APL Trivia

Edward Cherlin
APL News
6611 Linville Drive
Weed, California, U. S. A.

Everything is either trivial or undecidable.—Mathematician's joke.

ABSTRACT

APL derives great expressive power from seemingly trivial features, such as empty arrays, but there is still resistance to the implementation of the elementary trivial functions including left /stop ⊣ and right/pass ⊢, which perform no calculations. This paper defines functional triviality; describes trivial functions and operators and their uses; discusses the mathematical basis of their expressive power; and defines new trivial functions and operators. It urges implementation of several of these functions as primitives for reasons of efficiency and expressiveness, and considers their efficient implementation via idiom recognition.

Introduction: Trivia and Triviality Defined

Definition 0. The definitive collection of APL Trivia is the Trivial Pursuit card deck produced for APL86 [Zi86]. For example, in the General category, card 78: "According to the draft ISO APL standard, what is the name for the function in the expression *R*←→*W*?" The answer is "conjugate", and in APLs without complex arithmetic it is a trivial function, the identity function with domain all real numbers. It is this sort of trivial function that we are pursuing here—functions and operators that do neither arithmetic nor array manipulation.

Definitions 1 and 2. "trivial 1. Of little importance or significance. 2. Ordinary; commonplace." [Am82]

Donald McIntyre has frequently drawn attention to the importance of certain trivial features of APL [McI83]. Comparing the use of empty arrays with the conceptual advance represented by the invention of the 0 in India, he has gone on to praise even more APL's ability to return, not merely a 0 or an empty answer, but nothing at all as the result, for example, of ⍬''.

Edsger Dijkstra placed a real value on the trivial programs *skip* and *abort* in [Di76]. The first succeeds for any input that is acceptable as output simply by doing nothing. In his notation for the weakest precondition wp(*skip*,R)=R, which is to say that any proposition that is true before a skip remains true afterwards. The second fails on all input. Aborting the program cannot result in any statement about the computation becoming true, so (wp(*abort*,R)=F). In other words, if there is an Abort statement anywhere in a program then correct execution is guaranteed only if that statement can never be executed. Implementations of *skip* and *abort* appear below.

At first sight the functions to be discussed here appear to be useless in programming, and of slight value in teaching, fulfilling definition 1. For example, several authors list all 16 possible truth tables for two variables, and point out that six of them are trivial and have no corresponding function in APL, while the other ten are all important, and are represented by the four logical and six comparison functions. [Be87] is a recent example. This paper is a contribution to fulfilling definition 2 by encouraging the implementation and routine use of these and other trivial functions and operators.

The determination of what is trivial in sense 1 is based on expectations of usage, not just on the mathematical definitions. For example a similar analysis of the logical primitives produced a somewhat different result in the construction of Loglan, a language designed to make mathematical logic speakable [Br75, pp. 272-279]. Fourteen of the possible truth tables were "implemented" in Loglan, that is given corresponding words. The sentence forms

p u q	whether q, p
p nuu q	whether p, q
p nou q	whether q, not p
p nuunoi q	whether p, not q

correspond to common forms in natural languages, where p and q stand for any Loglan sentences and "u", "nuu", "nuunoi" and "nou" are Loglan words. For example, "Mi gotso u tu gotso." is Loglan for "I'm going whether you go or not." These four words also correspond to the left and right functions of Dictionary APL and their negations. The constant-valued predicates *T*:1 and *F*:0, applied to two arguments, are the remaining two of the 16 possible Boolean functions.

Consider these expressions in Dictionary APL [Iv87] or SAX APL, ordered by increasing complexity of results:

Table 1
Trivial expressions

```
          +0 ←→ DOMAIN ERROR
        ⍬'' ←→
        →'' ←→
          ⊣⍵ ←→
  (m⍤0) ⍵ ←→ m
α (m⍤0) ⍵ ←→ m
          ⊢⍵ ←→ ⍵
        α⊢⍵ ←→ ⍵
        α⊣⍵ ←→ α
      m⍥f ⍵ ←→ m f ⍵
      f⍥m ⍵ ←→ ⍵ f m
    α f⍤2 ⍵ ←→ ⍵ f α
    f⍤g ⍵ ←→ f g ⍵
```

The arguments α and ω are scalars in the third and fourth examples and the monadic rank of g in the last example is unbounded. In each case the answer is no more than we started with, or even less. Since they accomplish nothing we can properly say that these expressions are trivial (sense 1).

What is gained by writing the expression on the left in place in the value it produces on the right? Well—by itself, nothing. But then, what is the value of writing any computer program if you already know its results? Who, for example, writes 2+2 in APL as anything but a trivial exercise to introduce APL to novices? Who writes 0∨1? We write programs to produce results that are more complicated or at least larger in volume than the program, or to carry out calculations too tedious to do by hand.

So where do these functions and operators produce results that are not trivial? In combination with other functions and operators or features that have or induce side effects. For example, v by itself prints a value, and $v \leftarrow \omega$ suppresses printing. So $\vdash v \leftarrow \omega$ does something different, both assigning and displaying the value. Naturally this can also be done as $\square \leftarrow v \leftarrow \omega$.

The purpose of this paper is to examine this set of trivial APL functions and operators. They have been neglected because their operations were thought to be trivial (sense 1) and not worth implementing. Some have been discussed in various papers, especially [Iv87] and [Iv89], and some have been implemented, notably the commute operator ~ in Dyalog APL [Dy82], and some Dictionary APL primitives or variations in versions of Sharp APL [Sh89]. Iverson has remarked on their individual utility but not on the general principal of triviality in mathematics, which underlies their importance.

TRIVIALITY IN APL

For our purposes triviality can be defined thus:

Definition 3. An APL function or operator is completely trivial if it can be completely defined by the shortest possible identity $c \leftarrow\rightarrow exp$, where c is null, an error, an expression of a single character, or a name or description of a single APL object, and exp is of the simplest kind, such as $\alpha\ f\ \omega$. A less trivial class allows simple expressions on either side of the definition, but not the use of any other functions.

This is not a mathematically precise definition, but a precise definition could be made if there were good reason for it. Coarse distinctions suffice since it is the trivial functions themselves that concern us and not their exact degree of triviality on some necessarily arbitrary scale.

The extreme form of triviality for functions consists in producing results that do not depend on the values of arguments (constants or null), and a slightly less trivial class returns an argument unchanged. The most trivial function of all was described in [Ar83]—a niladic function without result. In direct definition, such a function can be written `nil f:⍬`. It is exactly Dijkstra's skip. An almost equally trivial function is `fail:+0`, which implements Dijkstra's abort.

An ambivalent nil function with arguments, `∘:⍬0⍴⍵` (or the monadic stop function in SAX APL, written $\dashv\omega$), the constant zilde `⍬←→⍳0` in some dialects, the constant logical functions `t:1` or `f:0` and other constant-valued functions are the next most trivial, followed closely by the identity function $\vdash$ ($\omega \leftarrow\rightarrow \vdash \omega$), the identity operator $f \leftarrow\rightarrow f\ id$ and the left and right functions $\dashv$ ($\alpha \leftarrow\rightarrow \alpha \dashv \omega$) and $\vdash$ ($\omega \leftarrow\rightarrow \alpha \vdash \omega$). These are completely trivial as defined here.

The somewhat less trivial class includes the swap operator, defined as $\alpha\ f\subset\ \omega \leftarrow\rightarrow \omega\ f\ \alpha$ [Iv87] and the analogous commute operator of Dyalog APL. Subscripting and other selection functions are not trivial because, although they return unchanged elements of one argument, they select them based on the other argument. Structural functions in APL are far from triviality but also derive much of their importance from simple identities. For example ⌽ and ⍉ both satisfy the relation $x \leftarrow\rightarrow f\ f\ x$. However neither is defined by that identity. The mathematical functions are the least trivial primitives, and system functions are in general totally non-trivial.

MATHEMATICAL TRIVIA

Mathematics is the study of the complex results of trivial definitions. Building the loftiest structures firmly on the slightest possible supports has been a goal of practitioners since well before Euclid. The attempt to reduce all of mathematics to the trivial, i. e. to axioms and rules of inference which could be accepted as obvious, was a major part of mathematics from Euclid to Hilbert. The program has failed in its grand goal of proving the truth or at least the consistency of mathematics, but it has succeeded admirably in demonstrating the importance of trivia. Trivia such as identity elements and empty sets occupy an important and honored place in the foundations of set theory and in algebraic structures including groups, rings and fields.

The definition of primitive recursive functions by Kurt Gödel shows a masterful grasp of the trivial [Gö31]. Two families of trivial functions (the constant 0-valued functions of one or more variables, and the projection functions which return one of the arguments), plus the minimal non-trivial arithmetic function $s:\omega+1$ suffice as a foundation for all of mathematics together with substitution and recursion.

The branch of mathematics that makes the most of trivia is Curry's combinatory logic [Cu58]. A combinator is, approximately, a trivial operator. The primitive combinators are defined by the trivial relations listed in Table 2. These also suffice to define all of mathematics. In particular Curry showed that any computable function can be expressed entirely in this set of combinators with no variables.

Each of these combinators corresponds somewhat to a Dictionary APL function or operator, as also shown in Table 2, although combinators are more general. A combinator can be applied to any combinator expression, whereas an APL function can only be applied to an expression that returns an array value. Even in the most liberal APL product there are no operators that apply to operators to create new operators, although the idea has been tried in experimental versions such as [Gf89].

Table 2
Primitive combinators

Name	Definition	Description	APL analog
I	Ia=a	identity	$\vdash\alpha \leftarrow\rightarrow \alpha$
K	Kab=a	left	$\alpha \dashv \omega \leftarrow\rightarrow \alpha$
C	Cabc=acb	swap	$\alpha\ f\subset\ \omega \leftarrow\rightarrow \omega\ f\ \alpha$
B	B(ab)c=abc	composition	$f\circ g\ \omega \leftarrow\rightarrow f\ g\ \omega$
W	Wab=abb	duplicate	$f\subset\omega \leftarrow\rightarrow \omega f \omega$

Another useful technique from combinatory logic is to attach an argument to a dyadic function to produce a monadic function, as in lines 9 and 10 in Table 1, using Iverson's *with* operator. This operation is called currying, after combinatory logic founder Haskell Curry. The most familiar use of currying to APLers is the definition of the trigonometric functions as special cases of ○ using functions analogous to $SIN:1\circ\omega$.

USES OF TRIVIALITY

The assumption that trivial functions are useless comes as we have seen from viewing them in isolation. A trivial function or operator adds nothing to a simple expression. But in combination they become quite powerful, just as combinators are trivial individually but suffice for all of mathematics.

The simplest use of trivial functions is in conjunction with side effects of expressions, especially assignment and display. Thus

```
⊢⍺←exp
```

which Iverson used extensively in the Dictionary, or

```
+⍺←exp
```

(if complex arithmetic is not implemented) are equivalent to

```
□←⍺←exp
```

As Iverson noted in the Dictionary, the left and right functions can be used to provide the benefits of the non-functional comment and statement separator notations, but with greater flexibility and right-to-left execution. Reading left as "where" it is possible to unravel quite complex expressions into sequences of short, digestible, idiomatic APL, and incidentally to remove parentheses to any degree desired. [Ch89a] points out that the left function and direct definition allow any computable function to be expressed in one line of APL.

Here is the substitute for the diamond separator

```
x⊥4 8 1⊣x←⍟y
```

and the next example is the substitute for comment. It requires quote marks, but it also allows comments in the middle of a line.

```
x←'comment'⊢exp
```

An indication that a trivial function is needed is the frequent idiomatic use of some other useless expression such as

```
0 0⍴exp1,0⍴exp2
```

which could be written more clearly with the monadic nil and dyadic left functions as

```
⍬exp1⊣exp2
```

or with dyadic nil as

```
exp1⍬exp2
```

Other idioms have been used in place of the nil function, such as assigning a value to a variable and never using it. This is wasteful of storage and confusing to readers and optimizing compilers.

There is a class of new operators that apply their function operand in some characteristic pattern, where it may be useful to have the pattern and in effect no function. Cut, defined in the Dictionary of APL [Iv87] and implemented in SAX APL [Sh89] is a good example. It breaks a vector into subvectors and applies its operand to each, but sometimes breaking the vector is sufficient. For example:

```
      ¯1⍤<'This is a test'
This is a test
      ¯1⍤⊢'This is a test
This
is
a
test
```

In the first case the box function is applied, resulting in a vector of boxed vectors. In the second, using an identity function, a rectangular array results when the vectors are catenated after being brought to the same shape.

Trivial operators will come into their own when arrays of functions and operators as in Functional Programming [Ba78], NIAL [Je85] or Sharp Australia's APL/HP [Ho86] are accepted, or when more general operators are implemented (such as a LISP-like λ or Landaeta's *lift* elevator[La86, La88]), and the expressive power of compound operator expressions is thereby greatly increased. Proposals for functions arrays go back to Iverson's first book [Iv62] and have been renewed by Benkard, Bunda and others.

The possibilities for operators with function vector arguments or operator vector arguments are far more varied than for operators limited to two single operands. Consider a simple example. We take two arguments, ⍺ and ⍵, together with the monadic function *f* and the dyadic functions *g* and *h*. There is no problem at all in forming the array `(f ⍺)(⍵ g ⍺)(⍺ h ⍵)`, written here in strand notation, in any nested APL, although syntaxes differ. There is also no problem writing a function that takes the two arguments and returns this result:

```
foo: (f ⍺)(⍵ g ⍺)(⍺ h ⍵)
```

But we cannot, in most APLs, write an expression for such a function. Let us suppose then that we have a syntax for function vectors (and here I choose to use strand notation again) and define the trivial Left operator

```
    ∇ ⍺ f L ⍵
[1]   f ⍺
    ∇
```

Now the function vector `(f L) (g ⊂) h` does just what we want, where ⊂ is the swap or commute operator :

```
⍺ (f L) (g ⊂) h ⍵
(⍺ f L ⍵)(⍺ g ⊂ ⍵)(⍺ h ⍵)
(f ⍺)(⍵ g ⍺)(⍺ h ⍵)
```

A Right operator is equally trivial and equally useful. Indeed the operator analogues of all of the trivial functions are immediately useful.

Other Trivial Functions and Operators

It is a useful exercise to write out tables of all trivial relations and then play around with some of them. Another useful exercise is to see how such primitives have been used in other areas, especially combinatory logic and functional programming. The completely trivial functions with null or simple result have mostly been put to use in some APL dialects, but a few remain unexplored and unexploited.

The constant functions have been particularly neglected. One advantage of defining a function with a constant value is that it cannot be changed by assignment. Recognition of the constant function idiom permits efficient evaluation without function call overhead, and can also allow pre-evaluation of functions that will always return the same answer. This is known as "constant folding" in the design of optimizing compilers (see [Bu88]). A few frequently-used constants are `pi:○1`, `e:*1`, `max:⌊/⍳0`, `min:⌈/⍳0`, and `zilde:⍳0`.

This finishes off the gaps in the completely trivial function table for functions with null or minimally complex values. The next level, operators returning trivial combinations of their operands and arguments, has not been thoroughly explored in APL, but some guidance is available from combinatory logic and from the few cases that have been tried.

Here is a table of the simplest cases. Uses for many of these combinations are unknown today but may be discovered later, just as uses for many of the 441 inner products of the old APL appeared

many years after the general concept was defined, and many other inner products are being discovered in nested APLs. The possibilities are many, for monadic or dyadic functions as operands to monadic or dyadic operators and producing monadic or dyadic functions as results. For each column, results in some columns to the left are also possible and many more complicated expressions are ignored.

Table 3
Trivial operators

`f op ω`	`α f op ω`	`f op g ω`	`α f op g ω`
`ω`	`f α`	`g ω`	`α f g ω`
`f ω`	`α f α`	`f g ω`	`f α g ω`
`ω f ω`	`α f ω`	`g f ω`	`α g f ω`
`f f ω`	`ω f α`	`ω f g ω`	`g α f ω`
`ω f f ω`	`α f f ω`	`ω g f ω`	`ω f g α`
		`f ω g ω`	`f ω g α`
		`g ω f ω`	`ω g f α`
		`ω f ω g ω`	`g ω f α`
		`ω g ω f ω`	`α f α g ω`
			etc.

Iverson and McDonnell have found two combinators, Sfgx=fx(gx) and Φfghx=f(gx)(hx) to be valuable enough to suggest implementing them as extensions to APL syntax, without any operator symbol [Iv89]. The defining identities, in APL infix notation rather than the prefix notation of combinators, would be

```
   (f g)ω ↔ ω f g ω
  α(f g)ω ↔ α f g ω
 (f g h)ω ↔ (f ω) g h ω
α(f g h)ω ↔ (α f ω)g α h ω
```

The second of these (called dyadic hook) is the most trivial in appearance. It allows expressions like `(-+)\ω` for continued fractions. The first (monadic hook) permits `(+ ∨/)ω,1` for generating rational approximations to real numbers, where ∨ is the greatest common divisor function as in Sharp APL. Thus `(+ ∨/).75 1` is `.75 1+.25` or `3 4`. A trivial hook operator can be defined as

```
Hook:: α f g ω : 0=⎕nc'α' : ω f g ω
```

in an extension of direct definition with formal operands *f* and *g*.

The third and fourth forms (monadic and dyadic fork) create function vectors in the form +,-,×,+ (with multiple applications of the same syntax), or compound functions such as <∨= for ≤, or with currying `>¨0 ∧ ≤¨100`. These forms are extremely useful and cannot be represented directly using APL operators, since there is no way to supply three operands. It would be necessary to write an operator corresponding to each central function, supplying that function implicitly. But this is multiplying entities beyond necessity [Oc30]. Other notations for function vectors, hyper-operators and the like abound in the APL literature. [Iv62, Ho86, Gf86, Je85] are among the most notable.

Assuming a class of objects that accepts functions and returns an operator, which can be called a transform, we define a trivial fork transform

```
Fork::: (α f ω) H α g ω : 0=⎕nc'α' : (ω f ω) H
      ω g ω
```

in a further extension of direct definition. Here the fork transform applies to the function H to produce a dyadic operator which in turn creates an ambivalent function.

It is obvious that we do not want to implement all possible trivial operators as primitives, if only because there are so many. But it is not necessary to. The forms corresponding to the usual combinators, and perhaps a few of the more complicated but powerful combinators, would give a basis for producing any of the forms listed here and also much more complicated expressions. The trick is to identify a set which combines ease of learning (implying a small set of conceptually simple and regular forms) with expressive power (implying theoretical completeness and a sufficient selection to make expressions concise, as in other aspects of APL notation). These conflicting criteria indicate that a period of investigation and trial are needed to pick a reasonable set.

Alternatives

Trivial functions are trivial to implement in any APL. The various nested array APLs with user-defined operators make it easy to define a set of trivial operators and try them out. So there is no need to rush into implementation. The operators can also be simulated in flat APL using the technique of [Ch89b].

As usual, there is a tradeoff between cluttering up the workspace with defined functions and operators and cluttering the reference manuals, and there are serious questions of efficiency. Also as usual, the problem of efficient implementation will be resolved when these functions come into widespread use.

Proposals have sometimes been made for extensions to APL syntax that would avoid the invention of new symbols, functions or operators. Strand notation is a well-known example. The fork and hook constructs of [Iv89] are others. [Bk90] gives references to several more, including a notation for currying in the form (2+) as implemented in NIAL, and a variety of suggestions for function arrays. It also suggests a notation in which expressions involving functions with the α and ω symbols can be treated as definitions of "nonce functions". In the function vector example given earlier, the function could be applied directly to its arguments in the form

```
A (f α)(ω g α)(α h ω) W
```

Nonce function vectors would save most applications of explicit trivial operators. For example currying coud be done in the form `(2+ω)W`. This style would take some time to learn to read and is inconsistent with direct definition, but the concept could be implemented in some other syntax.

Implementation

Several trivial functions and operators can be implemented so that there is no time or space penalty for using them. In particular there is no need to incur the overhead of a function call for functions such as nil, left or right, or for identity functions and operators. Of course most APLs use reference counts for arrays, so that array arguments do not have to be copied, but in these cases even the incrementing and decrementing of reference counts can be avoided. Similarly a commute or swap operator can be implemented in the interpreter logic rather than as an operator call. These cases are just like the recognition of other idioms such as ⍴⍴ or ⍳⍴ for which intermediate values do not have to be materialized in the workspace.

The first function I define in any new I-APL workspace is left. I have begun to add right and nil, which I use less often. I don't bother to define fork, since it is as much work to use it as a simulated defined operator as to write out the expressions it generates. But if it were part of the syntax, as Iverson has proposed, or even a primitive, I would use it constantly.

Although there were several votes against them, the right and left functions have been accepted in the working draft for the second APL standard [ISO90].

Conclusions

The move toward adoption of left and right functions, and the beginnings of user-defined operators, in the proposed new standard is a step in the right direction toward full support of APL trivia. The trivial operators can be left as defined operators until there is more experience with them and with other alternatives, and with their inteeractions with function arrays. The stop function ($-\omega$) of SAX APL or a monadic nil ($\circ\omega$), together with dyadic nil ($\alpha\circ\omega$) would be extremely useful additions to APL that would make many programs more readable.

References

"Yea, from the table of my memory
I'll wipe away all trivial fond records..."
Hamlet, Helsingør, ca. 900
by William Shakespeare, ca. 1600

Am82 American Heritage Dictionary, Second College Edition, Houghton Mifflin 1982

Ar83 Bob Armstrong, "The Minimal Expression", APL Quote Quad, 14, 1, September 1983, pp. 13–14.

Ba78 Backus, "Can Programming be Liberated from the Von Neumann Style? A Functional Style and Its Algebra of Programs", Turing award lecture, CACM **21**,8, August 1978, pp. 613-641.

Be87 Gary A. Bergquist, *APL:Advanced Techniques and Utilities*, p. 232. Zark, 1987.

Bk90 J. Philip Benkard "Nonce Functions", these Proceedings.

Br75 James Cooke Brown, *Loglan 1: A Logical Language*. Loglan Institute, 1975.

Bu87 Timothy Budd, *An APL Compiler*. Springer-Verlag New York, 1987.

Ch89a Stop ∧ Trace, APL News, **21**,1 Springer-Verlag New York 1989.

Ch89b. ⍎, APL News, **21**,4 Springer-Verlag New York 1989.

Cu58 Haskell B. Curry and Richard Feys, *Combinatory Logic*. North-Holland 1958.

Di76 Edsger W. Dijkstra, *A Discipline of Programming*. Prentice-Hall 1976.

Dy82 Dyalog APL Manual, Dyadic Systems

Gf86 Martin Gfeller, "Operators Considered Harmful", APL Quote Quad, 17, 1, p. 7.

Gf89 Martin Gfeller, "A Future APL: Examples and Problems", APL89 Conference Proceedings, APL Quote Quad 19, 4 August 1989.

Gö31 Kurt Gödel, "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I", Monatshefte für Mathematik und Physik, Vol. 37, pp. 349-360. Translated into English as "On Formally Undecidable Propositions of the Principia Mathematica and Related Systems I", in Davis, *The Undecidable*, Raven Press 1965.

Ho86 Robert Hodgkinson, "APL Procedures", APL86 Conference Proceedings, APL Quote Quad, 16, 4, pp. 179-186.

ISO90 Programming Language APL, Extended, International Standards Organisiation, Working Draft 1. Eugene McDonnell, Editor, February 15, 1990.

Iv62 K. E. Iverson, *A Programming Language*. Wiley 1962.

Iv87 K. E. Iverson, "A Dictionary of APL". APL Quote Quad , 18, 1 Sept. 1987 pp. 5-40.

Iv89 K. E. Iverson and E. E. McDonnell, "Phrasal Forms" APL89 Conference Proceedings, APL Quote Quad, **19**, 4, pp. 197–199. Corrections, APL News, **20**, 3 pp. 5–6 and **20**, 4 p. 1, Springer-Verlag New York 1989.

Je85 Michael A. Jenkins and William H. Jenkins, Q'NIAL Reference Manual, NIAL Systems Ltd., July 1985.

Oc30 William of Ockham, *Summa Totius Logicae*. Ca. 1330.

Sh89 I. P. Sharp Associates, SAX Language, Edition 1.2, March 30, 1989.

Zi86 APL Trivial Pursuit card deck, edited by David Ziemann for the APL86 conference, Manchester, England, U. K.