Title

op_assignment — Assignment operator

Syntax

lval = *exp*

where *exp* is any valid expression and where *lval* is

name
name[*exp*]
name[*exp*, *exp*]
name[|*exp*|]

In pointer use (advanced), *name* may be

**lval*
*(*lval*)
*(*lval*[*exp*])
*(*lval*[*exp*, *exp*])
*(*lval*[|*exp*|])

in addition to being a variable name.

Description

= assigns the evaluation of *exp* to *lval*.

Do not confuse the = assignment operator with the == equality operator. Coding

```
x = y
```

assigns the value of y to x. Coding

```
if (x==y) ...                    (note doubled equal signs)
```

performs the action if the value of *x* is equal to the value of *y*. See [M-2] **op_logical** for a description of the == equality operator.

If the result of an expression is not assigned to a variable, then the result is displayed at the terminal; see [M-2] **exp**.

Remarks and examples

stata.com

Remarks are presented under the following headings:

Assignment suppresses display

When you interactively enter an expression or code an expression in a program without the equal-assignment operator, the result of the expression is displayed at the terminal:

```
: 2 + 3
  5
```

When you assign the expression to a variable, the result is not displayed:

```
: x = 2 + 3
```

The equal-assignment operator

Equals is an operator, so in addition to coding

```
a = 2 + 3
```

you can code

```
a = b = 2 + 3
```

or

```
y = x / (denominator = sqrt(a+b))
```

or even

```
y1 = y2 = x / (denominator = sqrt(sum=a+b))
```

This last is equivalent to

```
sum = a + b
denominator = sqrt(sum)
y2 = x / denominator
y1 = y2
```

Equals binds weakly, so

```
a = b = 2 + 3
```

is interpreted as

```
a = b = (2 + 3)
```

and not

```
a = (b=2) + 3
```

lvals, what appears on the left-hand side

What appears to the left of the equals is called an *lval*, short for left-hand-side value. It would make no sense, for instance, to code

```
sqrt(4) = 3
```

and, as a matter of fact, you are not allowed to code that because `sqrt(4)` is not an *lval*:

```
: sqrt(4) = 3
invalid lval
r(3000);
```

An *lval* is anything that can hold values. A scalar can hold values

```
a = 3
x = sqrt(4)
```

a matrix can hold values

```
A = (1, 2 \ 3, 4)
B = invsym(C)
```

a matrix row can hold values

```
A[1,.] = (7, 8)
```

a matrix column can hold values

```
A[.,2] = (9 \ 10)
```

and finally, a matrix element can hold a value

```
A[1,2] = 7
```

lvals are usually one of the above forms. The other forms have to do with pointer variables, which most programmers never use; they are discussed under *Pointer lvals* below.

Row, column, and element lvals

When you assign to a row, column, or element of a matrix,

```
A[1,.] = (7, 8)
A[.,2] = (9 \ 10)
A[1,2] = 7
```

the row, column, or element must already exist:

```
: A = (1, 2 \ 3, 4)
: A[3,4] = 4
                 <istmt>:  3301  subscript invalid
r(3301);
```

This is usually not an issue because, by the time you are assigning to a row, column, or element, the matrix has already been created, but in the event you need to create it first, use the `J()` function; see [M-5] **J()**. The following code fragment creates a 3 × 4 matrix containing the sum of its indices:

```
A = J(3, 4, .)
for (i=1; i<=3; i++) {
     for (j=1; j<=4; j++) A[i,j] = i + j
}
```

Pointer lvals

In addition to the standard *lvals*

```
A = (1, 2 \ 3, 4)
A[1,.] = (7, 8)
A[.,2] = (9 \ 10)
A[1,2] = 7
```

pointer *lvals* are allowed. For instance,

```
*p = 3
```

stores 3 in the address pointed to by pointer scalar `p`.

```
(*q)[1,2] = 4
```

stores 4 in the (1,2) element of the address pointed to by pointer scalar `q`, whereas

```
*Q[1,2] = 4
```

stores 4 in the address pointed to by the (1,2) element of pointer matrix `Q`.

```
*Q[2,1][1,3] = 5
```

is equivalent to

```
*(Q[2,1])[1,3] = 5
```

and stores 5 in the (1,3) element of the address pointed to by the (2,1) element of pointer matrix `Q`.

Pointers to pointers, pointers to pointers to pointers, etc., are also allowed. For instance,

```
**r = 3
```

stores 3 in the address pointed to by the address pointed to by pointer scalar `r`, whereas

```
*((*(Q[1,2]))[2,1])[3,4] = 7
```

stores 7 in the (3,4) address pointed to by the (2,1) address pointed to by the (1,2) address of pointer matrix `Q`.

Conformability

a = *b*:

input:

b: $r \times c$

output:

a: $r \times c$

Diagnostics

a = *b* aborts with error if there is insufficient memory to store a copy of *b* in *a*.

Also see

[M-5] **swap()** — Interchange contents of variables

[M-2] **exp** — Expressions

[M-2] **intro** — Language definition