# APL Prototype Functions

**Bob Smith**
Sudley Place Software
Shepherdstown, WV USA
bsmith@sudleyplace.com

***Old APL Joke:***
***"How many empty arrays does it take to fill a workspace?"***
***"One, if it's big enough."***

## Abstract

The concept of empty arrays, while not unique to APL, has been most thoroughly developed there. Nonetheless, the rules for handling empty arrays are inconsistent across the major APL implementations where, for example, the sum of two empty vectors can not only give different results, but also, sometimes (and correctly so) a `LENGTH`, `RANK` or `DOMAIN ERROR`. Moreover, on certain implementations addition of empty arrays is not always commutative (e.g., `L+R` does not match `R+L`).

This paper attempts to give a comprehensive treatment of prototypes and set down consistent rules for computations with them in order to encourage all APL implementations to produce the same results. Along the way, we introduce the concept of a prototype function – a function associated with each primitive and derived function to be used in lieu of the original function when applied to empty argument(s).

## Rationale

From the very beginning, APL designers and implementors have prided themselves on how the primitive functions worked "as expected" in the limiting empty array case. Few if any other languages take such care to handle such edge conditions, largely because empty arrays play such a small role elsewhere. In APL, empty arrays are huge!

Programmers don't normally start out creating empty arrays, they just happen in the course of normal (or abnormal) computation. This is why developers need to expend the effort to ensure that their implementation handles them seamlessly. By treating empty arrays as a natural and limiting case of data, developers can shift the burden from programmers so the latter can concentrate on algorithms, not edge conditions. More than in most languages, it is the APL developers' responsibility to simplify the programmers' job, part of which is to get the edge conditions right. Effort in support of this goal can reduce programmers' frustration as they wonder why a particular piece of code doesn't seem to work, or works differently from vendor to vendor.

## Introduction

Before nested arrays (call that era APL1, a simpler time for empties), there were but two types: numeric (`⍬`) and character (`''`), both of which could be reshaped to an arbitrary rank as long as one or more dimensions remained at zero.

The **prototype** of a simple array was then defined to be the scalar **zero** (the **prototypical number**) for numeric arrays and the scalar **blank** (the **prototypical character**) for character arrays.

In the process of defining the various primitives in APL1, the question came up of what to do with the take function (or more accurately the overtake function) when its left argument was greater than the size of the right argument, e.g. `5↑1 2` or `5↑'ab'`. In these cases the decision was made to pad or fill the elements in the result with the prototype of the right argument, yielding `1 2 0 0 0` and `'ab   '`, respectively.

Moreover, in order not to require the programmer to test for and handle empty arrays separately, the concept of fill was extended to include them. The question of whether an empty array was numeric or character could not be decided by looking at their elements because there are none. Instead, the fill element is a separate attribute of an empty array and is inherited from the function and/or array which created it. The fill elements for `⍬` and `''` are defined to be a zero and blank, respectively, and `⍳0` is defined as creating an empty numeric array. All other primitive functions that may create an empty array have a well-defined fill element for the result which depends upon the right argument. Not incidentally, these definitions had the pleasant side effect of providing a way to tell the difference between the two types of APL1 arrays, as in `0=1↑0⍴R`.

Nested arrays are a different story. When they were included in the language, they brought not only more complicated empties but also more complicated (derived) functions whose argument(s) may be empty. Nested arrays introduce structure at arbitrary depths below the topmost which then raises the question of their fill element. For uniform nested arrays (arrays each element of which has the same rank, shape, and type at all depths), the choice is easy – fill with the common element after replacing all numbers and characters with their respective prototypical value. For non-uniform arrays, the choice was made to use the first element of the array. This choice was made based on More's Theory of Arrays[1], and is explained in more detail in Orth[2] and Gull/Jenkins[3].

The prototype of an arbitrary non-empty array is its first item with the numbers and characters replaced by their prototypical elements.

What then is the prototype of an empty nested array – what is its first item if it's empty? To answer this, we need to consider how the array became empty in the first place.

The array isn't just empty, but contains a memory of what (to some extent) it was like before it became empty. For example, given a vector of pairs of numbers, the prototype is then a pair of numbers, and if we reshape that array to be empty

```
      ↑0⍴(1 2)(3 4)
0 0
```

the prototype is a typical pair of numbers: `0 0`.

The most common way to create an empty array is via reshape, but the same argument applies to the few other ways to create an empty array (replicate, expand, indexing, take, drop, etc.). When a non-empty array is reshaped to be empty, the prototype of the result is the prototype of the first element of the non-empty array.

**Note**: In this paper the verb **prototype** when applied to an array refers to the process of taking an array and replacing all of its numbers at all levels with zeros and all of its characters with blanks. The noun **prototype** when applied to a non-empty array refers to the first item of the array (`↑R`) with all of its numbers at all levels replaced with zeros, and all its characters replaced with blanks – it has been prototyped (using the verb form); when applied to an empty array, it refers to the prototype of the non-empty array from which it was created.

## Identities on Empties

How should we define how empty arrays and their prototypes interact with each other? Sadly, the Extended APL Standard[4] is mostly silent on the topic of prototypes (called therein **typical-elements**) of nested arrays, leaving it to an **implementation-defined-algorithm**. However, one time-honored way to decide is via identities defined on non-empty cases and then see how they extend to the empty case.

For example, we all know the identity that says that primitive scalar monadic and dyadic functions distribute over selection functions (in fact, this is what makes them **scalar**):

```
(KρL) f KρR ←→ KρL f R
```

for `f` a primitive scalar dyadic function (**psdf**) and conformable `L` and `R`. Is there any reason not to expect this identity to hold in the empty (`K=0`) case?

## Try This

In any modern APL interpreter, if you enter

```
(⊂0 (0 0)) + ⊂(0 0) 0
```

you can expect it to return the value

```
⊂(0 0) (0 0)
```

Correspondingly, when you “empty” each argument as in

```
(0ρ⊂0 (0 0)) + 0ρ⊂(0 0) 0
```

based on the above identity, you should expect it to return the value

```
0ρ⊂(0 0) (0 0)
```

a result returned by most, but not all of the major APL implementations.

In fact, in those other implementations, addition (along with `×⌈⌊=≠∨∧⍱⍲`) between empty arrays is not always commutative. For example,

```
      L←0ρ⊂0 (0 0)
      R←0ρ⊂(0 0) 0
      (L+R)≡R+L
0
```

## Errors, too

The same argument from above can be made for error conditions. You expect to signal a `LENGTH ERROR` when you enter

```
(⊂0 0) + ⊂0 0 0
```

and so you should expect one from

```
(0⍴⊂0 0) + 0⍴⊂0 0 0
```

and similarly for `RANK` and `DOMAIN ERROR`s. Again, not all of the major implementations signal these errors – those that don't return one of the (empty) arguments (typically, the right).

## Except For...

Alas, there is one class of exceptions universally applied across APL implementations, a template for which is (say) `''+''`. A simple empty character array may be used in lieu of a simple empty numeric array where a numeric array is normally required. All of the major APL implementations return a simple empty **numeric** vector for the above expression.

Were implementations completely consistent, the above expression would signal a `DOMAIN ERROR` for the same reason `' '+' '` does. This design decision was made in the days of APL\360 as early programmers lazily used `''⍴`... instead of the slightly longer form `(⍳0)⍴`... (saving two characters and one primitive!). After that, it was a hop, skip, and a jump to allow things like `''+''`, and so a convenience/exception was born.

## The One Coin Flip Case

There's really only one type of expression in APL1 where the answer isn't completely clear: `'',⍬` or its twin `⍬,''`, or any equivalent higher rank expression. According to the rules of catenation we know the answer must be a simple empty vector, we just don't know whether it's numeric or character. Across all major APL implementations, the common rule (even when empty nested arrays are considered) seems to be to choose the prototype of one of the arguments as the prototype of the result. Alas, one vendor chooses the left, and all the others the right.

## In General

In order to provide a general rule for how primitive and derived functions act on empty arguments, it is convenient to associate with each such function a **Prototype Function** whose purpose is to act as a substitute for the original function on the same empty arguments to produce the result. This auxiliary function is chosen to signal errors as appropriate, but as we'll shortly see, not too many errors.

The **psdf**s are the simplest cases where implementations differ, so let's define their prototype functions. For empty and rank- and length-conformable `L` and `R` (modulo the exception mentioned above), there are two cases depending upon the **psdf**. In the first case, for the two **psdf**s `L=R` and `L≠R`, the prototype of the result is

```
↑0⍴⊂(↑L)≠↑R
```

For all other **psdf**s, the prototype of the result is

```
↑0⍴⊂(↑L)+↑R
```

There are two cases because the **equal** and **not-equal** functions accept characters in either or both arguments, and none of the other **psdf**s do. Hence the first form does not signal a `DOMAIN ERROR` if either side contains a character, and the second form does; both forms may signal `RANK` or `LENGTH ERROR`s.

Because we are defining a prototype, we must ensure that its leaves contain prototype values (zeros or blanks) only, hence the leading `↑0⍴⊂` is used to convert numbers to zeros (at the moment, no **psdf** returns characters). Note that both of these algorithms are defined recursively on the depth of the arguments.

The prototype function for **equal** and **notequal** is **notequal** because that function doesn't signal a `DOMAIN ERROR`, and the prototype function for all the other **psdf**s is **plus** because it does signal a `DOMAIN ERROR` on characters, but doesn't signal an error on zeros as `0⍟0` might.

The prototype of applying a primitive scalar monadic function to an empty argument `R` is just `+↑R`. The prototype function for all primitive scalar monadic functions (**psmf**s) is **plus**, so as to signal a `DOMAIN ERROR` on characters, but not in the case of (say) `÷¨⍬`.

In general, the rationale for choosing a particular prototype function is that although zero is the prototypical number, it happens not to be in the domain of all **psmf**s. It seems inappropriate to single out (say) divide because of its somewhat more limited domain; it works on almost all numbers, just not all numbers. This same rationale applies to prototype functions for **psdf**s, primitive non-scalar functions, and derived functions.

## Primitive Non-Scalar Functions

Going over the list of such functions, it appears that the only ones that need to be treated specially are Matrix Inverse and Matrix Divide. All of the others have an obvious definition when used in a prototype context. To illustrate the two exceptions,

```
      (0 3⍴⊂0 0 0)≡⌹¨0 3⍴⊂0 0 0
1
```

and

```
      (0 3⍴0)≡(⊂0 0 0)⌹¨0 3⍴⊂0 0 0
1
```

These two functions require special treatment so as not to signal an error on singular arguments as `⌹0 0 0` and `0 0 0⌹0 0 0` both do.

The prototype function for Matrix Inverse (monadic `⌹`) is `+∘⍉`. This expression uses the Compose operator (see **Compose** in the next section) which when used monadically applies its operands sequentially to its argument. The transpose function is present so as to get the same result shape as does Matrix Inverse, and the plus function is present to weed out characters.

The prototype function for Matrix Divide (dyadic `⌹`) is `⊢.+⍨∘⍉`, which is less complicated than it looks This expression uses both the Compose and Commute operators, the latter of which is defined in the Extended APL Standard[4]. These operators allow the prototype function to be written as a single derived function.

Through the application of various definitions and identities, the shape of the result of this expression changes from

```
    ρL⊢.+⍨∘⍉R
 ↔  ρL⊢.+⍨⍉R                  Definition of Compose operator
 ↔  ρ(⍉R)⊢.+L                 Definition of Commute operator
 ↔  (¯1↓ρ⍉R),1↓ρL             Shape rule for Inner Product operator
 ↔  (1↓ρR),1↓ρL               Definition of Transpose function on rank < 3
```

the last line of which is the shape rule for Matrix Divide. This shows that the above mentioned prototype function for Matrix Divide has the same shape rule as does Matrix Divide and, like that function, disallows characters (via the `+` in the inner product). The occurrence of `⊢` is irrelevant because it is not invoked in this case (see **Inner Product** in the next section).

## Derived Functions

The following derived functions may be applied via the each operator on an empty argument to invoke its prototype function, or if one or more of its operands is applied on an empty argument or between empty arguments. The corresponding prototype function is defined below. In the following descriptions, **`e`** represents the prototype function for `f`, and **`h`** represents the prototype function for **`g`**.

### Inner Product: `L f.g R`

In the table below, `m`, `i`, `n` are all positive integer scalars. Without loss of generality, we can assume that `L` and `R` are matrices.

| | `ρL` | `ρR` | **Result** | **Comments** |
|---|---|---|---|---|
| 1. | `0 i` | `i 0` | `0 0ρ⊂(↑L)h↑R` | (**`f`** not called) |
| 2. | `0 i` | `i n` | `0 nρ⊂(↑L)h↑R` | (**`f`** not called) |
| 3. | `m i` | `i 0` | `m 0ρ⊂(↑L)h↑R` | (**`f`** not called) |
| 4. | `m i` | `i n` | Normal case | |
| 5. | `0 0` | `0 0` | `0 0ρ⊂(↑L)h↑R` | (**`f`** not called) |
| 6. | `0 0` | `0 n` | `0 nρ⊂(↑L)h↑R` | (**`f`** not called) |
| 7. | `m 0` | `0 0` | `m 0ρ⊂(↑L)h↑R` | (**`f`** not called) |
| 8. | `m 0` | `0 n` | `m nρf/0ρ⊂(↑L)h↑R` | (**`f`** called once) |

The idea in the above cases (except for cases 4 and 8) is that the result is empty so it must have a prototype. That value can be computed from the left and right argument prototypes as the left and right arguments to **h**. For case 8, the common value in the m by n result can only come from the **f** reduction of an empty vector which produces an identity element (if any).

The prototype function for **f.g** is invoked in cases 1-3 and 5-7 only, and is **⊢.h** where **⊢** is irrelevant because it is not invoked in this case.

**Outer Product: L ∘.g R**

The outer product operator uses its operand in a scalar manner, so it's no different from a **psdf**. The prototype of the result is (↑L)**h**↑R.

The prototype function for **∘.g** is **∘.h**.

**Compose: f∘g R and L f∘g R**

The compose operator applies the two functions sequentially, so the prototype function for **f∘g** is **e∘h**.

**Reduction: f/[X] R and L f/[X] R**

The reduction operator on an empty argument invokes an identity function, which does not involve prototypes.

The reduction operator applied via the each operator on an empty argument applies **f/[X]** to the prototype of **R**. In order to avoid the same errors as in the **psdf** case, the proper thing to do is to apply **e/[X]**.

The prototype function for **f/[X]** is **e/[X]**.

**Scan: f\[X] R**

For the same reasons as reduction, the prototype function for **f\[X]** is **e\[X]**.

**Duplicate/Commute: f⍨ R and L f⍨ R**

The duplicate/commute operator applies its operand between the argument(s), so the prototype of its result in the monadic case is (↑R)**e**↑R, and in the dyadic case (↑R)**e**↑L.

The prototype function for **f⍨** is **e⍨**.

**Each: f¨[X] R and L f¨[X] R**

The each operator uses its operand in a scalar manner, so it's no different from a **psmf**/**psdf**. The prototype of the result in the monadic case is **e**↑R, and in the dyadic case (↑L)**e**↑R.

The prototype function for **f¨[X]** is **e¨[X]**.

**Rank: f⍤[X]v R and L f⍤[X]v R**

The rank operator applies its operand to or between subsets of its argument(s). The argument(s) are partitioned according to **v**, but the operand is applied normally.

The prototype function for **f⍤[X]v** is **e⍤[X]v**.

## Recursive Definitions

When applying a derived function via (say) the each operator to an empty argument, the above rules should be applied recursively. If **f** is a derived function applied as in **f¨0ρ⊂R**, find the principal operator of **f** and look it up in the above tables, and continue applying the rules until you get to a primitive function.

For example, in the case of **∘.÷/¨0ρ⊂R**, the prototype function for **f/** is **e/** where **e** is the prototype function for **f** (which is **∘.÷**). The prototype function for **∘.f** is **∘.e** where **e** is the prototype function for **f** (which is **÷**). Finally, the prototype function for **÷** is **+**. The prototype function for **∘.÷/¨** is **∘.+/¨**.

## Conclusion

The goal of prototype functions is to provide a consistent treatment of primitive and derived functions when applied in the context of empty argument(s), either via the each operator on empty argument(s) or in certain cases on or between empty argument(s). Prototype functions are defined for **psdf**s, **psmf**s, and the operators, and then recursively for more complicated derived functions.

By following the above rules, all APL implementations can return the same results for the same argument(s), simplifying the programmers' job as well as reducing code migration issues.

## References

1. T. More, "**Types and Prototypes in a Theory of Arrays**", Technical Report No. G320-2112. IBM Cambridge Scientific Center, Cambridge, MA, May 1976.

2. Orth, Don, "**Empty arrays in extended APL**", IBM Journal of Research and Development, Volume 28, Issue 4 (July 1984), pp. 412-427.

3. Gull, W.E. and Jenkins, M.A., "**Decisions For "Type" In APL**", Sixth Annual Symposium on POPL Symposium Proceedings, 1979, pp. 190-196.

4. Extended APL Standard (ISO/IEC 13751:2001).

## Acknowledgement

The anonymous reviewer made many helpful suggestions and is much appreciated.

## Appendix T: Test Cases

The prototype of `(0ρ⊂0 (0 0)) + 0ρ⊂(0 0) 0` should be `(0 0) (0 0)`.

The prototype of `(0ρ⊂0 0) + 0ρ⊂(0 0) 0` should be `(0 0) 0`.

The expression `(0⍴⊂0 0) + 0⍴⊂0 0 0` should signal a `LENGTH ERROR`.

The expression `(0⍴⊂0 0) + 0⍴⊂1 3⍴0 0 0` should signal a `RANK ERROR`.

The expression `(0⍴⊂'abc')+0⍴⊂'abc'` should signal a `DOMAIN ERROR`.

The result of `⌹¨0 3⍴⊂0 0 0` should be `0 3⍴⊂0 0 0`.

The result of `(⊂0 0 0)⌹¨0 3⍴⊂0 0 0` should be `0 3⍴0`.