Bond m&v u&n _

`m&v y` is defined as `m v y`; that is, the left argument `m` is bonded with the dyad `v` to produce a monadic function.	

For example:

```
   10&^. 2 3 10 100 200
0.30103 0.477121 1 2 2.30103
   base10log=: 10&^.
   base10log 2 3 10 100 200
0.30103 0.477121 1 2 2.30103

   sine=: 1&o.
   sine o. 0 0.25 0.5 1.5 2
0 0.707107 1 _1 0
```

Similarly, `u&n y` is defined as `y u n`; in other words, as the dyad `u` provided with the right argument `n` to produce a monadic function (that is, a function whose dyadic case has an empty domain). For example:

```
   ^&3 (1 2 3 4 5)
1 8 27 64 125

   ^&2 3"0 (1 2 3 4 5)
 1   1
 4   8
 9  27
16  64
25 125
```

Use of the bond conjunction is often called Currying in honor of Haskell Curry.

Compose `u&v mv mv mv`

`u&v y` ↔ `u v y`. Thus `+:&- 7` is `_14` (double the negation). Moreover, the monads `u&v` and `u@v` are equivalent.	`x u&v y` ↔ `(v x) u (v y)`. For example, `3 +&! 4` is `30`, the sum of factorials.

The monadic case is equivalent to the composition used in mathematics, but the dyadic case opens up other possibilities. For example:

```
   3 +&^. 4
2.48491
```
Sum of natural logarithms

```
   ^ 3 +&^. 4
12
```
Multiplication using natural logs

```
   3 +&(10&^.) 4
1.07918
```
Sum of base ten logarithms

```
   10 ^ 3 +&(10&^.) 4
12
```
Multiplication using base ten logs

```
   3 +&.^. 4
12
```
See the related conjunction under (`&.`).

```
   3 +&.(10&^.) 4
12
```

Compare the behavior of `&` with that of `&:` . They differ only in the ranks of the verbs that they produce.