Operators

Table of Contents

Operators and Precedence

The following table is essentially a superset of **Table 5.2** in Ousterhout’s 1994 classic Tcl and the Tk Toolkit. As there, groups of operators between horizontal lines have the same precedence; higher groups have higher precedence.

Operators are left-associative unless specified otherwise. For example, ** is right-associative, as shown by:

```
% [nap "10 ** 2 ** 3"]
1e+08
```

The nature of operands is indicated as follows:
a and b represent general arrays.
x, y and z represent scalars.
u and v represent vectors.
A and B represent matrices.
n represents a Tcl name, which may include namespaces.
p represents a boxed vector of pointers to arrays a0, a1, a2,

’AP’ means *arithmetic progression* .

Syntax	Result	height	Right-associative

+a New copy of a -a Negative of a !a Logical NOT: 1 if a is zero, else 0 —a Absolute value of a = abs(a) ˆa Nearest integer to a = i32(nint(a)) <a Largest integer not greater than a = i32(floor(a)) >a Smallest integer not less than a = i32(ceil(a)) a Bit-wise complement of a #a Frequencies of values 0, 1, 2, @a Indirect subscript @@a Indirect subscript
v@b smallest s such that vs=b v@@bi32 s for which —vs-b— is least v@@@b smallest i32 s for which vs=b
u#vu copies of v p#b Cross-product replication
u.v (u and v vectors) Scalar (dot) product A.B (A and B matrices) Matrix product
a*ba χ b a/ba ÷ b a%b Remainder after dividing a by b
a+ba + b a-ba b
a<<b Left-shift a by b bits a>>b Right-shift a by b bits
a<<<b Lesser of a and b a>>>b Greater of a and b
a<b 1 if a < b, else 0 a>b 1 if a > b, else 0 a<=b 1 if a b, else 0 a>=b 1 if a b, else 0
a==b 1 if a = b, else 0 a!=b 1 if a b, else 0
a&b Bit-wise AND of a and b
aˆb Bit-wise exclusive OR of a and b
a—b Bit-wise OR of a and b
a&&b Logical AND: 1 if a 0 and b 0, else 0
a——b Logical OR: 1 if a 0 or b 0, else 0
x..y AP from x to y in steps of +1 or -1 x..y...z AP from x to y in steps of z x...y..z AP from y to z with x elements
a?b:c Choice: if a 0 then b, else c
a//b Concatenate along existing dimension a///b Concatenate along new dimension
[a],[b] Boxed vector pointing to a and b (unless already boxed) If a or b is already boxed then concatenate it
n=a Result is a. Right-associative Side Effect: Set n to OOC-name of a

Assignment Operator “=“

The “nap“ command (unlike “expr“) allows the assignment operator “=“. The left-hand operand must be a Tcl name, which is used to define a Tcl variable whose (string) value is set to the OOC-name of the right-hand operand. The assignment operator has a result like any other operator. This result is the value of the right-hand operand. This is shown in the following:

```
% nap "a = (b = 6) + 2"
::NAP::15-15
% $b
6
% $a
8
```

The assignment operator has the lowest precedence and is right-associative, allowing expressions such as:

```
% nap "a = 3 + b = {1.5 0}"
::NAP::16-16
% $b
1.5 0
% $a
4.5 3
```

Link Operator “,“

The link operator “,“ produces a boxed vector pointing to the operands. A common use of “,“ is to pass multiple arguments to a function. For example the logarithm function log takes an optional second argument specifying base, as in:

```
% [nap "log(32, 2)"]
5
```

The operator “,“ is also used in *cross-product indexing* , as discussed in the section NAP Indexing.

The left-hand operand of “,“ generates one boxed vector and the right-hand operand generates another. These two boxed vectors are concatenated to form the result, which is also a boxed vector. If the data-type of an operand is not boxed then it generates a single-element boxed vector pointing to it. If an operand is a boxed vector then it generates a copy of itself. If an operand is a boxed scalar then it is treated as a boxed vector with a single element. If an operand is absent (NULL) then it generates a single-element (whose value is 0, the missing-value) boxed vector.

Arithmetic Progression Operators “..“ and “...“

The operator “..“ generates an arithmetic progression. If both operands are simple numeric scalars then the step size is +1 or 1, the left-hand operand specifies the first value and the right-hand operand specifies the final value. For example:

```
% [nap "3 .. 6"]
3 4 5 6
% [nap "6 .. 3"]
6 5 4 3
% [nap "1.8 .. -1.2"]
1.8 0.8 -0.2 -1.2
```

If the difference between the operands is not an integral multiple of the step size then the final step is smaller than the preceding steps. This is shown by:

```
% [nap "2.3 .. 5.9"]
2.3 3.3 4.3 5.3 5.9
```

The right-hand operand can be a boxed two-element vector pointing to the final value and the step size. Such a boxed operand is usually generated using the operator “...“, as in:

```
% [nap "3 .. 9 ... 2"]
3 5 7 9
% [nap "0 .. -1.6 ... -0.5"]
0 -0.5 -1 -1.5 -1.6
```

The left-hand operand can be a boxed two-element vector pointing to the number of elements and the first value. Such a boxed operand is also usually generated using the operator "...", as in:

```
% [nap "5 ... 1 .. 7"]
1 2.5 4 5.5 7
```

It is not legal for both operands to be boxed. It is legal to specify a non-integral number of elements, as in:

```
% [nap "3.5 ... 2 .. 12"]
2 6 10 12
```

Note that 3.5 elements means 2.5 steps. There are two full steps of 4, followed by a half step of 2. When the left-hand operand is boxed the step size is calculated using (final first)/(n 1), where n is the number of elements.

The data-type of the result depends on the data-types of first, final and step. For example:

```
% [nap "1 .. 7.0 ... 2"] all
::NAP::262-262  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Value:
1 3 5 7
```

Concatenation Operators "//" and "///"

The following example illustrates the difference between "//" and "///" with vector operands:

```
% [nap "{5 2} // {9 8}"]
5 2 9 8
% [nap "{5 2} /// {9 8}"]
5 2
9 8
```

The following example illustrates the difference between "//" and "///" with matrix operands:

```
% [nap "{{6 2 1}{0 9 4}} // {{7 2 7}{3 3 8}}"] all
```

```
::NAP::29-29  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 4      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4
7 2 7
3 3 8
% [nap "{{6 2 1}{0 9 4}} /// {{7 2 7}{3 3 8}}"] all
::NAP::35-35  i32  MissingValue: -2147483648  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 2   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4

7 2 7
3 3 8
```

Note that "//" concatenates along the most significant existing dimension, whereas "///" concatenates along a new dimension. This new dimension is of size 2 and is more significant that the existing dimensions.

The above examples had operands with identical shapes and data-types. It is obviously desirable to allow the operands of "//" to have different sized leading (most significant) dimensions. NAP does allow this, as shown by:

```
% [nap "'Hello' // ' world.'"]
Hello world.
% [nap "{{6 2 1}{0 9 4}} // {{7 2 7}}"]
6 2 1
0 9 4
7 2 7
```

In fact, both operators allow any combination of shapes. Operands of "///" are reshaped to the same shape. Operands of "//" are reshaped so all dimensions except the leading one have the same size. The following examples illustrate this reshaping process (with data-type conversion when required):

```
% [nap "{{6 2 1}{0 9 4}} // {7 2 7}"]
6 2 1
0 9 4
7 2 7
% [nap "{{6 2 1}{0 9 4}} // 3.0"] all
::NAP::142-142  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4
```

```
3 3 3
% [nap "{{6 2 1}{0 9 4}} /// 3.0"] all
::NAP::148-148  f64  MissingValue: NaN  References: 0  Unit: (NULL)
Dimension 0   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 1   Size: 2      Name: (NULL)    Coordinate-variable: (NULL)
Dimension 2   Size: 3      Name: (NULL)    Coordinate-variable: (NULL)
Value:
6 2 1
0 9 4

3 3 3
3 3 3
```

Inverse Indexing Operators “@“, “@@“ and “@@@“

These three operators all take an optional vector left-hand operand. (The “@“ operator also allows the left-hand operand to have a rank greater than 1.) The result is a subscript of this vector. The left-hand operand defaults to the coordinate-variable of the dimension (only relevant to indirect indexing).

The right-hand operand is attached to the result using its *link slot* . This enables the right-hand operand to be automatically used as a coordinate variable if the result is directly used as an index. Note that the results of operators/functions do not normally retain any links in their operands/arguments, so this only applies to *direct* use. (The right-hand operand would not be an appropriate coordinate variable if there were further arithmetic prior to indexing.) **Interpolated Subscript “@“** The result of v@b is the smallest (possibly fractional) f32 subscript s such that v(s)==b. For example:

```
% [nap "{1.5 3.4 3.6 4} @ 3.5"]
1.5
% [nap "{1.5 3.4 3.6 4} @ 3.7"]
2.25
```

Note that 3.5 is halfway between 3.4 (subscript 1) and 3.6 (subscript 2), so the first result is 1.5. Similarly, 3.7 is quarter-way between 3.6 (subscript 2) and 4 (subscript 3), so the second result is 2.25.

Combining these two examples into one:

```
% [nap "{1.5 3.4 3.6 4} @ {3.5 3.7}"]
1.5 2.25
```

We can check this result by using it as an index:

```
% [nap "{1.5 3.4 3.6 4}({1.5 2.25})"]
3.5 3.7
```

The following example has multiple exact matches. In this case the result is defined as the mean of the matching subscripts.

```
% [nap "{1.3 6.5 6.5 7.1} @ 6.5"]
1.5
```

The following example shows how extrapolation is used to define the result when the right-hand operand is outside the range of the left-hand operand:

```
% [nap "{-1 0 2} @ {-2 5}"]
-1 3.5
```

Such extrapolation can be prevented by adding end points with missing or infinite values, as in:

```
% [nap "{_ -1 0 2 _} @ {-2 -1 2 5}"]
_ 1 3 _
% [nap "{-1i -1 0 2 1i} @ {-2 -1 2 5}"]
1 1 3 3
```

The effect of other missing values is shown by:

```
% [nap "{_ 2 4 _ 6 8 _} @ (1 .. 9)"] value
_ 1 1.5 2 _ 4 4.5 5 _
```

If the left-hand operand is not monotonic (sorted) then the result is defined by the first match, as in:

```
% [nap "{2 4 5 3} @ (1 .. 6)"]
-0.5 0 0.5 1 2 _
```

The left-hand operand can have a rank greater than 1. In this case the search takes place over the most significant dimension (0) of the left-hand operand. The following example searches down each column for the value 0.7.

```
% nap "mat = {
    {0.3 0.1 0.9}
    {0.5 0.5 0.8}
    {0.6 0.1 0.6}
    {0.8 0.0   _}
}"
::NAP::157-157
% [nap "mat @ 0.7"]
2.5 _ 1.5
```

Thus this combines the effect of the following three commands.

```
% [nap "{0.3 0.5 0.6 0.8} @ 0.7"]
2.5
% [nap "{0.1 0.5 0.1 0.0} @ 0.7"]
_
% [nap "{0.9 0.8 0.6 _} @ 0.7"]
1.5
```

The right-hand operand can have any rank, but trailing dimensions (excluding dimension 0 of the left-hand operand) must match. The following example has a right-hand operand with the same number (3) of columns as mat.

```
% [nap "mat @ {{0.7 0.7 0.7}{0.4 0.5 0.8}}"]
2.5   _ 1.5
0.5 1.0 1.0
```

The following 3D array contains ocean temperature data for 4 depths, 2 latitudes and 3 longitudes. Note that some (shallower) points have missing values at the deepest level. For each (latitude, longitude) point, we want to find the depth (subscript) corresponding to a temperature of 10 degrees. The missing value in the result corresponds to an oceanic column whose minimum temperature is 12.

```
% nap "temperature = {
    {{11 12 13}{11 11 12}}
    {{ 9  9 13}{11  8 10}}
    {{ 8 10 12}{ 9  8 10}}
    {{ 6  2  _}{ 5  _  _}}
}"
% [nap "temperature @ 10"]
0.500000 0.666667        _
1.500000 0.333333 1.000000
```

Subscript of Closest "@@"

The result of v@@b is the i32 subscript s for which abs(v(s)-b) is least. For example:

```
% [nap "{1.5 3.4 0 2.4 -1 0} @@ {2 -99}"]
3 4
```

Element 3 has the value 2.4, which is the closest to 2. Element 4 has the value -1, which is the closest to -99.

The following example shows how the right-hand operand becomes the coordinate variable if the result is used directly as an index, but not if there is further arithmetic.

```
% nap "coarse = {4 8 7}"
::NAP::14-14
% nap "time = {2 3 5}"
::NAP::16-16
% $coarse set coo time
% [nap "fine = coarse(time@@(2.4 .. 4.6 ... 0.2))"] value
4 8 8 8 8 8 8 8 8 7 7 7
% [$fine coo] value; # Display coordinate variable
2.4 2.6 2.8 3 3.2 3.4 3.6 3.8 4 4.2 4.4 4.6
% [nap "fine = coarse(time@@(2.4 .. 4.6 ... 0.2)+1)"] value; # Do further arithmetic
8 7 7 7 7 7 7 7 7 4 4 4
% [$fine coo] value; # Display coordinate variable
3 5 5 5 5 5 5 5 5 2 2 2
```

Subscript of Match “@@@“

The result of v@@@b is the smallest i32 subscript s for which v(s)==b. For example:

```
% [nap "{3 2 9 2 0 3} @@@ {0 3 2}"]
4 0 1
```

Element 4 is the only 0, element 0 is the first 3 and element 1 is the first 2.

The following example shows that this operator can be used with character data:

```
% [nap 'hello world' @@@ 'wol']
6 4 2
```

Tally Unary Operator “#“

Unary “#“ produces a frequency table. It tallies the number of 0s, 1s, 2s, , as in the following:

```
% [nap "#{2 5 4 5 2 -3 0 2}"]
1 0 3 0 1 2
```

There is one zero, no ones, three twos, no threes, one four and two fives. Note that the negative value (-3) is ignored.

If the operand has more than 1 dimension then the result has the same shape, except that the size of the first dimension is changed to m+1, where m is the maximum value. Each element of the result is a frequency tallied over the first dimension. For example:

```
% [nap "{{2 5 4 5}{2 -3 0 2}}"]
 2  5  4  5
 2 -3  0  2
```

```
% [nap "#{{2 5 4 5}{2 -3 0 2}}"]
0 0 1 0
0 0 0 0
2 0 0 1
0 0 0 0
0 0 1 0
0 1 0 1
```

If the operand is boxed and points to n arrays (which each have the same number of elements) then the result is the n-dimensional array of joint frequencies. For example:

```
% [nap "#({2 1 1 0 1},{1 1 3 2 1})"]
0 0 1 0
0 2 0 1
0 1 0 0
```

The boxed operand defines the five pairs (2,1), (1,1), (1,3), (0,2) and (1,1). The above result gives the frequencies of these pairs.

Replicate Binary Operator "#"

# can appear within array constants, as in:

```
% [nap "{7 3#8 0}"]
7 8 8 8 0
```

The # operator has a related meaning, as shown by:

```
% [nap "3#8"]
8 8 8
% [nap "{4 1 0 2} # {7 12 9 8}"] value
7 7 7 7 12 8 8
```

Each element of the left-hand operand defines the number of replications of the corresponding element of the right-hand operand. The operands can be vectors or scalars. The result is a vector.

Note that one can use this operator to select from a vector those elements which satisfy some condition. The following example selects the even elements:

```
% nap "x = {9 1 0 2 3 -8 0}"
::NAP::286-286
% [nap "(x % 2 == 0) # x"]
0 2 -8 0
```

This works because the left-hand operand is:

```
% [nap "(x % 2 == 0)"] value
0 0 1 1 0 1 1
```

If the right-hand operand b is multidimensional then the left-hand operand must be a boxed vector pointing to a vector corresponding to each dimension of b. For example:

```
% nap "mat = reshape(1 .. 12, {3 4})"
::NAP::316-316
% $mat
 1  2  3  4
 5  6  7  8
 9 10 11 12
% [nap "({2 0 1},{3 2 0 1}) # mat"]
 1  1  1  2  2  4
 1  1  1  2  2  4
 9  9  9 10 10 12
```

This is equivalent to using the following cross-product index:

```
% [nap "mat({0 0 2},{0 0 0 1 1 3})"]
 1  1  1  2  2  4
 1  1  1  2  2  4
 9  9  9 10 10 12
```

Remainder Operator “%“

The value of the remainder r = a % b is defined for all real a and b so that:
if b > 0 then 0 r < b
if b = 0 then r = 0
if b < 0 then b < r 0
if a 0 and b = then r = a
if a 0 and b = then r = a
if a < 0 and b = then r =
if a > 0 and b = then r = .
Thus:

```
% [nap "0.7 % {0.3 0 -0.3}"]
0.1 0 -0.2
% [nap "{7 0 -7} % 1if32"]
7 0 Inf
% [nap "{7 0 -7} % -1if32"]
-Inf 0 -7
```