Progress In NARS2000
October 2017 to September 2019

Bob Smith
Sudley Place Software
Originally Written
7 Aug 2019
Updated
16 Sep 2019

Released Features

Language Features

Implement Ball Arithmetic

Up to now, the simple scalars of APL have been single values, even though they may have multiple axes such as Hypercomplex numbers (e.g., `1i2`) – they are still considered single points in the appropriate space such as the Complex plane. Ball Arithmetic is different – here, a simple scalar in Ball Arithmetic is a range of Floating Point (FP) numbers such as `1.2±1E¯10`. This very intuitive notation represents a Ball of Radius `1E¯10` around the midpoint of the Ball, `1.2`. The purpose of this datatype is to provide a first cut of Numerical Analysis of an algorithm. That is, instead of analyzing your algorithm's use of FP numbers, first run it through with Ball Arithmetic.

For example, here are three ways of calculating the square root of two, first using fixed precision FP, second as Multiple-Precision FP, and last using Ball Arithmetic:

```
      √2
1.414213562373095
      ⎕FPC←128
      √2v
1.41421356237309504880168872420969807857
      √2±
1.41421356237309504880168872420969807857±5.9E¯39
```

The last result is qualitatively different from the first two in that, as a Ball, it is **guaranteed to contain the exact answer**, in this case, for the Square Root function, but **regardless of the primitive function**. We know that the Square Root function returns an irrational result guaranteed to be **inexact**. Since we can't return an exact result, the next best representation is as a Ball. Essentially, Balls contain a built-in error term (i.e., Radius).

Note that the Radius is a function of the specified precision (in ⎕FPC). If you desire a narrower Radius, just increase the precision as in

```
      ⎕FPC←256
      √2±
1.41421356237309504880168872420969807857±1.7E¯77
```

By doubling the number of bits of precision, we narrowed the Radius by about 38 orders of magnitude! The additional precision in this result is not displayed only because of a small ⎕PP.

If your FP algorithm is fed Balls as input, it propagates them **without type demotion** where each successive primitive calculates the appropriate Ball Radius for its result. At the end the result is guaranteed to contain the exact answer. No more guessing, no more hoping that there are no ill-conditioned dark corners in your FP code. If you are concerned about the precision and accuracy of your FP code, you need Ball Arithmetic.

For more details see Ball Arithmetic[1].

Implement Hyperators

The sequence of objects: Arrays, Functions, and Operators appear in ascending order with the property that each later object consumes one or two of the earlier objects and produces an object in the sequence one order less. For example, a Function takes Array(s) and produces an Array; an Operator takes Function(s) and/or Array(s) and produces a (derived) Function.

Hyperators (as described by John Scholes[3]) extend this sequence by one:

- a **Hyperator** has one or two **Hyperands** (which may be any of the three lower classes of Operators, Functions, or Arrays) and yields ...
- a (derived) **Operator** has one or two **Operands** (which may be any of the two lower classes of Functions or Arrays) and yields ...
- a (derived) **Function** has zero, one, or two **Arguments** (Arrays) and yields...
- an **Array**.

This new class of objects leads to new syntax and new ways to describe algorithms in APL. As an example of a primitive Hyperator, there is the Transform Hyperator[4] (DownTackOverbar ⍒ Alt-'B').

For more details, see Hyperators[2].

Implement Numerical Differentiation Operator

While working on the Matrix operator, I've found I needed a Numerical Differentiation operator which is now implemented. The symbol chosen for this monadic operator is CurlyD (∂) (Alt-'D' U+2202) used in mathematics for Partial Derivatives.

For example,

```
      !∂0 ⋄ ¯1g1
¯0.5772156649015363
¯0.5772156649015329
      !∂∂0 ⋄ 1g2+1p2÷6
1.9781119906525646
1.97811990655945
```

That is, the exact value of first derivative of the Factorial function at 0 is $-\gamma$, where γ is Gamma, the Euler-Mascheroni constant, and the exact value pf the second derivative is $\gamma^2+\pi^2/6$. The derived function may be called dyadically, too:

```
      X←0.6 ⋄ 1○∂X ⋄ 2○X
0.8253356149096687
0.8253356149096783
```

where the first derivative of the Sine function is the Cosine function.

For more details, see the paper “Numerical Differentiation in APL”[5].

Implement ⎕DFT (Discrete Fourier Transform)

The usual DFT algorithm is implemented as a Fast Fourier Transform (FFT) algorithm three times, one for each FP datatype:

- Fixed Precision Integer or Floating Point, the FFT algorithm from Gnu Scientific Library is used
- Multiple Precision Integer/Rational or Floating Point, the FFT algorithm MPFFT is used
- Ball Arithmetic, the FFT algorithm from ARB is used

The dyadic version with a left argument of ¯1 returns the **Inverse** DFT.

In the context of Hyperators, this System Function is equivalent to the function

```
      DFT←+{*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J¯2÷⍴⍵}⍤1×
```

where the Anonymous Function is the (Left) Hyperand to the Transform Hyperator (⍤). This specific text is recognized as an idiom

and the appropriate FFT function is executed. Similarly, the **Inverse** DFT function (`¯1 ⎕DFT`) is equivalent to

```
      ⍝DFT←+{(*(⍳⍴⍵)∘.×(⍳⍴⍵)×○0J2÷⍴⍵)÷⍴⍵}⍤1×
```

the text of which is also recognized as an idiom so as to invoke the appropriate **Inverse** FFT code.

For example,

```
      ⎕DFT 1 2 3 4
10 ¯2J2 ¯2 ¯2J¯2
      ¯1 ⎕DFT ⎕DFT 1 2 3 4
1 2 3 4
```

This System Function has been superseded by the Transform Hyperator. For more details, see Hyperators[2].

Implement APL2 Vector Notation W.R.T. Right Operands

Following the lead of APL2, numeric strands as a Right Operand (which has **short right scope**) are split apart with the first number alone as the Right Operand and the rest of the strand forming part of the Right Argument. This treatment minimizes the number of parentheses needed, except in the case of a dyadic operator (such as Rank) when its Right Operand is a multi-element numeric vector.

For example, with this change

```
      +DOP 1 2 3 ↔ (+DOP 1) 2 3
```

For more details, see Vector Notation[9].

This treatment is especially needed with Hyperators both of whose Hyperands are defined to have **short** scope. For more details, see Hyperators[2].

Implement Mask and Mesh

Following the proposal by Iverson[8] and the syntax proposed in the original NARS manual[7], these two functions are implemented as

Mask: `L (a∘/) R` or `L (a∘≠) R` or `L (a∘/[X]) R`

Mesh: `L (a⍤\) R` or `L (a⍤⌿) R` or `L (a⍤\[X]) R`

where ***a*** is a control vector which chooses whether the next element of the result is from the Left or Right Arguments and its multiplicity.

Mask returns **a subset** of the items from the Left and Right Arguments **in order but interleaved**, possibly repeated multiple times and possibly omitting values from one or both arguments, combined into a single result, according to the values in ***a***. In the vector case, and except for scalar extension, the two arguments and the control vector have the same number of elements. During processing of the two arguments and the control vector, indices into all three components are incremented in parallel. A negative value in the control vector selects the same indexed value from the Left Argument; a positive value selects the same indexed value from the Right Argument; a zero skips the same indexed value in both arguments. The absolute value of the control vector value indicates how many times the selected value is repeated in the result.

For example,

```
      'ABC' (¯1 3 ¯2⍤/) 'abc'
AbbbCC
      (3 3⍴⎕A) (¯1 3 ¯2⍤/) 3 3⍴⎕a
AbbbCC
DeeeFF
GhhhII
```

The function `L (LO⍤/[X]) R` is essentially equivalent to

```
(|LO~0)/[X] (⊂(⍋⍋×LO,-LO)[⍋⍋×LO~0])⌷[X] L,[X] R
```

Mesh returns **all** items from the left and right arguments **in order but interleaved**, possibly repeated multiple times and possibly including the Right Argument's fill element, combined into a single result, according to the values in ***a***. During processing of the two arguments and the control vector, indices into all three components are incremented independently. A negative value in the control vector

selects the next value from the Left Argument; a positive value selects the next value from the Right Argument; a zero inserts the Right Argument's fill element. The absolute value of the control vector value indicates how many times the selected value is repeated in the result.

For example,

```
  'I' (1 ¯1 2 ¯1 2 ¯1 2 ¯1 ⍤\) 'MSSP'
MISSISSIPPI
  'BIB' (¯1 ¯1 1 ¯1 1 0 1 1 2 1 1 1 ⍤\) 'LOBAGINS'
BILBO BAGGINS
```

This function `L (LO⍤\[X]) R` is essentially equivalent to

```
(|LO)\[X] (⊂⍋⍋×LO~0)⌷[X] L,[X] R
```

Note that this definition is dependent upon extending the Expand function to (Signed) Integer Left Arguments.

Also, notice the amount of code shared between the two algorithms, especially the interesting snippet `⍋⍋×LO~0`.

Implement 'ah' Point Notation

Complementing the three other Polar (or Angle) Point Notations of `'ad'`, `'ar'`, and `'au'`, the `'ah'` point notation maps the Angle into the interval `[¯0.5, 0.5]` where the Line of Discontinuity is the **negative** real axis, as opposed to the other three forms where the Line of Discontinuity is the **positive** real axis.

Allow All Input Point Notations As Output Forms

With this change, all Input Point Notations may appear in Output. In particular, the Complex Number Display Separator now has more choices, all controlled by the System Variable `⎕FC[7]`. The value of `⎕FC[7]` may be one of `'iJdhru'` which correspond to the separators `'i'`, `'J'`, `'ad'`, `'ah'`, `'ar'`, and `'au'`. For example,

```
      fc←{⎕FC[1⊃⍺]←2⊃⍺ ⋄ ⍵}
      7 'i' fc 2i¯3
2i¯3
      7 'J' fc 2i¯3
2J¯3
      7 'd' fc 2i¯3
3.6056ad303.7
      7 'h' fc 2i¯3
3.6056ah¯0.1564
      7 'r' fc 2i¯3
3.6056ar5.300
      7 'u' fc 2i¯3
3.6056au0.8436
```

All Numeric Output Display Is Sensitive To ⎕FC

For example,

```
      ⍝ Decimal point, Negative sign, Complex sep
      (1 6 7)':@J' fc 2.3i¯4.5
2:3J@4:5
```

The only exception to this rule is ⎕FMT which has its own symbol substitution mechanism.

Implement Chained Guards In AFOs

By Chained Guards, I mean two or more consecutive Guard statements in an Anonymous Function/Operator, such as

```
        f←{
➥0=≡⍵:     ⍝ Scalar
➥0≡⍴⍵:     ⍝ Simple Numeric
➥1==⍵:     ⍝ Real
➥0=2|⍵:    ⍝ Even
➥'Scalar, Simple Numeric, Real, Even'
➥ ⋄ 'None of the above'}
        f 2
Scalar, Simple Numeric, Real, Even
        f 'a'
None of the above
        f ¯2
Scalar, Simple Numeric, Real, Even
        f 1i2j3k4
None of the above
```

The several conditions cannot be combined into a single APL statement separated by the And function because the interpreter will evaluate all of the conditions even if an early one proves to be FALSE, and then signal an error on (say) character input.

Session Manager and Workspaces

Use Line Continuations In Output For Wrapped Lines

For example, with a narrow Page Width,

```
       )LOAD J:\workspaces\allmfs
SAVED 08/10/2019 16:18:34
      ⎕pw←40
       )ops
 DetSing   DydConv   DydDot2   DydMask
➥DydMesh   DydRank   DydScan   DydScan1
➥IdnConv   IdnJotDot MatOpr    MonDot
➥MonDotCr  MonRank   RoS2       RoS3
➥RoS1L     RoS1R
```

Implement)BOX [ON|OFF]

This just means that output may be run through Monadic ⎕FMT before display.

Implement Workspace Features To Allow Multiple Non-Downward Compatible Features To Co-exist

This allows me to have different levels of workspace formats for each project, such as Ball Arithmetic and Hyperators each of which has its own different format for storing its new functions and datatypes.

Preserve Visual Fidelity When Mixing CRs, LFs, And BSs In Character Output

In other words, CR moves the output cursor to the first column of the line, LF moves it to the same column in the next line, and BS moves it to the preceding column unless that's the first column.

For example,

```
      'abc',⎕TCLF,⎕TCBS,'d',1,⎕TCLF,'ef',⎕TCNL,'gh'
 abc
➥   d 1
➥gh    ef
```

In order to see the formatting characters, turn on Output Debugging as in "Edit | Customize... | User Preferences | Enable Output Debugging", and uncheck "Except for CR and LF":

```
      'abc',⎕TCLF,⎕TCBS,'d',1,⎕TCLF,'ef',⎕TCNL,'gh'
abc 00 0A 00 08 d 1 00 0A e f 00 0D gh
```

Allow (and Ignore) Underbar In Numeric Input

The Underbar is used as a visual separator when entering long numbers. It is ignored when calculating the value of the number. For example,

```
      12_345_678
12345678
```

Make More Commands Sensitive To Line Continuations

In particular, commands that produce multiple lines of output such as `)FNS`, `)VARS`, `)OPS`, `)LIB`, etc. display a Line Continuation Marker at the start of a Continued Line.

Implementation Features

Rename MFO (Magic Function/Operator) Locals To Avoid Conflict With User Variables

This means that the local names in a MFO are in a different Namespace from the user-accessible names, so that if a Magic Operator calls a User-Defined Function/Operator from within the MFO, the UDFO's locals cannot see the MFO's locals. In particular, because the MFO local names all start with a `$`, those names are inaccessible to UDFOs.

Miscellaneous Features

Change Definition Of `¯9○` and `9○` Functions

The Old and New definitions are

	`¯9○R`	`9○R`
Old	`R`	`Re(R)`
New	`(R-+R)÷2`	`(R++R)÷2`

For all numbers, the change to `9○` is insignificant: the only difference might be whether the result is still a Hypercomplex number whose Imaginary part(s) are all `0`, or is a Real number – solely an

implementation choice. In either case, the value is the same.

For Complex numbers, the change to `¯9○` is significant: the result changes from `R` to a copy of `R` with the Real part set to `0`, and th

e Imaginary parts unchanged. This differs from the previous result where both the Real and Imaginary parts are unchanged. However, given the trivial nature of the old definition, I'd be very interested to read how anyone uses it.

For Hypercomplex numbers, the change to `¯9○` is very useful: the result is now a **Pure Imaginary** number. That is, the result is the original number with the Real part set to `0`. Once you get beyond Complex numbers, this concept is an integral part of the various definitions and theorems on Hypercomplex numbers. Even Wikipedia[6] uses the shorter term **Imaginary number** for the more accurate term **Pure Imaginary number** indicating how often that concept is needed.

Note the pleasant symmetry between the new definitions of `9○R` and `¯9○R` along with the identity `R≡+⌿¯9 9○.×R`.

For example, the definition of a Cross Product of two Quaternions numbers is now

```
{¯9○<(○⍥¯9○⍺)+.×>¯9○⍵}
```

where

- `>⍵` splits out the coefficients of a Hypercomplex number into a Real vector of length 1, 2, 4, or 8
- `<⍵` is the inverse of `>⍵`
- `○⍥` returns the Matrix Representation of a Hypercomplex number.

Implement Lexicographic and Gray Code order of certain results from Combinatorial operator

When the Left Operand of the Combinatorial Operator has two numbers, the second one is a flag where `0` means Count the

answers, 1 means Generate the answers, 2 means Generate the answers in Lexicographic Order (alphabetic), and 3 means Generate the answers in Gray Code order (single swap from one line to the next).

For example,

Permutations	
Lexicographic Order	**Gray Code Order**

`      110 2‼3`
`1 2 3`
`1 3 2`
`2 1 3`
`2 3 1`
`3 1 2`
`3 2 1`
	`      110 3‼3`
	`1 2 3`
	`1 3 2`
	`3 1 2`
	`3 2 1`
	`2 3 1`
	`2 1 3`

Implement 10 □AT To Distinguish Function Types

For example, where **FOH** = **F**unction, **O**perator, or **H**yperator:

		Function Example	Operator Example	Hyperator Example
0	The object is **not** a **FOH**			
1	**Primitive FOH**	`f←+`	`f←/`	`f←⍤`
2	**Derived FOH**	`f←+/`	`f←⍨∘÷`	`f←⍤⍤×`
3	**User-defined FOH**	`∇f`	`∇(LO f) R`	`∇(LO (LH f RH)) R`
4	**System FOH**	`f←⎕NL`		
5	**Anonymous FOH**	`f←{α+ω}`	`f←{αα/¨ω}`	`f←{ααα⍨αα/¨ω}`
6	**Train**	`f←(+/÷≢)`	`???`	`???`
7	**Name-Associated Function**	Not as yet implemented		
8	**Java-Associated Function**	See Java Support		

Implement Ascending/Descending Subsequences

An **Ascending** subsequence of one vector in another (`L⍳⍠'a' R`) is a set of indices of `L` (barring items Not Found) such that `R≡L[L⍳⍠'a' R]` and `∧/2<⌿L⍳⍠'a' R`, that is the indices are monotonically increasing. This primitive is written such that it returns the smallest values that satisfy the above conditions. All datatypes are valid for the left and right arguments; the result is always an integer vector of the same length as `R`.

For example:

```
      L←2 7 1 5 7 1 2 1 7 1
      R←5 2 7 1
      L⍳⍠'a' R
4 7 9 10
      L[L⍳⍠'a' R]
5 2 7 1
```

A **Descending** subsequence of one vector in another (`L⍳⍠'d' R`) is a set of indices of `L` (barring items Not Found) such that `R≡L[L⍳⍠'d' R]` and `∧/2>/L⍳⍠'d' R`, that is the indices are monotonically decreasing. This primitive is written such that it returns the largest values that satisfy the above conditions. All datatypes are valid for the left and right arguments; the result is always an integer vector of the same length as `R`.

For example:

```
      L←1 2 5 7 2 1 5 7 1 2 5 5 1 2 7
      R←5 2 7 1
      L⍳⍠'d' R
12 10 8 6
      L[L⍳⍠'d' R]
5 2 7 1
```

Both of these useful idioms are hard to write non-looping in APL, so this is one way to gain access to them.

Implement Additional Hypercomplex Multiplication Variants

For `L×⍠C R`, the following definitions hold for Quaternion/Octonion (Non-commutative) numbers:

C	Name	Quaternion/Octonion — Non-commutative
`'i'`	**I**nterior product	`((L×R)+R×L)÷2`
`'e'`	**E**xterior product	`((L×R)-R×L)÷2`
`'x'`	**Cross** product	`¯9○<(○⍟¯9○L)+.×>¯9○R` **Quaternions only** where `¯9○R` returns `(R-+R)÷2`, i.e. as a pure imaginary number
`'d'`	**D**ot product	`(>L)+.×>R`
`'c'`	**C**onjugation product	`L×R÷L`

Handle `¯0` In More Cases

Negative zero is a great idea foiled by the lack of hardware support. IEEE-754 Floating Point numbers support it well, but Integers don't.

Among other places, it is useful as a value in the identity `R≡÷÷R`. If `R` is `¯∞`, then `÷R` is `¯0`, and `÷÷R` is back to `¯∞`. However, if we don't support `¯0`, then the identity fails because `÷¯∞` is `0`, not `¯0`. Essentially, the two numbers `¯0` and `¯∞` go together – support one, support the other.

The problem is that there is no hardware representation of `¯0` as a **64-bit integer**. As a result, when we negate a Boolean or Integer zero, the entire array must blow up to FP, which is not very desirable.

I continue to support this feature, but only if you tell me to. That is, if you set `⎕FEATURE[2]←1`, then negative zero should appear when you expect it. Otherwise, don't expect that identity to hold.

Implement APL2's Definition Of Inner Product

The definition of Inner Product in both APL2 and APL+Win is different from the definition in NARS/Dyalog in an interesting way. The two definitions (on vectors) are

```
L f.g R ↔ f/L g  R     APL1, APL2, and APL+Win
        ↔ f/L g¨ R     NARS/Dyalog
```

When Nested Arrays came around, this was extended by NARS and Dyalog to replace `g` in the righthand part above with `g¨` so as to **maintain the same shape rule** for the result. Before APL2 was released, their definition was changed so as not to make that substitution.

One compelling reason I like the APL2/APL+Win definition is that it can emulate the NARS/Dyalog definition simply by writing `f.(g¨)`, but not the other way around. In other words, the APL2/APL+Win definition is more general than the NARS/Dyalog definition and is compatible with NARS/Dyalog for primitive scalar dyadic functions because the Each operator on a PSDF is idempotent. Nonetheless, I went with the shape-rule-preserving definition and support the APL2/APL+Win definition via a feature switch: `⎕FEATURE[5]←1`.

For example,

```
      a←3 6⍴'Queue EschewAchoo '
      a
Queue
Eschew
Achoo
      ⎕FEATURE[5]←0
      a+.∊'aeiou'
LENGTH ERROR
      a+.∊'aeiou'
         ^
      ⎕FEATURE[5]←1
      a+.∊'aeiou'
4 1 2
```

Extend Index Coalescing To Indexed Assignment And Modified Indexed Assignment

Index Coalescing means that you may write as many instances of `[…]` in a row as you like and they will be processed Left-to-Right in the expected way. For example,

```
 a      [2 3⍴⍳6][1 2;2 3][⊂2 2]
≡a               [1 2;2 3][⊂2 2]
≡a                        [⊂2 3]
≡a[6]
```

in origin-1. With this change, Index Coalescing now works in Indexed Assignment and Modified Indexed Assignment.

Implement Extended Replicate And Expand For Negative Integer Left Arguments

These two features were implemented some time ago, but I thought I'd mention them as **Mask** and **Mesh** depend upon them. For example,

```
        1 ¯2 1/1 2 3
1 0 0 3
       2 ¯1 1 2\1 2 3
1 1 0 2 3 3
```

Implement Full Support For NaNs, Remove From ⎕FEATURE

A NaN (Not-a-Number) is a concept from the IEEE-754 Floating Point Standard and is used as a fill element for an unknown value (among other uses). The symbol used is ⍬, where ⍬=⍬ is TRUE. This feature used to be temporary and was enabled by setting ⎕FEATURE[4]←1 – now it's fully supported.

Implement Power Operator Except For Inverses

I hesitated to mention this feature as I haven't implemented the most important part which is Inverses, but they'll happen. I did it once in the original NARS, I can do it again (if only I had more time).

Implement Conforming Disclose in ⊃R

This feature is actually needed in the Rank operator when merging the individual results back into a single array. Previously, each item in the Right Argument was required to be a scalar or have the same rank and shape as every other item. With this change, the items are all massaged first to have the same rank and shape, then they are joined together. For example,

```
      ⊃(1 2 3 4)(2 3⍴⍳6)(¯10 20)
 1  2  3  4
 0  0  0  0

 1  2  3  0
 4  5  6  0

10  0  0  0
20  0  0  0
```

Implement short left argument to dyadic Squad, UpArrow, and DownArrow

A handy shortcut from SHARP APL. For example,

```
      2⌷2 3⍴⍳6
4 5 6
      3↑2 3⍴⍳6
1 2 3
4 5 6
0 0 0
      1↓2 3⍴⍳6
4 5 6
```

Essentially, the Axis Operator [⍳≠⌿] is assumed.

Implement ?0 To Return A Random Number In [0, 1)

Another handy feature from Dyalog APL. For example,

```
      ⎕pp←5
      ?3⍴0
0.16404 0.51644 0.22442
```

Note that this primitive is sensitive to the value of ⎕DT.

Change name Of ⎕DQ to ⎕LR to reflect its wider usage

This System Variable was originally defined and named to provide access to the two possible Division Quotients when dividing non-commutative numbers (i.e., Quaternions and Octonions). Later, I expanded its usage considerably from the Division primitive function to the

- Circular
- Multiplication
- Residue
- Matrix Inverse/Divide
- Encode
- And
- Or

primitive functions, as well as the

- Matrix
- Inner Product

primitive operators.

These are all places where a programmer needs to choose between two possible (Left and Right) results.

Extend Variant Operator To Inner Product

As you may be aware, Inner Product has two identity elements: a Left and a Right one. For example, the Left identity element for a 2 by 3 matrix is a 2 by 2 identity matrix, whereas the Right identity element is a 3 by 3 identity matrix. Which identity element is returned (when needed) depends upon the setting of the System Variable ⎕LR. However, instead of setting this System Variable directly, you may use the Variant operator to put it into effect as in

```
      ⊃+.×⍤'l'/0⍴⊂2 3⍴⍳8
1 0
0 1
      ⊃+.×⍤'r'/0⍴⊂2 3⍴⍳8
1 0 0
0 1 0
0 0 1
```

In the process of implementing this feature, I rewrote the Variant operator code to be data-driven as there are now a great many cases (see above) where the Variant operator is used.

Implement Axis Operator For Condense and Dilate

These functions create and take apart Hypercomplex numbers. For example,

```
      <2 2⍴⍳4
1J2 3J4
      <[1]2 2⍴⍳4
1J3 2J4
      >⎕←<2 2⍴⍳4
1 2
3 4
      >⎕←<[1]2 2⍴⍳4
1J3 2J4
1 3
2 4
      >[1]⎕←<[1]2 2⍴⍳4
1J3 2J4
1 2
3 4
```

Change definition of ⎕CS to sort numbers before letters

This niladic function is a four-dimensional character array useful when sorting characters. The four dimensions are

- 10 for digits
- 6 for accented character sets
- 2 for upper/lower case
- 27 for the alphabets plus one for a separate column of digits

For example,

```
      ⍴⎕CS
10 6 2 27
      ⎕CS[;1;2;1]
0123456789
```

```
      ⎕CS[1;;1;]
abcdefghijklmnopqrstuvwxyz
ábćdéfǵhíjḱĺḿńóṕqŕśtúvẃxýź
àbcdèfgh̀ìjklmǹòpqrstùvẁxỳz
âbĉdêfĝĥîĵklmnôpqrstuvwxyz
äbcdëfgẗïjklmnöpqrsẗüvẅẍÿz
ãbcdẽfghĩjklmñõpqrstũṽwxỹz
```

This change moves the digits from the end of the alphabets (column 27) to the beginning (column 1).

Allow '∇' as argument to ⎕AT, ⎕CR, ⎕STOP, ⎕TRACE, ⎕VR

Inside a function (Anonymous or User-defined), you may use the character '∇' as an argument to the above System Functions to refer to the current active function without having to name it.

Allow ∘. As A Monadic Operator Including f←∘.

This change comes with Hyperators so as to allow it to be used as a Hyperand.

For example,

```
      f←∘.
      ⌊f⍨⍳4
1 1 1 1
1 2 2 2
1 2 3 3
1 2 3 4
```

Allow Scalar Right Arguments With Partitioned Enclose

For some reason I cannot justify, this used to produce an error, although the same scalar reshaped to a one-element vector worked just fine.

Before this change, this means that a solution to (say) the current APL Problem Solving Competition Phase I Problem 1 such as

```
fn←{(+\(≢⍵)⍴⍺↑1)⊂⍵}
```

fails on `4 fn 5` using the old definition (but works on `4 fn,5`) and needs to be written as

```
fn←{(+\(≢⍵)⍴⍺↑1)⊂1/⍵}
```

in order to handle the scalar Right Argument case. By supporting this case, the solution is simpler at no appreciable cost.

APL2 and Dyalog APL both signal an error.

Implement 4 ⎕DR to Extract Numerators and Denominators

This feature separates out the numerator and denominator of a Multiple-Precision Rational Number. I'm quite surprised that I haven't needed this feature before now.

For example,

```
      ⎕←a←3 3⍴(⍳9)÷2x
1r2   1 3r2
  2 5r2   3
7r2   4 9r2
      4 ⎕DR a
1 1 3
2 5 3
7 4 9

2 1 2
1 2 1
2 1 2|
      a≡÷≠4 ⎕DR a
1
```

Implement UTF-8/-16/-32 In Dyadic ⎕UCS

UTF-8 is pretty handy as in

```
      ,'%',¯2↑[2] ⎕←2 ⎕DR ⎕←'UTF-8' ⎕UCS '⎕'
226 142 149
0000000000000E2
00000000000008E
0000000000000095
%E2%8E%95
```

when I needed to include a '⎕' in the text of a link such as

When To Use ⎕CT With Rationals.

which translates the name part to

When%20To%20Use%20%E2%8E%95CT%20With%20Rationals.pdf

Promote Large Factorials From Error To RAT

For example, it used to be the case that !171 signalled an error because that was outside the limits of the GSL floating point library. Now it displays all 310 digits as a Multiple-Precision Integer/Rational number.

Remove Support For The Uppercase Underbar Latin Alphabet

Say goodbye.

Implement Variant For Grade Functions To Grade All Arrays

I know that Dyalog has defined the Grade primitives on all arrays, I just haven't had the chance to do the same. In the meantime, this uses an old definition of Grade on all arrays via ⍋⌸'a' R.

Allow Negative Sign In BasePoint

I have no idea why I missed this, but here it is:

```
      16b¯1FF
¯511
```

Define Identity Elements For Left/Right Tack Functions

For some reason, I thought this hole needed to be filled:

```
      ⊢/⍬
0
      ⊣/⍬
0
```

Demote dimension of Complex arrays of Eigenvalues, Eigenvectors, and Schur vectors

The expression ⌸⍤n M for a Matrix M and n∊1 2 3 4 produces an array of Eigenvalues, Eigenvectors, and/or Schur vectors all important concepts in Linear Algebra. This implementation is somewhat hindered by the fact that some of the libraries I'm using take input of Real arrays, not Complex arrays. If the Complex arrays have tiny Imaginary parts, they can be demoted to Real arrays, and that's what this change accomplishes.

I really need to write my own routines for Eigenvalues/vectors which would also have the very much needed side effect of allowing me to provide support for these concepts represented as Multiple-Precision FP and Ball Arithmetic.

Allow Accented Chars In ⎕EX,)ERASE, etc.

More of a bug fix than a feature.

Current & Future Work

Some of these ideas are leftover from the last time!

Java Support

Working with Dave Rabenhorst, we have spent considerable time creating a bridge between APL and the Java language where now we can call arbitrary Java code from APL. As a part of this interface, Dave has created a matching Session Manager with the capabilities of

the original Session Manager along with many extras.

Some of the features of the new SM include:

- distinguish different kinds of output with configurable color and style
- display numbers in colors to distinguish their sign, magnitude, and angle
- define any key to any glyph or string or action or combination
- define any function key to one or more immediate or delayed actions
- define any mouse click to an action
- hundreds of shift combinations are configurable for keys and clicks
- use a complementary configurable editor with enhanced new features
- nine different help windows are available any time
- direct calls from APL to functions in Java classes can arranged with the name association system function.
- over a dozen new built-in tools written in Java

Implement Inverses

The Power operator isn't the same without Inverses. This feature will also open up implementing the Dual and Commutator operators as well as Numerical Integration.

MP Floats: Set Precision As Lexical, Not Tokenized

This very subtle issue has caused me considerable pain when I was trying to track down a bug caused by it that appears at first to be far removed. The question revolves around the simple statements

```
□FPC←128
```

```
      ⎕FPC←256 ⋄ A←1.2v
      3 ⎕DR A
128
```

As you can see, the actual precision of `A` is **not** the expected value of `256`. The problem is that the precision of constants is a property you think is set at execution time, but is actually set at Tokenization time.

Finish Support for Matrix Operator and Eigenvalues/Eigenvectors

The current implementation is incomplete in that it doesn't handle non-diagonalizable matrices. Such matrices can be resolved using Jordan Canonical Form along with successive order derivatives of the function operand on the Eigenvalues.

However, now that I have a reliable and accurate Numerical Differentiation operator that works on all numeric datatypes, this goal is much closer.

Online Version

This paper is an ongoing effort and can be out-of-date the next day. To find the most recent version, go to http://sudleyplace.com/APL/ and look for the title of this paper on that page.

References

1. "Ball Arithmetic", wiki.nars2000.org/index.php/Ball_Arithmetic
2. "Hyperators", wiki.nars2000.org/index.php/Hyperators
3. John Scholes, San Quirico Moot, archive.vector.org.uk/art10011760
4. "Transform Hyperator", www.sudleyplace.com/APL/A%20Transform%20Hyperator%20in%20APL.pdf
5. "Numerical Differentiation in APL", www.sudleyplace.com/APL/Numerical%20Differentiation%20in%20APL.pdf

6. “Imaginary Number”, Wikipedia, https://en.wikipedia.org/wiki/Imaginary_number
7. "Nested Arrays System", 1981, STSC, Inc., Carl M. Cheney, p.35, 38
8. Iverson, Kenneth E., “A Programming Language”, John Wiley & Sons, Inc. New York, NY, USA ©1962 ISBN:0-471430-14-5
9. “Vector Notation”, http://wiki.nars2000.org/index.php/Binding_Strength#Vector_Notation