Progress In NARS2000
October 2011 to September 2013

Bob Smith
Sudley Place Software
4 Sep 2013

Released Features

Since the last Minnowbrook Conference, much has changed in NARS2000. First the past:

Index Of Primitive Function – `L⍳R`

The definition of Index Of (from the original 30+ year old NARS system) extended the APL1 definition to include arbitrary rank left arguments where the scalars of `R` are looked up in `L`, returning the first matching index in `L` (if `L` is a matrix, the result is an arrays of pairs, if `L` is a three-dimensional array, the result is an array of triples, etc.). The lookup part of this behavior mimics Member Of (its sister function) which also looks up the scalars of one argument in the other.

The Dyalog proposal to extend this function to any rank left arguments is based on an extension of Matrix Iota, where sub arrays in the right argument are matched against sub arrays in the left argument, and is defined to be entirely consistent with the APL1 definition.

Here's a table of rank, shape, and identities for the old and new definitions:

	APL1	NARS	Matrix Iota
Syntax	`L⍳R`	`L⍳R`	`L⍳⍤1 2 R`
Rank	`⍴⍴R`	`⍴⍴R`	`0⌈1+(⍴⍴R)-⍴⍴L`
Shape	`⍴R`	`⍴R`	`(1-⍴⍴L)↓⍴R`
Identity (no NotFound)	`R≡L[L⍳R]`	`R≡L[L⍳R]`	`R≡(⊂L⍳⍤1 2 R)[⍳≢L;⎕IO] L`

The NARS extension is more appealing because it retains the rank, shape and identities from the APL1 definition. The Matrix Iota extension is useful, but its new rank and shape rules make it look more like a different function which is why I assigned it to dyadic Iota Underbar.

Also, as implemented in NARS2000, Index Of works on scalar left arguments.

Factoring & Number-Theoretic Primitive Functions – `∏R` `L∏R`

The monadic version of this function factors the singleton right argument. At the moment, it

works well factoring (say) a 28-digit number into the product of two 14-digit primes, as in

```
      π×/□←1π?2ρ10*14x
58268447806519 71379419094611
58268447806519 71379419094611
```

The dyadic version, implements various related functions as seen in the following table:

Rth Probable Prime	¯2πR
Previous Probable Prime	¯1πR
Probable Prime Test	0πR
Next Probable Prime	1πR
Number of Probable Primes ≤R	2πR
Divisor Count	10πR
Divisor Sum	11πR
Mobius	12πR
Euler's Totient	13πR

So the first few primes may be generated from

```
      ⍳0π⍳100
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97
```

Sequence Primitive Function – L..R

This idea came from John Scholes of Dyalog. It starts with the simple (and origin-free) case of

```
      5..10
5 6 7 8 9 10
      5 2..10  ⍝ Step by 2
5 7 9
      (<5 2)..10   ⍝ Start at 5 2, continue through 10 10, nested
 5 2   5 3   5 4   5 5   5 6   5 7   5 8   5 9   5 10
 6 2   6 3   6 4   6 5   6 6   6 7   6 8   6 9   6 10
 7 2   7 3   7 4   7 5   7 6   7 7   7 8   7 9   7 10
 8 2   8 3   8 4   8 5   8 6   8 7   8 8   8 9   8 10
 9 2   9 3   9 4   9 5   9 6   9 7   9 8   9 9   9 10
 10 2  10 3  10 4  10 5  10 6  10 7  10 8  10 9  10 10
```

Where Primitive Function – ⍳R

This monadic primitive function is a generalization of the old Where function of R/⍳ρR, to (,R)/,⍳ρ1/R . For example,

```
      ⍸'Now is the time'=' '
4 7 11
      ⍸(5 5⍴⎕a)∈'AEIOU'
 1 1  1 5  2 4  3 5  5 1
```

Array Lookup Primitive Function – L⍸R

As mentioned above, this dyadic primitive function looks up the subarrays of the right argument in the left argument, as in Matrix Iota. It is assigned to the symbol Iota Underbar instead of Iota so as to retain the rank and shape rules of APL1's Index Of function. For example,

```
      L←5 3⍴⎕a ⋄ R←L[3 1 4;] ⋄ L R
 ABC  JKL
 DEF  DEF
 GHI  MNO
 JKL
 MNO
      L⍳R
 3 0  3 1  3 2
 1 0  1 1  1 2
 4 0  4 1  4 2
      L⍸R
3 1 4
```

Variant Primitive Operator – f⍠b R L f⍠b R

The dyadic Variant operator creates a version of the function `f` with a set of properties bound to it as specified by the array right operand. For example, `⍳⍠0` is a version of ⍳ that executes in origin-0 independent of the current value of `⎕IO`.

The array right operand to the operator may take several different forms depending upon the function left operand. In the general case, for any kind of function left operand, the right operand may specify a single property in the form of `Key Val`, or multiple properties in the form of `(Key1 Val1) (Key2 Val2) ...`, or if the left operand is a primitive function, a shorthand notation may be used as in `Val` or `Val1 Val2`.

In the general case, the keys specify a property such as Index Origin as `'IO'` or Comparison Tolerance as `'CT'`, and the value part specifies a value appropriate to the property specified in the matching key. At the moment, the keys that may be used are `'CT'`, `'DT'`, `'IO'`, and `'PP'` which stand for the corresponding system variable.

```
      ⍳⍠('IO' 0) 3
0 1 2
      ⍳⍠('IO' 1) 3
1 2 3
      ⍳⍠('IO' 0) ⍠('IO' 1) 3
0 1 2
      ⍳⍠('IO' 1) ⍠('IO' 0) 3
1 2 3
```

Note that in the last two examples, the lefthand (innermost) value of ⎕IO takes precedence. My understanding of the design of the same feature in Dyalog APL is that in their implementation, unless something has changed, the results of the last two examples would be reversed.

If the left operand is a primitive function, the following shorthands may be used:

⍳R and L⍳R	*N*	⎕IO←*N*
	N1 *N2*	⎕IO←*N1* and ⎕CT←*N2*
L⎕R, L⍜R, ⍋R, L⍋R, ⍒R, L⍒R, L⊂R, ⍸R, L⊓R	*N*	⎕IO←*N*
⍕R and L⍕R	*N*	⎕PP←*N*
L<R, L≤R, L=R, L≠R, L≥R, L>R, L∈R, L≡R, L≢R, L∪R, L∩R, L⊆R, L⊇R, L§R, L~R, L⍷R, L∣R ⌊R, ⌈R, ∪R	*N*	⎕CT←*N*
?R and L?R	*N*	⎕IO←*N*
	C	⎕DT←*C*
	N C	⎕IO←*N* and ⎕DT←*C*
	C N	⎕IO←*N* and ⎕DT←*C*

Native File System Functions – ⎕NTIE, etc.

Similar to other systems, but with explicit workspace-to-file and file-to-workspace conversions, for example

```
⎕NREAD tn ('int8' 'int64')    or    ⎕NREAD tn (812 6412)
```

reads 8-bit integers (`'int8'`) from the file and converts them to 64-bit integers (`'int64'`) in the workspace.

VFP Numbers

Notation change for VFP numbers is now suffix rather than infix: e.g., `12.3v` instead of `12v3`, etc.

Also, the implementation of VFP numbers uses MPFR exclusively, instead of the MPIR floating point functions as the latter functions are now deprecated.

Anonymous Functions & Operators (AFOs) – {...}

This feature is taken almost directly from the design of Dyalog's Direct Functions with a few exceptions:

1. Valences

An AFO may be

- Ambivalent (must be called with either one or two arguments),
- Dyadic (must be called with two arguments),
- Monadic (must be called with one argument), or
- Niladic (must be called with no arguments),

depending upon which of the special symbols are present in its definition.

Disregarding special symbols inside of character constants

- For anonymous functions (and operators):
  - If **⍺←** appears as a sequence of tokens, then the (derived) function is ambivalent,
  - Otherwise, if **⍺** appears as a token, the (derived) function is dyadic,
  - Otherwise, if **⍵** appears as a token, the (derived) function is monadic,
  - Otherwise, if neither **⍺** nor **⍵** appears as a token, the (derived) function is niladic.
- For anonymous operators:
  - If **⍵⍵** appears as a token, the operator is dyadic (must be called with two operands — left and right),
  - Otherwise, if **⍺⍺** appears as a token, the operator is monadic (must be called with one operand — left only).

One effect of these rules is that you never need supply a dummy argument to an AFO – you always use the exact number of arguments the function requires.

For the moment, the case of a niladic derived function from either a monadic or dyadic operator signals a **SYNTAX ERROR**.

For example,

```
      {⍵}2
2
      1{⍵}2
VALENCE ERROR
      1{⍵}2
        ^
      1{⍺+⍵}2
3
      {⍺+⍵}2
VALENCE ERROR
      {⍺+⍵}2
      ^
      {⍺←2 ⋄ ⍺+⍵}2  ⍝ Ambivalent function, monadic default left arg 2
4
      3{⍺←2 ⋄ ⍺+⍵}2 ⍝ Ambivalent function, dyadic with left arg 3
5
```

```
      {⍳3}           ⍝ A three-element simple vector
1 2 3
      {⍳3}23         ⍝ A two-element nested vector
 1 2 3  23
      12{⍳3}23       ⍝ A three-element nested vector
 12  1 2 3  23
      3 {∘.=⍨⍳⍺⍺}    ⍝ A niladic derived function from monadic operator
1 0 0
0 1 0
0 0 1
```

At the moment, the last example doesn't work – it's waiting for the 2by2 parser to be completely integrated into NARS2000, at which time niladic derived functions from user-defined operators will inherit the same behavior.

2. AFOs may be written on one line only.

3. Comments not allowed

4. Tail recursion not implemented as yet

5. Error guards not implemented

Keyboard Tables

US, UK, French, Danish, etc.

Execution Timer

A simple-minded, yet still useful instance of an execution timer is the calculation of the difference in times from one call to a function in immediate execution to when it returns.

This calculation is displayed in the status bar at the bottom of the program window.

Syntax Coloring

There is a new syntax coloring class for Point Notation separators (`beEprvx`): e.g., `123E4`, `123r45`, etc, that allows these separators to appear in a different color.

This helps simplify the game of Where's Waldo? when viewing output such as

```
      (?5⍴1000x)÷?5⍴100000x
163r10684 743r76689 51r5243 735r36767 669r65642
```

APL2 Array Spacing Rules

APL2 has a well-documented set of rules for the spacing between the rows and columns of a formatted array. These rules are clear and reasonable, although they are not the only such possible rules. They are however, clearly described in the APL2 Language Manual[1].

Display Of Wide Arrays

Sometime in the early 1980s when we were working on the ANSI X3J10 APL Standards committee, Larry Breed told me about a clever idea he had to display wide arrays: fold them at the page width in a group of rows rather than in each individual row. That is, the old way of displaying a wide array is

```
      ⎕pw←30
      3 20⍴⍳60
 1  2  3  4  5  6  7  8  9 10
      11 12 13 14 15 16 17 18
      19 20
21 22 23 24 25 26 27 28 29 30
      31 32 33 34 35 36 37 38
      39 40
41 42 43 44 45 46 47 48 49 50
      51 52 53 54 55 56 57 58
      59 60
```

and the new way is

```
      3 20⍴⍳60
 1  2  3  4  5  6  7  8  9 10
21 22 23 24 25 26 27 28 29 30
41 42 43 44 45 46 47 48 49 50

       11 12 13 14 15 16 17 18
       31 32 33 34 35 36 37 38
       51 52 53 54 55 56 57 58

       19 20
       39 40
       59 60
```

Update Processing

NARS2000 now supports automatic update detection and processing. The default setting is to check once a month. If a new update is found, a dialog asks if you want to download and install it.

Google Code

At the beginning, I hosted on my own site all the tools needed for version control, viewing source code, discussing the program on a forum, etc. Keeping all these tools up-to-date was a pain. Moreover, as I found out the hard way, if you aren't diligent, the bad guys will find holes in the tools, and compromise your web site. The solution was to move all the source code to an external service such as Google Code, and use an outside forum hosting service for discussion groups. What a pain!

Current & Future Work

2by2

This parser technique comes from the paper by Bunda and Gerth[2]. It is easy to modify to implement different syntax rules. I've copied the rules of APL2 with extensions to cover Trains, Anonymous Functions/Operators, Hybrid symbols (`/ ≠ ⍀ \`), special handling of niladic functions, etc.

Niladic Functions

In APL1, when a niladic function is encountered it is executed and the result (a variable) is passed on. That is, a niladic function is treated as a variable (sometimes with side effects). The new behavior allows one to assign a niladic function to another name as in `date←{3↑⎕TS}` without executing the function. Subsequent references to that name re-execute the function over and over, possibly with a different result each time. Only when the function is used in a context where a result is required (essentially, everything but assignment) is it executed.

Complex Numbers

The usual APL implementation of these numbers is with IEEE-754 double-precision floating point coefficients, however I also see a need for Rational and VFP coefficients, but not (64-bit) integers. From a design view, the datatype of the coefficient is irrelevant; only the implementation cares. The implementation of the VFP coefficient complex numbers is aided by the LGPL library MPC.

Quaternions and Octonions

These four- and eight-dimensional numbers are the next (and last) two steps up the chain after Real and Complex numbers.

From the Wikipedia article on quaternions, "..., quaternions are used in computer graphics, computer vision, robotics, control theory, signal processing, attitude control, physics, bioinformatics, molecular dynamics, computer simulations, and orbital mechanics. For example,

it is common for the attitude-control systems of spacecraft to be commanded in terms of quaternions. Quaternions have received another boost from number theory because of their relationships with the quadratic forms".

From the Wikipedia article on octonions, "Octonions are not as well known as the quaternions and complex numbers, which are much more widely studied and used. Despite this they have some interesting properties and are related to a number of exceptional structures in mathematics, among them the exceptional Lie groups. Additionally, octonions have applications in fields such as string theory, special relativity, and quantum logic".

Both of these numbers will be implemented with the same three coefficient datatypes as for complex numbers: 64-bit doubles, rational, and VFP.

These two number systems are also implemented in Sam Sirlin's APLc compiler[3].

Idioms

I have a fantasy that we can detect idioms using PCRE (Perl Compatible Regular Expressions). The idea is to use a technique I employ to analyze control structures (and later on will use to handle multi-line AFOs) over an entire program.

References

1. IBM, "APL2 Programming: Language Reference", Second Edition, February 1994, pp. 137-138.
2. "APL two by two-syntax analysis by pairwise reduction", Bunda, J.D. and Gerth, J.A., APL '84 Proceedings of the International Conference on APL
3. http://home.earthlink.net/~swsirlin/aplcc.html