Simple Arrays

Outline

1. Introduction

2. Creating Test Arrays

The primitive functions illustrated in this section are:
Reshape: Dyadic ⍴
Interval: Monadic ⍳
Roll and Deal: Monadic and Dyadic ?
Assignment: Dyadic ←

3. Characteristics of Arrays

The primitive functions illustrated in this section are:
Shape: Monadic ⍴
Indexing: [;⍳;]
Count and Choose: Monadic and Dyadic #
Type: Monadic ⍕

4. Scalar Functions

5. Structural Functions and the Items of Arrays

The primitive functions illustrated in this section are:
Take and Drop: Dyadic ↑ and ↓
Replicate and Expand: Dyadic / and \
Reverse and Rotate:Monadic and Dyadic ⌽
Catenate: Dyadic ,
Laminate: Dyadic ⍪

6. Reduction and Scan

7. Frames and Cells

7a. The Primitive Functions Member Of and Index Of

7b. The Rank Operator

8. Idioms and Phrases

1. Introduction

These tutorials are designed not only to familiarize you with the A+ language, but to help you adopt the attitude that much can be learned about A+ from experimentation. As you work on your programming assignments you will undoubtedly encounter A+ expressions, indeed entire functions, where you have no clue as to how they work. Often it is not necessary to know how they work, since you will only have to use them as is. And often you will be too busy to figure them out at the moment. Even so, you should get in the habit of reserving time to work your way through such functions. What you will learn are new ways to say things succinctly, new ways to say things in computationally efficient ways, and - this is rare, of course - some things not to do. In that spirit, this tutorial begins with a section on creating test data. The test data created here is fairly simple, because we aren't looking at very specific things. In order to understand real applications, it's worthwhile to generate realistic test data, although it may be difficult, and you may learn something new while doing it.

Following the section on test data, there are several illustrating various things about various A+ primitives. The last section presents a series of basic A+ phrases, some sample test data, and some

exercises to work through as you figure out what the expressions do and how they do it. Feel free to create your own test data and explore the boundaries - shape, type, and values - of where the expressions work and where they don't work.

With one exception, the A+ primitives discussed in this chapter correspond to APL\360, which is the common part of all commercial APL systems. Thus, if you have an APL background, this chapter will show you where A+ differs from what you already know about APL\360, although most times the comparisons are left to you to make. If you do not have an APL background, the material in this chapter is a good place to start.

The exception referred above is the rank operator, which replaces the axis operator in APL\360, but is much more general in its application and much more effective.

This tutorial is not meant to be self-contained. You may have to look things up in the A+ Reference Manual, particularly in the later sections. Also, while many of the test expressions can be executed using F2, as described in the Getting Started tutorial, others must be typed in directly. Moreover, you must type in any test expressions of your own design, and any expressions that create your own test data. Consequently, you should have a printout of the A+ keyboard showing how to type the special graphic characters (see Chapter 2 and Appendix B of the A+ Reference Manual).

2. Creating Test Arrays

The primitive functions illustrated in this section are:
Reshape: Dyadic ⍴
Interval: Monadic ⍳
Roll and Deal: Monadic and Dyadic ?
Specification: Dyadic ←

In order to experiment with expressions in A+ it is very helpful to know how to produce test data, and the primitive functions illustrated in this section are the basic tools for doing it.

Examples

Execute each of the following:

10 20 3 ¯2 ⍀ A 4-element vector of 10, 20, 3, and minus 2.
2 2⍴10 20 3 ¯2 ⍀ 10, 20, 3, minus 2 arranged 2-by-2.

```
  10⍴25 ⍝ A vector of 10 copies of 25.
?10⍴25 ⍝ 10 random numbers between 0 and 24.
⍳5 12 ⍝ The integers 0 through 59 arranged 5-by-12.
  3 9⍴100  ⍝ A 3-by-9 matrix of 27 copies of 100.
?3 9⍴100         ⍝ 27 random numbers between 0 and 99 arranged
⍝   3-by-9.
  20?100 ⍝ 20 distinct random numbers between 0 and 99.
  100+5×⍳2 3       ⍝ 100, 105, ..., 125 arranged 2-by-3.
  3 5⍴10 2 31 107  ⍝ A 3-by-5 matrix filled with 10, 2, 31, and
⍝   107, repeated as necessary.
'abcdefgh'     ⍝ A vector of 8 characters.
4 2⍴'abcef'      ⍝ A 4-by-2 matrix filled with a, b, c, e, and
⍝ f, repeated as necessary.
  12.012 ¯10e5    ⍝ A vector of two floating point numbers.
⍳5 7 3 ⍝ The integers 0 through 104 arranged
⍝   5-by-7-by-3.
a←⍳5 12 ⍝ The variable a now has the array value of
⍝   the expression on the right of the arrow.
a                ⍝ Enter the name alone and its value is
⍝ displayed.
```

3. Characteristics of Arrays

The primitive functions illustrated in this section are:
Shape: Monadic ⍴
Indexing: [;⍳;]
Count and Choose: Monadic and Dyadic #
Type: Monadic ∨

Some A+ Primitives deal with the most basic properties of arrays. Arrays have:

- shape, e.g. the number of rows and columns of a matrix;
- count, e.g. the number of rows of a matrix;
- type, this is, whether the elements are characters, integers, etc.

Not only can the shape of arrays be determined, arrays can reshaped into any shape (see the examples of dyadic ⍴ in the preceding section.) And, finally, their elements can be extracted, either one at a time, or in contiguous blocks called subarrays.

Examples

Execute each of the following:

```
ρ?10ρ25
V?10ρ25
ρ↻5 12
V↻5 12
ρ'abcdefgh'
V'abcdefgh'
ρ12.012 ¯10e5
V12.012 ¯10e5
c←'abcdefgh'⍝ Make a character vector c.
c[0] ⍝ The first element in c.
0⌷c
c[7] ⍝ The last element in c.
7⌷c
c[0 7 2 1]   ⍝ 4 elements of c in a 4-element vector.
0 7 2 1⌷c
c[2 3ρ 7 2 3 3 1 5] ⍝ 6 elements of c arranged 2-by-3, with
⍝   element 3 repeated.
⍴c
(2 3ρ 7 2 3 3 1 5)⍴c
xy←?4 6ρ75 ⍝ Make an integer matrix xy.
xy[2;3] ⍝ The element at row 2, column 3.
(2;3)⌷xy
xy[2;3 0 4] ⍝ The elements at row 2, columns 3, 0 and 4.
(2;3 0 4)⌷xy
xy[1 2 0;3] ⍝ The elements at rows 1, 2 and 0, column 3.
(1 2 0;3)⌷xy
xy[0 2;0 2 1] ⍝ The cross-section of rows 0 and 2 and
⍝   columns 0, 2 and 1.
(0 2;0 2 1)⌷xy
xy[0 2 0;] ⍝ Rows 0, 2 and 0 again.
(0 2 0;)⌷xy
xy[;4 1] ⍝ Columns 4 and 1.
(;4 1)⌷xy
xy[0 2 1] ⍝ Rows 0, 2 and 1.
0 2 1⌷xy
```

The shape of an array x, as evaluated by ρx, is a vector. It therefore has a shape of its own, which is ρρx. ρρx is called the rank of x. The rank of x is the number of axes of x. Execute each of the following:

```
ρρxy
ρρ↻4 5
ρρ'abcdeftg'
ρρ1 2 ¯6 10
ρρ4
```

The last example illustrates an array of rank 0, which is called a scalar. A scalar has no axes, and therefore can't be indexed with bracket indexing. The shape of a scalar is an empty vector. For example, execute:

```
⍴4
```

and nothing prints out in your A+ session (there is a "new line", however, which puts the next prompt two lines down instead of one.)

4. Scalar Functions

This tutorial assumes that you are familiar with the idea of scalar functions and how they apply to arrays (see the A+ Reference Manual or the Scalar Functions tutorial).

5. Structural Functions and the Items of Arrays

The primitive functions illustrated in this section are:
Take and Drop: ↑ and ↓
Replicate and Expand: / and \
Reverse and Rotate: ⌽
Catenate: ,
Laminate: ~

The complementary notions of leading axis and items are important in understanding how many of the structural functions in A+ work (a structural function is one that rearranges the elements of arrays, but does not change their values. For example, 10×x is not a structural function because it multiplies the values of the elements of x by 10.)

The leading axis of an array in the first one indexed. For example, in a[i;j;k] the leading axis is the one indexed by i; in b[n;m] it is the one indexed by n; in c[v] it is the only axis. The items of an array are the subarrays obtained by indexing the array with an index expression having the following properties:

> the leading axis is indexed with a single, scalar value;
> all other axes are indexed with null.

For example, all the following index expressions produce items. Execute them:

```
m←⍳3 5
```

```
m[0;]
m[2;]
a←⍳5 2 7
  a[3;;]
a[1;;]
v←⍳10
v[1]
v[8]
```

In fact, items are such an important idea in A+ that there is a special, bracket notation for producing them, no matter what their rank. Repeating the above examples:

```
m[0]
m[2]
a[3]
a[1]
v[1]
v[8]
```

Most A+ structural primitive functions apply along the leading axes of their right arguments and rearrange the items of that argument. The following are good test values to see how items are rearranged by these primitive functions. Bring them into the A+ session with F2.

Examples

```
v←⍳10
m←⍳10 10
```

Ex 1. ↑ denotes the dyadic primitive function called take. For a positive left argument n, the result is the first n items of the right argument. For example, execute:

```
3↑v
```

Since ↑ is defined in terms of the items of the right argument, what do expect 3↑m to be? Confirm your guess by evaluating the expression.

Ex 2. Experiment with n↑v and n↑m when n is greater than 10. Describe what you see in terms of items. Will this be true for character arrays too? What is true for character arrays?

Ex 3. When n is negative, n↑a is the last -n items of a. For example, try

```
¯4↑v
```

What do you expect `¯4↑m` to be?

Ex 4. Repeat Ex 2 when n is negative and less that `¯10`.

Ex 5. `0↑a` is an empty array, and consequently you cannot learn much by looking at it. Still, `0↑a` is consistent with `n↑a` for nonzero n, relative to its type and shape. Use the above test data and create some of your own to compare `V0↑a` and `Va`, as well as `ρ0↑a` and `ρa`. Describe what you see, and in particular describe `ρ0↑a` in terms of the shape of the items of a.

Ex 6. `↓` denotes the dyadic primitive function called drop. For a positive left argument n, the result is all but first n items of the right argument. Both its symbol and definition suggest that it is a complementary function to take. Rephrase Ex 1 - Ex 4 for drop and then do them.

Ex 7. Ex 5 deals with `0↑a`. Without testing it, describe what you think `0↓a` is. Now test it. Fill in the indicated expression below:

```
0↑a equals ( )↓a
                   ∧
                   What goes here?
```

Ex 8. More generally, for `n≥0` fill in the indicated expression:

```
n↑a equals ( )↓a
                   ∧
                   What goes here?
```

Ex 9. Fill in the expression in Ex 8 so that it works for both negative integers n as well.

Ex 10. Look up the definitions of replicate and expand, rotate and reverse. Be sure you understand how they apply to the items of their right arguments (or, in the case of reverse, to its only argument.)

Ex 11. Repeat Ex 10 for catenate. Note that catenate has an interesting special case when one argument has rank one less than that of the other one: the argument of lesser rank must have the same shape as the items of the other argument. For example, suppose that the argument of smaller rank is a. Then the definitions of both `a,b` and `b,a` are reduced to the equal rank case by replacing a with `(1,ρa)ρa`.

Ex 12. The primitive function called laminate, which is denoted by ~, is an interesting variation in the way it uses the concept of items. Namely, it is dyadic and applies to arrays with the same shape. It produces an array with two items, which are identical to the two arguments. For example:

```
1 2 3 ~ 4 5 6
```

When one of the arguments is a scalar, that argument is reshaped to the shape of the other. For example:

```
1 2 3 ~ 4
```
and
```
3 ~ 4 5 6
```

Test this primitive with other arrays. Try to answer the following little puzzle: for which arrays x and y are the results x,y and x~y identical? When you find the answer, you will have found the case where the general definition of catenate based on items breaks down. It is a useful special case, but one you must watch out for in expressions that apply catenate to arrays of varying rank.

6. Reduction and Scan

Reduction is an example of a mathematical entity called an operator, and it is called an operator in A+ as well. The operator is denoted by /, and it applies to certain scalar functions to produce a new function, called a derived function. For example, +/ is the derived function for + and is called + reduction, or summation. If v is a numeric vector then +/v sums the items of v. E.g., +/3 5 7 is 15. That is:

```
+/v equals v[0]+v[1]+⍳+v[¯1+#v]
```

This definition holds for v of any rank because v[i] denotes the ith item of v, no matter what the rank.

Scan is also an operator. For example, +\ is the derived function for + and is called partial sums. If v is a numeric vector then +\v is the partial sums of the items of v. E.g., +/3 5 7 is 3 8 15.

See the Scalar Functions tutorial for an introduction to these operators.

Ex 13. Make sure you understand that reduction and scan apply to the

items of arrays. Test your understanding with matrices and arrays of rank two and three (e.g., ⍳3 5 and ⍳3 5 2.)

7. Frames and Cells

Frames and cells are a more general way of partitioning arrays than leading axis and items. For example, consider:

```
a←⍳3 8 4
```

The array a has three axes. In an index expression of the form

```
a[i;j;k]  ⍝ This expression is not executable unless
⍝   i, j, and k are given values.
```

The axis indexed by i is called the leading axis. If i is a scalar, the subarrays a[i;;], or equivalently a[i], are the items of a. In terms of frames and cells, the leading axis is the frame of rank 1 and the items are the cells of rank 2.

Similarly, the first two axes are the frame of rank 2. If both i and j are scalars then the subarrays a[i;j;], or equivalently a[i;j], are the cells of rank 1.

To complete the picture:

the three axes of a are the frame of rank 3 and the scalar elements are the cells of rank 0;

the empty set of axes is the frame of rank 0 and the array a itself is the cell of rank 3.

The rank of the frame plus the rank of the cells within that frame equals the rank of the array. Instead of always referring explicitly to the rank of the frame and the rank of the cells, it is sometimes convenient to refer to "the frame for the cells of rank n", or the "the cells within the frame of rank k."

The concepts of frame and cells are useful in explaining several A+ primitives. We will look at these primitives before the main topic of this section, which is the rank operator.

7a. The Primitive Functions Called Member Of and Index Of

In its simplest form, where the arguments are scalars or vectors, the result of Member Of is a boolean vector where 1's mark the elements of the left argument that appear in the right. For example:

```
1 ¯5 4 10 9 ∈ 9 7 ¯5
 0 1 0 0 1
```

More generally, the definition of the function y∈x is this: let N denote the rank of the items of the right argument x, and let S denote their shape. Then the left argument y must have rank at least N, and the N cells of y must have shape S. If so, y∈x is defined and is a boolean array; its shape equals the shape of the frame for the cells of rank N; and an element of the result is 1 if the corresponding cell is identical to at least one item of the right argument x, and 0 otherwise.

Ex 14. Let's see if we understand the definition of Member Of. Define

```
x←⍳3 4
y←6 4 ⍴ ¯1 10 5 2 8 9 10 11 17 0 1 ¯9 5 4 6 7 0 1 2 3 7 9 ¯1 0
```

and look at these arrays by executing the following:

```
x
y
```

Since the rank of y equals the rank of x, the cells of y that are relevant to y∈x are just the items of y. Before evaluating y∈x, can you predict the rank of the result? The shape? The value? Evaluate y∈x to check your answer.

Ex 15. Replace y in the Ex 13 as follows:

```
y←3 2 4⍴y
```

Now, what is the rank of the cells of y that are relevant to y∈x. What is rank of the frame that holds these cells? What is the shape of the frame? What is the rank and shape of the result y∈x? What is the value? Evaluate y∈x to check your answer.

Ex 16. Replace y in the Ex 14 as follows:

```
y←8 9 10 11
```

Repeat Ex 14.

Ex 17. Index of is similar to Member Of. It is denoted y⍳x. There are two basic differences in the definitions. One, the roles of the left argument and right argument are interchanged from those of Member Of. That is, in Member Of, cells of the left argument are compared to items of the right. In Index Of, however, cells of the right argument are compared to items of the left. The second difference is that the value of Index Of is an array of integers: if a cell of the right argument is identical to an item of the left argument, the corresponding element of the result is the index of that item; if it matches more than one item, the element is the smallest index among those it matches; if there is no match, the element is the number of items in the left argument.

Here's a simple example:

```
'mxaz' ⍳ 'O1zAx'
 4 4 3 4 1
```

Make sure you understand this result.

Ex 18. Form x and y as in Ex 13. Describe the result of x⍳y in terms of the items of x and the appropriate cells of y.

Ex 19. Form y as in Ex 14, and then describe the result of x⍳y in terms of the items of x and the appropriate cells of y.

Ex 20. Form y as in Ex 15, and then describe the result of x⍳y in terms of the items of x and the appropriate cells of y.

7b. The Rank Operator

By now you should appreciate the uniformity with which many of the A+ primitive functions apply to the items of their arguments. You may also be wondering whether or not this is too restrictive. For example, what does one do to catenate one matrix to another row-by-row, instead of itemwise? Well, the answer to this question lies in another operator, called the rank operator and denoted by @.

The definition of the rank operator is based on the concepts of frames and cells, which were just introduced. In effect, the rank operator specifies the rank of the cells to which a monadic function is to be applied, or the ranks in the case of a dyadic function.

Ex 21. One of the easiest ways to see how the rank operator applies is with reduction. For example, execute:

```
ρ+/⍳3 5 8
```

The result shows that the leading axis disappears; it is the axis over which the reduction took place. Now execute:

```
ρ(+/@ 1)⍳3 5 8
```

The expression +/@ 1 means that reduction will be applied to cells of rank 1 of the array ⍳3 5 8. (The parentheses around +/@ 1 in the above expression are not necessary, but have been included for emphasis.) The cells of rank 1 are vectors along the last axis. The last result shows that the last axis is the one over which the reduction took place.

Now here is the challenge: execute the following and explain what you what see:

```
ρ(+/@ 2)⍳3 5 8
```

Hint: Answer the following questions. What are cells of ⍳3 5 8 to which the reduction is applied? What is the result when ordinary reduction is applied to those cells?

Here are two more challenges: compare ⍳3 5 8 and (+/@ 0)⍳3 5 8 and explain what you see; compare +/⍳3 5 8 and (+/@ 3)⍳3 5 8 and explain what you see.

Ex 22. In the rank operator we specify the function to be applied and the rank(s) of the cells to which it applies. For example:

```
a←3 4ρ'abcdefghujkl'
b←3ρ'ABC'
```

The expression

```
a(,@1 0) b
```

expresses the catenation of the rank 1 cells of a to the rank 0 cells of b. That is, each row of a is catenated to the corresponding element of b. Evaluate this expression and make sure that you understand what you see.

The rule of frames is:

When the Rank operator is applied dyadically, the shape of the

corresponding frames must be equal,

if they are of the same rank, and otherwise the shape of the
frame of lower rank, say r, must equal the last r dimensions
of the shape of the other frame.

Define

```
c←'zyxwvut'
```

Evaluate a(,⍀1 1)c and explain what you see in terms of the rule of frames.

Ex 23. For each of the following expressions explain what you think the result should be. Test your understanding by evaluating the expressions:

```
a(⍀,2 2) b
a(⍀,0 0) b
a(,⍀1 0)'MNO'
a(,⍀0 1)'MNO'
a(,⍀2 0)'MNO'
```

Consult the A+ Reference Manual regarding 3-element data operands and negative elements of data operands.

8. Idioms and Phrases

The purpose of this section is to illustrate the expressiveness of A+ and to help you get in the habit of experimenting with the A+ expressions you come across in order to understand them. Most topics begin with one or more A+ expressions, followed by test data in a subsection called Example(s). Use F2 to first bring in the test data, and then go back to the expressions themselves and apply F2 to them to see what they do.

8a. A To B

Make a vector of integers from a to b, where a is less than or equal b.

```
a+⍳1+b-a
```

Example

```
a←10
```

```
b←17
```

Ex 24. Modify this expression using max (⌈) and min (⌊) to work whether a is less or equal b, or vice versa.

Ex 25. Repeat using absolute value (monadic |) and signum (monadic ×) in place of ⌈ and ⌊.

8b. Bar Graph

This example uses the outer product operator (denoted by ∘.). For example, ∘.+ is the outer product of +. It is a dyadic function. x∘.+y forms an array consisting of the sums of all pairs of elements consisting of one from x and one from y. See the Scalar Functions tutorial for an introduction to the outer product operator.

Here is a use of another outer product:

```
' □'[v∘.≥⍳⌈/v]
or
' □'[(⍳⌈/v)∘.≥v]
```

Example

```
v←?20⍴20
```

Ex 26. Execute the above expressions for the test data and explain what you see. In particular, what role does ⍳⌈/v play?

8c. Remove Duplicate Items

```
((v⍳v)=⍳⍴v)/v
```

Example

```
  v←?30⍴10
```

Ex 27. The sample data is guaranteed to have duplicates. Why?

Ex 28. The key here is the expression (v⍳v)=⍳⍴v. To understand what's going on, execute v⍳v and ⍳⍴v separately so that their elements line up. Explain how duplicates can be identified by examining these two rows. (If the elements do not line up, execute

```
(v⍳v)~⍳#v
```

Why does this work)?

Ex 29. Try the above expression on matrices:

```
v←(3 5⍴'abcdefghijklmno')[?8⍴3]
```

Explain what you see.

8d. Append

```
d←(1#⍴a)⌈#b

(d(↑⍤0 1)a),d↑b
```

Example

```
a←'abcdfgeh'[?4 5⍴8]
b←'ABCDEFGHIJK'
```

Ex 30. Experiment with d↑a vs. d(↑⍤0 1)a. Describe the differences. The symbol ⍤ denotes the Rank operator. The notation ↑⍤0 1 signifies "the rank of ↑ applied to scalar (rank 0) items on the left and vector (rank 1) items on the right."

Ex 31. Modify the expressions in this section to apply to a matrix b?

8e. Eliminate Leading Blanks from a Vector

```
((v≠' ')⍳1)↓v
or
(\v≠' ')/v
or
(+/∧\v=' ')↓v
```

Example

```
v←'   abc de f'
```

Ex 32. Experiment with the expression (v≠' ')⍳1 and describe its behavior.

Ex 33. Experiment with the boolean expression \v≠' ' and describe its

behavior.

Ex 34. Experiment with the boolean expression ∧\v=' ' and describe its behavior.

8f. Eliminate Trailing Blanks from a Vector

You will write the expressions in this exercise.

Example

```
v←'std sqz  dabc  '
```

Ex 35. Use rotate (monadic ⌽) to modify the expression (∨\v≠' ')/v so that trailing blanks are removed.

Ex 36. Use rotate (monadic ⌽) and anything else to modify the expression (∨\v≠' ')/v so that trailing blanks are removed.

Ex 37. Repeat the last exercise for (+/∧\v=' ')↓v.

8g. Eliminate Multiple Blanks from a Vector

```
((1↓z)∨¯1↓z←1,' '≠w)/w
```

Example

```
w←'   srt  5cp   qrsx  '
```

Ex 38. Examine the boolean expression (1↓z)∨¯1↓z by first evaluating 1↓z and then ¯1↓z. Their elements should line up in the display, so that you can compare them quite easily. Explain how this expression removes duplicate blanks.

Ex 39. What is role is played by the catenation by 1 in 1,' '≠w? Suppose the 1 is replaced with 0. How does this change things?

8h. Left Justify a Character Vector

```
((v≠' ')↻1)⌽v
or
(+/∧\v=' ')⌽v
```

Example

```
   v←'   abc de f'
```

Ex 40. Compare these expressions to the ones used for eliminating leading blanks. Explain the difference between the effect of eliminating leading blanks and left justifying. Hint: in left justification, where do the blanks go?

8i. Left Justify the Rows of a Character Matrix (preparation)

```
(m≠' ')⍳1
or
+/∧\m=' '
```

Example

```
m←3 5⍴'  abcd e f ghij'

m                          Display m for reference below.
  abc
d e f
 ghij
```

Ex 41. Test these expressions for the sample data. If the expression executes successfully, explain whether it applies to the rows or columns of m.

Ex 42. If an expression applies to the columns of m, modify the expression using monadic transpose (⍉) (see the A+ Reference Manual) so that it applies to the rows.

8j. Left Justify the Rows of a Character Matrix Using the Rank Operator

```
f{x}:((x≠' ')⍳1)⌽x
(f@1) m
```

Example

```
m←3 5⍴'  abcd e f ghij'
```

Ex 43. Explain (f@1) m.

Ex 44. Use the rank operator in the same way to extend the other expression for left justifying vectors to one that applies to matrices.

Ex 45. Here is a challenge for using the rank operator. Instead of defining the function f above and applying the rank operator to it, figure out how to apply the rank operator to the primitive functions in the expression ((m≠' ')⍤1)⌽m, so that the result justifies the rows of a matrix.