of variables

```
A
beta
varwithverylongname
```

to being a full-fledged scalar, string, or matrix expression:

```
sqrt(2)/2
substr(userinput, 15, strlen(otherstr))
conj(X)'X
```

Syntax

exp

Remarks and examples

Remarks are presented under the following headings:

What's an expression

Everybody knows what an expression is: expressions are things like `2+3` and `invsym(X'X)*X'y`. Simpler things are also expressions, such as numeric constants

`2` is an expression

and string literals

`"hi there"` is an expression

and function calls:

```
sqrt(2)          is an expression
```

Even when functions do not return anything (the function is void), the code that causes the function to run is an expression. For instance, the function `swap()` (see [M-5] **swap( )**) interchanges the contents of its arguments and returns nothing. Even so,

```
swap(A, B)       is an expression
```

Assignment suppresses display, as does (void)

The equal sign assigns the result of an expression to a variable. For instance,

```
a = 2 + 3
```

assigns 5 to `a`. When the result of an expression is not assigned to a variable, the result is displayed at the terminal. This is true of expressions entered interactively and of expressions coded in programs. For instance, given the program

```
function example(a, b)
{
     "the answer is"
     a+b
}
```

executing `example()` produces

```
: example(2, 3)
  the answer is
  5
```

The fact that 5 appeared is easy enough to understand; we coded the expression `a+b` without assigning it to another variable. The fact that "the answer is" also appeared may surprise you. Nevertheless, we coded `"the answer is"` in our program, and that is an example of an expression, and because we did not assign the expression to a variable, it was displayed.

In programming situations, there will be times when you want to execute a function—call it `setup()`—but do not care what the function returns, even though the function itself is not void (that is, it returns something). If you code

```
function example(...)
{
     ...
     setup(...)
     ...
}
```

the result will be to display what `setup()` returns. You have two alternatives. You could assign the result of `setup` to a variable even though you will subsequently not use the variable

```
function example(...)
{
     ...
     result = setup(...)
     ...
}
```

or you could cast the result of the function to be void:

```
function example(...)
{
     ...
     (void) setup(...)
     ...
}
```

Placing `(void)` in front of an expression prevents the result from being displayed.

The pieces of an expression

Expressions comprise

- numeric literals
- string literals
- variable names
- operators
- functions

Numeric literals

Numeric literals are just numbers

```
2
3.14159
-7.2
5i
1.213e+32
1.213E+32
1.921fb54442d18X+001
1.921fb54442d18x+001
.
.a
.b
```

but you can suffix an `i` onto the end to mean imaginary, such as `5i` above. To create complex numbers, you combine real and imaginary numbers using the `+` operator, as in `2+5i`. In any case, you can put the `i` on the end of any literal, so `1.213e+32i` is valid, as is `1.921fb54442d18X+001i`.

`1.921fb54442d18X+001i` is a formidable-looking beast, with or without the `i`. `1.921fb54442d18X+001` is a way of writing floating-point numbers in binary; it is described in **[U] 12.5 Formats: Controlling how data are displayed**. Most people never use it.

Also, numeric literals include Stata's missing values, `.`, `.a`, `.b`, ..., `.z`.

Complex variables may contain missing just as real variables may, but they get only one: `.a+.bi` is not allowed. A complex variable contains a valid complex value, or it contains `.`, `.a`, `.b`, ..., `.z`.

String literals

String literals are enclosed in double quotes or in compound double quotes:

```
"the answer is"
"a string"
‘"also a string"’
‘"The "factor" of a matrix"’
""
‘""’
```

Strings in Mata contain between 0 and 2,147,483,647 bytes. `""` or `‘""’` is how one writes the 0-length string.

Any plain ASCII or UTF-8 character may appear in the string, but no provision is provided for typing unprintable characters into the string literal. Instead, you use the `char()` function; see [M-5] **ascii( )**. For instance, `char(13)` is carriage return, so the expression

```
"my string" + char(13)
```

produces “my string” followed by a carriage return.

No character is given a special interpretation. In particular, backslash (\) is given no special meaning by Mata. The string literal `"my string\n"` is just that: the characters “my string” followed by a backslash followed by an “n”. Some functions, such as `printf()` (see [M-5] **printf( )**), give a special meaning to the two-character sequence `\n`, but that special interpretation is a property of the function, not Mata, and is noted in the function’s documentation.

Strings are not zero (null) terminated in Mata. Mata knows that the string `"hello"` is of length 5, but it does not achieve that knowledge by padding a binary 0 as the string’s fifth character. Thus strings may be used to hold binary information.

Although Mata gives no special interpretation to binary 0, some Mata functions do. For instance, `strmatch(`*s*`,` *pattern*`)` returns 1 if *s* matches *pattern* and 0 otherwise; see [M-5] **strmatch( )**. For this function, both strings are considered to end at the point they contain a binary 0, if they contain a binary 0. Most strings do not, and then the function considers the entire string. In any case, if there is special treatment of binary 0, that is on a function-by-function basis, and a note of that is made in the function’s documentation.

Some string functions in Mata have variants that are designed specifically to deal with Unicode. For examples, `usubstr()` is the Unicode-aware version of `substr()`. See **[U] 12.4.2 Handling Unicode strings** for more details on working with Unicode strings.

Variable names

Variable names are just that. Names are case sensitive and no abbreviations are allowed:

```
X
x
MyVar
VeryLongVariableNameForUseInMata
MyVariable
```

The maximum length of a variable name is 32 characters.

Operators

Operators, listed by precedence, low to high

Operator	Operator name	Documentation
a = *b*	assignment	[M-2] **op_assignment**
a ? *b* : *c*	conditional	[M-2] **op_conditional**
a \ *b*	column join	[M-2] **op_join**
a :: *b*	column to	[M-2] **op_range**
a , *b*	row join	[M-2] **op_join**
a .. *b*	row to	[M-2] **op_range**
a :| *b*	elementwise or	[M-2] **op_colon**
a | *b*	or	[M-2] **op_logical**
a :& *b*	elementwise and	[M-2] **op_colon**
a & *b*	and	[M-2] **op_logical**
a :== *b*	elementwise equal	[M-2] **op_colon**
a == *b*	equal	[M-2] **op_logical**
a :>= *b*	elementwise greater than or equal	[M-2] **op_colon**
a >= *b*	greater than or equal	[M-2] **op_logical**
a :<= *b*	elementwise less than or equal	[M-2] **op_colon**
a <= *b*	less than or equal	[M-2] **op_logical**
a :< *b*	elementwise less than	[M-2] **op_colon**
a < *b*	less than	[M-2] **op_logical**
a :> *b*	elementwise greater than	[M-2] **op_colon**
a > *b*	greater than	[M-2] **op_logical**
a :!= *b*	elementwise not equal	[M-2] **op_colon**
a != *b*	not equal	[M-2] **op_logical**
a :+ *b*	elementwise addition	[M-2] **op_colon**
a + *b*	addition	[M-2] **op_arith**
a :- *b*	elementwise subtraction	[M-2] **op_colon**
a - *b*	subtraction	[M-2] **op_arith**
a :* *b*	elementwise multiplication	[M-2] **op_colon**
a * *b*	multiplication	[M-2] **op_arith**
a # *b*	Kronecker	[M-2] **op_kronecker**
a :/ *b*	elementwise division	[M-2] **op_colon**
a / *b*	division	[M-2] **op_arith**
-*a*	negation	[M-2] **op_arith**
a :^ *b*	elementwise power	[M-2] **op_colon**
a ^ *b*	power	[M-2] **op_arith**
a'	transposition	[M-2] **op_transpose**
**a*	contents of	[M-2] **pointers**
&*a*	address of	[M-2] **pointers**
!*a*	not	[M-2] **op_logical**
a[*exp*]	subscript	[M-2] **Subscripts**
a[|*exp*|]	range subscript	[M-2] **Subscripts**
a++	increment	[M-2] **op_increment**
a--	decrement	[M-2] **op_increment**
++*a*	increment	[M-2] **op_increment**
--*a*	decrement	[M-2] **op_increment**

Functions

Functions supplied with Mata are documented in [M-5]. An index to the functions can be found in [M-4] **Intro**.

Reference

Gould, W. W. 2006. Mata Matters: Precision. *Stata Journal* 6: 550–560.

Also see

[M-2] **Intro** — Language definition