Visual APL provides a large set of primitives which include both functions and operators. These are represented by symbols that specify which operations to perform in an expression. Visual APL predefines the usual arithmetic, data manipulation and logical functions and operators, as well as a variety of others as shown in the following table. In addition, many operators can be overloaded by the user, thus changing their meaning when applied to a user-defined type. There are two facilities provided to achieve this overloading, one is the using of a class with the appropriate attributes in place and the second is the overloading of .Net common operators, which can be overloaded in C# using the operator keyword. With the using keyword it is also possible to add functions and operators.

The primitive functions and operators provide support for all of the intrinsic .Net datatypes. As such, long, short, float, double, etc will be referred to as numeric. As there are a large number of intrinsic datatypes as well as Complex, IntN, BitArray, etc. not all types are included in the default array operator set. The default types are Int32, Double, and Char. However, scalar operations on all datatypes will work for the .Net base operator set.

In Visual APL, a function or operator is a term or a symbol that takes one or more expressions, called operands, as input and returns a value. Operators that take one operand, such as the increment operator (++), are called monadic or unary operators. Operators that take two operands, such as arithmetic operators (+,-,*,/) are called dyadic or binary operators. One operator.

The following Visual APL statement contains a single monadic operator, and a single operand. The increment operator, **++**, modifies the value of the operand `y`.:

```
Visual APL
```

```
y++;
```

The following Visual APL statement contains two dyadic operators, each with two operands. The assignment operator, **=**, has the integer `y`, and the expression `2 + 3` as operands. The expression `2 + 3` itself contains the addition operator, and uses the integer values `2` and `3` as operands:

```
Visual APL
```

```
y = 2 + 3;
```

An operand can be a valid expression of any size, composed of any number of other operations.
Operators in an expression are evaluated in a specific order, that is right to left. The following table divides the operators into categories based on the type of operation they perform.

Primary	x.y, f(x), a[x], x++, x--, new, typeof
Monadic (scalar and array)	+, -, !, ~, (T)x, ρ, ×, ÷, ι , ∈, ⌊, ⌈, ↑, ↓
Dyadic (scalar and array)	(,ravel) , !, ?, ⍋, ⍒, ⌽, ⍉, ⊂, ⊃, ⌽, ⍉, ⊖
Arithmetic --- Multiplicative (scalar and array)	×, ÷, |, ⍟, *, ○

Arithmetic --- Multiplicative (scalar)	%
Arithmetic --- Additive (scalar and array)	+, -
Shift (scalar)	<<, >>
Relational (scalar and array)	<, >, <=, >=, ≤ ≥
Type testing (scalar)	is, as
Equality (scalar and array)	==, ≈ ≡ ≠ ≊, ≣
Equality (scalar)	!=
Logical (scalar and array)	∧, ∨, ⍱, ⍲, ~
Logical (scalar)	&, ^, |
Data Analysis (scalar and array)	⍳, ∊, ⌈, ⌊, ⊥, ⊤, !, ?, ⍋, ⍒, ⍷, ⍎, ⍕
Data Manipulation (scalar and array)	⍴, ↑ ↓ (,catenate), ⍪, ⊂, ⊃, ⊖, ⌽, ⍉
Operator Functions (scalar and array)	**/, \\, [], (. dot),** ¨, ⌿, ⍀, ∘⍨
Conditional (Boolean)	&&, ||, then/else
Assignment	=, ← +=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, …

Dyadic Operators are evaluated from right to left, Monadic (unary) operators are evaluated from left to right.

```
Visual APL

num1 = 5;

num1++;

print num1
```

However, the output of the following example code is undefined:

```
Visual APL

num2 = 5;

num2 = num2++;  //not recommended

print num2
```

Therefore, the latter example is not recommended. Parentheses can be used to surround an expression and force that expression to be evaluated before any others. For example, 2 × 3 + 4 would normally become 14. This is because dyadic operators evaluate from right to left . Writing the expression as (2 × 3) + 4 results in 10, because it indicates to the Visual APL compiler that the multiplication operator (×) must be evaluated before the addition operator (**+**).

The Add function can act as either a monadic or dyadic primitive.

```
result ← expr1 + expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The dyadic + functions are predefined for numeric and string types. For numeric types, + computes the sum of its two operands. When one or both operands are of type string, + concatenates the string representations of the operands.

User-defined types can overload the dyadic + functions.

Example

```
function fn() {
    ⎕ ← 10 + 10
    ⎕ ← 10.5 + 10.5
    ⎕ ←  hello   +  world              "      "   "     "
    ⎕ ← 2j + 4j
    ⎕ ← 2.0 +   2                            "  "
}

    fn()
20
21
hello world
 6j
2 2

(only scalar Complex numbers are supported in this version)
```

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1     expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical AND of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    ⎕ ← 1 0 1 0 ∧ 1 0 1 0
    ⎕ ← 0 1 0 1 ∧ 0 0 0 0
    ⎕ ← 1 ∧ 1
    ⎕ ← 1 ∧ 0
    ⎕ ← 0 ∧ 0
    ⎕ ← 1 2 3 4 ∧ 4 3 2 1
}

      fn()
1 0 1 0
0 0 0 0
1
0
0
1 1 1 1
```

Specifies that *operatorexpr1* should apply its functionality across the dimension(s) specified in *axisexpr*

```
result ← expr1 operatorexpr1[ axisexpr ] expr2
result ← operatorexpr1[ axisexpr ] expr2
```

Where:

result
: An expression.

expr1
: An expression.

operatorexpr1
: An operator expression.

axisexpr
: An axis expression.

expr2
: An expression.

Remarks

The Axis operator provides a mechanism for applying the functionality of *operatorexpr1* to *expr1* and *expr2* across the dimension or dimensions specified by axisexpr.

axisexpr is a numeric vector.

Example

```
function fn() {
    ⎕ ←  apply a function across first "axis                              "
    ⎕ ← ⌽[0]3 3⍴⍳9
    ⎕ ←  apply a function across second"axis                              "
    ⎕ ← ⌽[1]3 3⍴⍳9
    ⎕ ←  apply a function with reductio"n across first axis                            "
    ⎕ ← +/[0]3 3⍴⍳9
    ⎕ ←  apply a function with reductio"n across second axis                            "
    ⎕ ← +/[1]3 3⍴⍳9
    ⎕ ←  apply a function with scan acr"oss first axis                            "
    ⎕ ← +\[0]3 3⍴⍳9
    ⎕ ←  apply a function with scan acr"oss second axis                            "
    ⎕ ← +\[1]3 3⍴⍳9
    ⎕ ←  apply a dyadic function across"first axis                            "
    ⎕ ← 1⌽[0]3 3⍴⍳9
    ⎕ ←  apply a dyadic function across"second axis                            "
    ⎕ ← 1⌽[1]3 3⍴⍳9
    ⎕ ←  apply a dyadic function with r"eduction across first axis                            "
    ⎕ ← 2+/[0]4 4⍴⍳16
    ⎕ ←  apply a dyadic function with r"eduction across second axis                            "
    ⎕ ← 2+/[1]4 3⍴⍳16
    ⎕ ←  apply a dyadic function with s"can across first axis                            "
    ⎕ ← 2+\[0]4 4⍴⍳16
    ⎕ ←  apply a dyadic function with s"can across second axis                            "
    ⎕ ← 2+\[1]4 4⍴⍳16
}

      fn()
apply a function across first axis
 6 7 8
 3 4 5
 0 1 2
apply a function across second axis
 2 1 0
 5 4 3
 8 7 6
apply a function with reduction across first axis
9 12 15
apply a function with reduction across second axis
3 12 21
apply a function with scan across first axis
 0  1  2
 3  5  7
 9 12 15
apply a function with scan across second axis
 0  1  3
 3  7 12
 6 13 21
```

```
apply a dyadic function across first axis
 3 4 5
 6 7 8
 0 1 2
apply a dyadic function across second axis
 1 2 0
 4 5 3
 7 8 6
apply a dyadic function with reduction across first axis
  4  6  8 10
 12 14 16 18
 20 22 24 26
apply a dyadic function with reduction across second axis
  1  3
  7  9
 13 15
 19 21
apply a dyadic function with scan across first axis
  4  6  8 10
 12 14 16 18
 20 22 24 26
apply a dyadic function with scan across second axis
  1  3  5
  9 11 13
 17 19 21
 25 27 29
```

Binomial !

Determines the number of groups of objects in the population represented by expr2 based on group size defined by expr1.

```
result ← expr1   expr2            !
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Binomial function supports positive arrays of numbers, and negative arrays of numbers.

Example

```
function fn() {
    ⎕ ← 2   10                !
    ⎕ ← 2 3 4   10 11 12          !
    ⎕ ← 2   ¯10               !
    ⎕ ← 2 3 4   ¯10 20 ¯30        !
}

      fn()
45
45 165 495
55
55 1140 40920
```

[] ***Operator***

Square brackets ([]) are used for arrays, indexers, attributes, and dynamic generic selection.

```
type[]
array[ indexexpr ]
generictype[ typeexpr ]
```

Where:

type
: A type.

array
: An array.

indexexpr
: An index expression.

generictype
: A generic type.

typeexpr
: A type expression.

Remarks

An array type is defined as a type followed by brackets:

```
int[] a = new int[10]
or dynamic
a = 0 0 0 0 0 0 0 0 0 0
```

To access an element of an array, the indices of the desired elements are enclosed in brackets after the expression:

Dependent state: ⎕IO

```
      a = 10 20 30
      a[0]
10
      a[0 1]
10 20
```

The array indexing operator cannot be overloaded; however, types can define indexers, properties that take one or more parameters. Indexer parameters are enclosed in square brackets, just like array indices, but indexer parameters can be declared to be of any type (unlike array indices, which must be integral).

Example

```
function fn() {
    a = 1 2 3 4 5
    ⎕ ← a[0 1 2]
    a = 3 3⍴⍳9
    ⎕ ← a[0 1; 0 1]
    ⎕ ← a[(0 1) (0 2)]
    a = Hashtable()
    a[ test ] = ⍳10                   "    "
    ⎕ ← a[ test ]                         "    "
    a = Dictionary[string, int]()
    a.Add( one , 10)                      "   "
    ⎕←a[ one ]                          "   "
}

      fn()
1
 0 1
 3 4
1 2
0 1 2 3 4 5 6 7 8 9
10
```

The Catenate function can act as either a monadic or dyadic primitive.

```
result ← expr1 , expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Catenates *expr1* with *expr2* along the last axis, unless another axis is provided.

Scalar expressions are expanded to conform with the non scalar expression.

Array expressions which differ by a rank of 1 are expanded to be conformable with the higher rank expression. Arrays must match in primary dimensions.

Example

```
function fn() {
    a = 1 2 3
    b = 4 5 6
    ⎕ ← a, b
    a =  test                          "     "
    b =  more                          "     "
    ⎕ ← a, b
    a = 3 3ρι9
    b = 3 4ρι12
    ⎕ ← a, b
    a = 10.4
    ⎕ ← a, b
    a =  test  10                      "     "
    b =  more  20                      "     "
    ⎕ ← a, b
}
      fn()
1 2 3 4 5 6
testmore
 0 1 2 0 1  2  3
 3 4 5 4 5  6  7
 6 7 8 8 9 10 11
 10.4 0 1  2  3
 10.4 4 5  6  7
 10.4 8 9 10 11
 test 10  more 20
```

Returns the smallest whole number greater than or equal to the specified number.

```
return ← ⌈ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Floor function returns the smallest whole number greater than or equal to a. If a is equal to NaN, NegativeInfinity, or PositiveInfinity, that value is returned.

The behavior of this function follows IEEE Standard 754, section 4. This kind of rounding is sometimes called rounding toward positive infinity.

Example

```
function fn() {
    ⎕ ← ⌈ 100.5
    ⎕ ← ⌈ 100.7
    ⎕ ← ⌈ 100.2
    ⎕ ← ⌈ 100.1 200.1 300.1
    ⎕ ← ⌈ 3 3⍴10.2
}
      fn()
101
101
101
101 201 301
 11 11 11
 11 11 11
 11 11 11
```

The Replicate function can act as either a monadic or dyadic primitive.

```
result ← expr1 / expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

expr1 must be a vector equal in length to the last dimension of expr2. If another axis is specified, then the length of expr1 must match the length of the specified dimension of expr2.

For values of 0 in expr1, elements in expr2 are removed. For positive integral elements in expr1, elements in expr2 are replicated integral times.

Example

```
function fn() {
    ⎕ ← 0 1 0 1 / 1 2 3 4
    ⎕ ← 0 1 0 1 / 4 4⍴⍳16
    ⎕ ← 1 2 3 4 / 1 2 3 4
}
      fn()
2 4
  1  3
  5  7
  9 11
 13 15
1 2 2 3 3 3 4 4 4 4
```

≡ *Depth*

The Depth function can act as either a monadic or dyadic primitive.

```
result ←≡expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

The Depth function determines the deepest level of nesting present in expr1.

Example

```
function fn() {
    a = 1
    ⎕←≡
    a = 1 2
    ⎕←≡
    a =  test  2                            "    "
    ⎕←≡
    a = ⊂⊂ 2 3
    ⎕←≡
}
      fn()
0
1
2
3
```

The Disclose function can act as either a monadic or dyadic primitive.

```
result ← ⊃ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Disclose function builds result from the elements of expr1.

If expr1 has only one (1) element, result is simply the contents of that first element. This simple case of Disclose is also known as un-nest, since it removes one (1) level of nesting from the data of expr1.

If expr1 contains two (2) or more elements, each element of expr1 is conformed such that every element of expr1 has the same rank and shape, and these elements are then structured into the result. The result is structured by concatenating together the conformed elements of expr1, and reshaping the result to be the shape of expr1 concatenated with the determined conformed shape applied to the elements of expr1.

The Disclose function is the inverse of the Enclose function.

Example

```
function fn() {
    a = 1
    ⎕ ← ⊃ a
    ⎕ ← ρ ⊃ a
    a = 1 2 3
    ⎕ ← ⊃ a
    ⎕ ← ρ ⊃ a
    a = ⊂⊂1 2 3
    ⎕ ← ⊃ a
    ⎕ ← ρ ⊃ a
    a = (1 2 3) 2 3
    ⎕ ← ⊃ a
    ⎕ ← ρ ⊃ a
    a = (3 3ρ1 2 3) 2 3
    ⎕ ← ⊃ a
    ⎕ ← ρ ⊃ a
}
      fn()
1

1 2 3
3
 1 2 3

 1 2 3
 2 0 0
 3 0 0
3 3
 1 2 3
 1 2 3
 1 2 3

 2 0 0
 0 0 0
 0 0 0

 3 0 0
 0 0 0
 0 0 0
3 3 3
```

The ÷ function can act as either a monadic or a dyadic primitive.

```
result ← expr1 ÷ expr2
```

Where:

result
An expression.
expr1
An expression.
expr2
An expression.

Remarks

The division operator (÷) divides its first operand by its second. All numeric types have predefined division operators.

```
Dependent state: ⎕DBZ, ⎕DBZV
```

The ⎕DBZ state variable provides control over the way in which divide addresses division by zero.

The default value is 0 to match .Net languages, however, you can set ⎕DBZ to the following:

```
⎕dbz:
 0 :  1÷0 = 0
      0÷0 = 0
 1 :  1÷0 = DOMAIN ERROR
      0÷0 = 1
 2 :  1÷0 = DOMAIN ERROR
      0÷0 = DOMAIN ERROR
 3 :  1÷0 = NaN or ⎕dbzv
      0÷0 = NaN or ⎕dbzv
 4 :  1÷0 = +-Infinity
      0÷0 = NaN
```

You can set ⎕DBZV to any object, and it will be returned when ⎕dbz is set to 3.

User-defined types can contain cross language overloads to the ÷ operator.

Example

```
function fn() {
    ⎕ ← 10 ÷ 20
    ⎕ ← 20 ÷ 10
    ⎕ ← 10 20 ÷ 20 10
    ⎕ ← 10.1 20.2 ÷ 10 20
    ⎕ ← 10 20 ÷ 10.1 20.1
    ⎕ ← (3 3⍴⍳9) ÷ 10
}

      fn()
0.5
2
0.5 2
1.01 1.01
0.9900990099 0.9950248756
   0 0.1 0.2
 0.3 0.4 0.5
 0.6 0.7 0.8
```

The Drop function can act as either a monadic or dyadic primitive.

```
result ← expr1 ↓ expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

The Drop function removes data from dimensions of expr2, according to the amounts specified in expr1.

The length of expr1 should match the rank of expr2, and each element of expr1 specifies the amount of data to drop from the respective dimension of expr2.

The elements of expr1 can be either negative, positive, or 0. If an element of expr1 is positive, that length is dropped from the related dimension of expr2. If an element of expr1 is negative, that length is dropped from opposite end of the related dimension of expr2. If an element of expr1 is 0, no data is dropped from the related dimension of expr2.

Example

```
function fn() {
    ⎕ ← 1 ↓ 10 11 12
    ⎕ ← 3 ↓ 10 11 12 13 14 15
    ⎕ ← ¯1 ↓ 10 11 12
    ⎕ ← ¯3 ↓ 10 11 12 13 14 15
    ⎕ ← 2 2 ↓ 3 3⍴⍳9
    ⎕ ← ¯2 ¯2 ↓ 3 3⍴⍳9
}
      fn()
11 12
13 14 15
10 11
10 11 12
 8
 0
```

Performs the specified operator expression across each element of expr1 and expr2.

```
result ← expr1 operator¨ expr2
```

Where:

result
An expression.

expr1
An expression.

operator
An operator expression.

expr2
An expression.

Remarks

The Each operator is a specialized short hand construct simulating a single for loop across the elements of *expr2*.

The Each data iterator performs the specified operator expression between each element of expr1 and expr2. If expr1 or expr2 is a scalar, that expression is considered to be the same rank and shape of the higher rank expression.

Example

```
function fn() {
    ⎕ ← 2 ρ ¨ 1 2 3
    ⎕ ← (⊂2 2) ρ ¨ 1 2 3
    ⎕ ← (⊂2 2) ρ ¨ (1 2) (ι4)
    ⎕ ← (⊂1 2 3) + ¨ (1 2 3) (10 20 30)
    ⎕ ← (⊂⊂1 2 3) + ¨¨ (1 2 3) (10 20 30)
    ⎕ ← 3 ι ¨ (1 2 3) (4 5 6)
    ⎕ ← (⊂2 3) ι ¨ (2 3) (4 5) (5 6)
    ⎕ ← ι ¨ 1 2 3
    ⎕ ← ρ ¨ (1 2) (3 4 5) (3 3ρι9)
}
      fn()
 1 1  2 2  3 3
  1 1   2 2   3 3
  1 1   2 2   3 3
  1 2   0 1
  1 2   2 3
 2 4 6  11 22 33
  2 3 4  3 4 5  4 5 6   11 12 13  21 22 23  31 32 33
 1 1 0  1 1 1
 0 1  2 2  2 2
 0  0 1  0 1 2
 2  3  3 3
```

The Enclose function can act as either a monadic or dyadic primitive.

```
result ←⊂expr1
```

Where:

result
: An expression.

expr1
: An expression.

Remarks

The Enclose function creates *result* by nesting *expr1* once.

The only exception to this enclosure rule is if expr1 is a native .Net type scalar, such as Int32, Double, or Char. If expr1 is a .Net native type scalar, the data is not enclosed, and *result* is exactly equal to *expr1.*

The Enclose function is the inverse of the Disclose function.

Example

```
function fn() {
    a = ⊂
    ⎕← shape of enclosed scalar      "                         "
    ⎕←ρ
    a = ⊂ 2 3
    ⎕← shape of enclosed vector      "                         "
    ⎕←ρ
    a = ⊂1 2 3) (5 6 7)
    ⎕← shape of enclosed vector of ve"ctors                               "
    ⎕←ρ
    a = ⊂(1 2 3) (4 5 6)
    ⎕← shape of enclose of each vector"                           "
    ⎕←ρ
    ⎕← shape of each enclosed vector "                           "
    ⎕←ρa
    a = ⊂⊂ 2 3
    ⎕← shape of the original vector us"ing each                             "
    ⎕←ρ¨a
}
      fn()
shape of enclosed scalar

shape of enclosed vector

shape of enclosed vector of vectors

shape of enclose of each vector
2
shape of each enclosed vector

shape of the original vector using each
  3
```

The Enlist function can act as either a monadic or dyadic primitive.

```
result ← ∊ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Enlist function produces a flattened version of *expr1. result* contains all data which was present in *expr1* and its sub elements, with all nesting, shape, and rank removed, so that *result* is a simple vector.

Example

```
function fn() {
    a = ∊1
    ⎕ ← a
    ⎕ ← ρa
    a = ∊1 2 3
    ⎕ ← a
    ⎕ ← ρa
    a = ∊3 3ρι9
    ⎕ ← a
    ⎕ ← ρa
    a = ∊(1 2 3) (4 5 6)
    ⎕ ← a
    ⎕ ← ρa
    a = ∊(⊂⊂⊂1 2 3) (⊂⊂ test )                         "    "
    ⎕ ← a
    ⎕ ← ρa
}
      fn()
1
1
1 2 3
3
0 1 2 3 4 5 6 7 8
9
1 2 3 4 5 6
6
 1 2 3 test
7
```

The Approximately Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≈ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state: ⎕CT`

The Approximately Equal function returns a 1 if *expr1* is equal to *expr2*, or if *expr2* is within ⎕CT of *expr1*. Otherwise, the return is 0.

Example

```
function fn() {
    ⎕ ← 10 ≈ 12
    ⎕ ← 10 ≈ 9 10 11
    ⎕ ← 10 ≈ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≈ 1 2 3
    ⎕ ← 1 2 3 ≈ 1+1 2 3
    ⎕ ← 1 2 3 ≈ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≈ 3 3⍴10 11
}
      fn()
0
0 1 0
 0 0 0
 0 0 1
 0 0 0
1 1 1
0 0 0
0 0 0
 0 0 0
 0 0 0
 0 0 0
```

⍎ ***Execute***

Executes the code supplied by expr1

```
result ← ⍎ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Execute expression dynamically executes the code returned by *expr1*. *expr1* can return either a string, a dynamic variable (IVariable), or a compiled code object (obtainable through the compile method). If *expr1* evaluates to a string, then the code is parsed, compiled, and then executed. If *expr1* is a compiled code object, no parsing and compilation is required, and the code object is executed immediately.

Note: Language features which effect the code flow of a function do not effect the function which initiated the dynamic execution. Examples of these kinds of statements include yield, return, break, continue, branching, and conditional branching. Such statements can be used within the respective constructs to which they apply, such as a yield statement within a function defined in the same dynamic execution.

Advanced Dynamic Execution Features:

Dynamic execution allows you to override the module dictionaries used within the context of the dynamic execution. Using this feature, you can specify either or both of the local variable and global variable dictionaries, which enables the dynamic execution of code within contexts other than the context of the function which called the dynamic execution. You can even create entirely new contexts under program control just for the purpose of dynamically executing code.

The following example calls dynamic execute and specifies that only "a" and "b" are to be used in the local dictionary of the execution:

```
      a = 10
      b = 20 30 40
      c = ⍎ a+b  in (a,b)                    "   "
      c
30 40 50
```

Depending on where an execute statement is programmed in your code, you will have access to either or both of the global dictionaries *ws* and *wsi*. The field *ws* contains all static data which exists in the current context of where you reference *ws*, and *wsi* contains all instance data for the context it which it is referenced.

In functions which are defined with the *static* access modifier, only the *ws* field will be accessible, because by definition no instance data can be referenced from a *static* method. In an instance method, or any method which does not exist in a static class or has the *static* modifier applied to its definition, you also have access to the global field *wsi*.

By default, when you run a dynamic execution and do not specify the global context in which it will run, the *wsi* (or *ws* for static methods) is passed as the default global dictionary.

Dynamically defining contexts:

You can dynamically create a global context under program control, which can be used in place of the default *ws* or *wsi* global fields.

Here is an example of creating a module dictionary which contains a single element "alist"*.* Once the dictionary is created and initialized, the dictionary is then passed to execute as the global dictionary:

Note: Any object which inherits from IDictionary can be used as a global dictionary.

```
      using System
      using System.Collections
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      a = ArrayList()
      a.Add(10)
0
      a.Add( test )                            "    "
```

```
1
      a.Count
2
      gd.Add( alist ,a)                     "     "
      ⍎  alist.Add('more')  in (),gd   "                 "
false
      a.Count
3
```

As you can see above, the variable "alist" does not exist in the context in which the execute is run, and only exists as an entry in the newly created Dictionary object which was passed to the execute statement. Using this methodology, you can dynamically create any arbitrary context in which to run your dynamic execution.

Dynamic Evaluation:

All code which is processed by dynamic execute is fully compiled to the lowest possible level in .Net, which allows the code to run as fast as any code compiled at runtime. In some cases, the code statement to be run by execute may be small enough that the extra time required to compile the code would be unnecessary, and in these cases it may be optimal to interpret the code directly.

To directly interpret a code snippet, use the eval statement. Here is the above example for execute, modified to instead use the eval method:

```
      using System
      using System.Collections
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      a = ArrayList()
      a.Add(10)
0
      a.Add( test )                        "    "
1
      a.Count
2
      gd.Add( alist ,a)                     "     "
      eval( alist.Add('more') , null, gd)  "                 "
false
      a.Count
3
```

The performance gain of directly interpreting code is only found when evaluating small and simple snippets of code. While fully supported, snippets which include statements such as *for* or *while* loops would not be normally appropriate, because the iteration process re-evaluates each line of code as it is run in the for loop, and is therefore not as highly optimized as direct compilation.

Example

```
      a = 10
      b = 20 30 40
      ⍎ a+b                           "   "
30 40 50
      c = ⍎ a+b                           "   "
      c
30 40 50
      c = ⍎ a+b  in (a,b)                 "   "
      c
30 40 50
      using System.Collections.Generic
      gd = Dictionary[object, object]()
      x
name 'x' is not defined
      c = ⍎ x = a+b  in (a,b),gd          "       "
      x
name 'x' is not defined
      gd[ x ]                           " "
30 40 50
      using System.Collections
      h = Hashtable()
      gd[ newhash ] = h                 "       "
      h.Count
0
```

```
      c = ⍎ newhash.Add(\ one\ ,100.9)  in"(),gd          "     "        "
      gd[ newhash ].Count                  "        "
1
      h.Count
1
      h[ one ]                            "   "
100.9
      e = compile( a = b+c ,ws)                     "       "
      b = 10
      c = 100
      ⍎e in (b,c)
110
      a
110
      ⍎e
110
```

The Expand function can act as either a monadic or dyadic primitive.

```
result ← expr1  \ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The length of the *result* is determined by the length of *expr1.*

expr1 is a numeric vector, where at every non zero (0) element the next element of *expr2* will be inserted into the result. Where a zero (0) occurs in *expr1*, the fill data element for *expr2* is inserted into the *result* instead.

Example

```
function fn() {
    ⎕ ← 1  0  1  0  1  \ 1  2  3
    ⎕ ← 1  0  1  0  1  \ 3  3ρι9
    ⎕ ← 1  0  1  \  test  (1  2  3)                  "    "
    ⎕ ← 1  0  0  1  1  \ 3  3ρι9
}
      fn()
1  0  2  0  3
 0  0  1  0  2
 3  0  4  0  5
 6  0  7  0  8
 test        1  2  3
 0  0  0  1  2
 3  0  0  4  5
 6  0  0  7  8
```

The Exponential function can act as either a monadic or dyadic primitive.

```
result ← ⋆ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Exponential function expands the Math.Exp method to work with numeric arrays.

Math.Exp returns the number **e** raised to the power *expr1*. If *expr1* equals *NaN* or *PositiveInfinity*, that value is returned. If *expr1* equals *NegativeInfinity*, 0 is returned.

Example

```
function fn() {
    ⎕ ← ⋆0
    ⎕ ← ⋆1
    ⎕ ← ⋆2
    ⎕ ← ⋆3
}

      fn()
1
2.718281828
7.389056099
20.08553692
```

The Factorial function can act as either a monadic or dyadic primitive.

```
result ←   expr1            !
```

Where:

result
: An expression.

expr1
: An expression.

Remarks

The Factorial function determines the mathematical factorial of *expr1*. For non integral *expr1*, the standard mathematical procedure of determining the factorial result through the *Gamma* function is applied.

Example

```
function fn() {
    ⎕ ←  1                  !
    ⎕ ←  2                  !
    ⎕ ←  3                  !
    ⎕ ←  4                  !
    ⎕ ←  1 2 3 4            !
}
      fn()
1
2
6
24
1 2 6 24
```

The Find function can act as either a monadic or dyadic primitive.

```
result ← expr1    expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Find function returns an integer array of the same shape and rank as *expr2*, with a one (1) wherever the array *expr1* was found in *expr2*. *expr1* and *expr2* can be arrays of any shape, rank, and depth.

```
Dependent state:  CT
```

Example

```
function fn() {
    ⎕ ← 1 2 3 ∊ 1 2 3 4 1 2 3
    ⎕ ← 0 1 2 ∊ 3 3ρ⍳9
    ⎕ ←  what  ∊  morewhatofwhat        "    "   "              "
    ⎕ ←  hey  ∊ 4 3ρ heyyouheyyou       "   "       "            "
}
      fn()
1 0 0 0 1 0 0
 1 0 0
 0 0 0
 0 0 0
0 0 0 0 1 0 0 0 0 0 1 0 0 0
 1 0 0
 0 0 0
 1 0 0
 0 0 0
```

The First function can act as either a monadic or dyadic primitive.

```
result ← ↑ expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

The First function returns the first element of *expr1,* disclosing the element if it is enclosed.

The First function is a short hand for accessing the first element of an array.

Example

```
function fn() {
    ⎕ ← ↑ 1 2 3
    ⎕ ← ↑ 3 3ρι9
    ⎕ ← ↑ 3 3 3ρι27
    ⎕ ← ↑  test                          "    "
    ⎕ ← ↑ 4 4ρ test                          "    "
}
      fn()
1
0
0
t
t
```

⌊ ***Floor***

The Floor function can act as either a monadic or dyadic primitive.

```
result ← ⌊ expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

```
Dependent state:  CT
```

The Floor function returns the largest whole number less than or equal to *expr1*. If *expr1* is equal to *NaN*, *NegativeInfinity*, or *PositiveInfinity*, then that value is returned.

The behavior of this function follows IEEE Standard 754, section 4. This kind of rounding is sometimes called rounding toward negative infinity.

Note: The Floor function uses CT when determining if *expr1* is already equal to an integral value. If *expr1* is within CT of the next greater whole number, than the Floor function does not apply. Instead, Floor assumes that if *expr1* cannot be rounded to the next lesser whole number, than it must match the next greatest whole number, and the next greatest whole number is returned. This guarantees that only integers will return from the Floor function.

Example

```
function fn() {
    ⎕ ← ⌊ 1.1
    ⎕ ← ⌊ 1.5
    ⎕ ← ⌊ 1.8
    ⎕ ← ⌊ 1.1 1.5 1.8
    ⎕ ← ⌊ 3 3⍴10.1 11.1 12.1
}

      fn()
1
1
1
1 1 1
 10 11 12
 10 11 12
 10 11 12
```

The Format function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍕ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state:   NFI,` PP

The Format function Creates

Simple formatter that provides simple width control and converts objects to their string representation. Relies on ⎕nfi

```
⍕2 3⍴⍳6
 0 1 2
 3 4 5
      (2 3⍴⍳6).ToString()
 0 1 2
 3 4 5
```

The ToString method in most cases is equivalent.

```
      1 0 4 1 6 2 ⍕ 2 3⍴⍳6
0 1.0  2.00
3 4.0  5.00
```

Notice that the width of each column was controlled by the left argument. The left argument is composed of value pairs, width and number of decimals.
Using a negative value for number of decimals formats objects in Exponential.

```
      10 ¯5 ⍕10 20 30 999.4
  1.0000E1  2.0000E1  3.0000E1  9.9940E2
```

Example

```
function fn() {
    ⎕ ← ⍕1 2 3
    ⎕ ← 3 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 2 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 ¯2 ⍕ 1.2 2.3 3.4
    ⎕ ← 7 2 ⍕ 3 3⍴1.2 2.3 3.4
    ⎕ ← 1 0 6 2 7 3 ⍕ 2 3⍴1 2 3
}
      fn()
1 2 3
1.200 2.300 3.400
   1.20   2.30   3.40
  1.2E0  2.3E0  3.4E0
   1.20   2.30   3.40
   1.20   2.30   3.40
   1.20   2.30   3.40
1  2.00  3.000
1  2.00  3.000
```

Produces a vector of numbers, which is the representation of expr2 with radix specifications expr1.

```
result ← expr1 ⊤ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

From Base 10 (Encode) is the inverse function of To Base 10 (Decode)

Example

```
function fn() {
    ⎕ ← 10 10 10 10 ⊤ 1776
    ⎕ ←  Convert 3622 minutes to 2 days," 12 hours, 22 minutes                                    "
    ⎕ ← 0 24 60 ⊤ 3622
    ⎕ ←  Convert 10 to 8 bits           "                     "
    ⎕ ← 2 2 2 2 2 2 2 2 ⊤ 10
}
      fn()
1 7 7 6
Convert 3622 minutes to 2 days, 12 hours, 22 minutes
2 12 22
Convert 10 to 8 bits
0 0 0 0 1 0 1 0
```

The Grade Down function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍒ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state: ⎕I O`

The Grade Down function returns an integer array of indices which specify the sorted order of *expr2,* **in descending order***,* according to either the order of *expr1* if it is supplied, or the IComparable interface implemented by the argument data in *expr2*.

The Grade functions extend the Microsoft Array.Sort method to work with arrays of all rank and depth.

The Microsoft Array.Sort method performs a highly optimized, unstable Q-Sort on the elements of vectors to be sorted, using the IComparable interface implemented by each element of the array being sorted for determining if one value is greater than another.

The Grade functions extend Array.Sort to function on arrays in general, and also stabilize the result so that elements which are considered equal appear in the result in the same order that they appeared in *expr2*. Also, if *expr2* is all of a single type, only one comparitor is utilized, further optimizing the sorting process.

If *expr1* is supplied, a custom comparitor is created which sorts the elements of *expr2* according to the order of their appearance in *expr1*. If an element of *expr2* does not exist in *expr1*, that element is considered to have the least importance in the sorting process, and will appear after all other elements in the result which did exist in *expr1*. Of course, all elements of the result which do not appear in *expr1* are stabilized as the sort progresses, and appear in the order in which they occurred in *expr2*.

Note that the result might vary depending on the current CultureInfo.

Note: The IComparable interface defines a generalized comparison method that a value type or class implements to create a type-specific comparison method. Visit the Microsoft web site to see examples of how to implement the IComparable interface on your Visual APL classes.

Example

```
function fn1() {
    a = 50 40 30 20 10
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 10 20 30 40 50
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 3 3⍴⍳9
    ⎕ ← ⍒a[;0]
    ⎕ ← a[⍒a[;0];]
    a =  abcde                         "     "
    ⎕ ← ⍒a
    ⎕ ← a[⍒a]
    a = 3 3⍴ abcdefghi                     "         "
    ⎕ ← ⍒a
    ⎕ ← a[⍒a;]
}
      fn1()
0 1 2 3 4
50 40 30 20 10
4 3 2 1 0
50 40 30 20 10
2 1 0
 6 7 8
 3 4 5
 0 1 2
4 3 2 1 0
edcba
```

```
2 1 0
ghi
def
abc
0 1 2
 1 2 3  2 3 4  3 4 5



function fn2() {
    a =  abcde                         "     "
    c =  edcba                         "     "
    ⎕ ← c ⍒ a
    ⎕ ← a[c⍒a]
    a = 1 2 3 4 5
    c = 5 4 3 2 1
    ⎕ ← c ⍒ a
    ⎕ ← a[c⍒a]
    a = 3 3⍴⍳9
    c = 9 8 7 6 5 4 3 2 1 0
    ⎕ ← c ⍒ a
    ⎕ ← a[c⍒a;]
    a = (1 2 3) ( test ) (3 4 5)                "    "
    c = (3 4 5) ( test ) (1 2 3)                "    "
    ⎕ ← c ⍒ a
    ⎕ ← a[c⍒a]
}

      fn2()
0 1 2 3 4
abcde
0 1 2 3 4
1 2 3 4 5
0 1 2
 0 1 2
 3 4 5
 6 7 8
0 1 2
 1 2 3  test  3 4 5
```

The Grade Up function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍋ expr2
```

Where:

result
An expression.
expr1
An expression.
expr2
An expression.

Remarks

```
Dependent state:  ⎕IO
```

The Grade Up function returns an integer array of indices which specify the sorted order of *expr2,* **in ascending order***,* according to either the order of *expr1* if it is supplied, or the IComparable interface implemented by the argument data in *expr2*.

The Grade functions extend the Microsoft Array.Sort method to work with arrays of all rank and depth.

The Microsoft Array.Sort method performs a highly optimized, unstable Q-Sort on the elements of vectors to be sorted, using the IComparable interface implemented by each element of the array being sorted for determining if one value is greater than another.

The Grade functions extend Array.Sort to function on arrays in general, and also stabilize the result so that elements which are considered equal appear in the result in the same order that they appeared in *expr2*. Also, if *expr2* is all of a single type, only one comparitor is utilized, further optimizing the sorting process.

If *expr1* is supplied, a custom comparitor is created which sorts the elements of *expr2* according to the order of their appearance in *expr1*. If an element of *expr2* does not exist in *expr1*, that element is considered to have the least importance in the sorting process, and will appear after all other elements in the result which did exist in *expr1*. Of course, all elements of the result which do not appear in *expr1* are stabilized as the sort progresses, and appear in the order in which they occurred in *expr2*.

Note that the result might vary depending on the current CultureInfo.

Note: The IComparable interface defines a generalized comparison method that a value type or class implements to create a type-specific comparison method. Visit the Microsoft web site to see examples of how to implement the IComparable interface on your Visual APL classes.

Example

```
function fn1() {
    a = 50 40 30 20 10
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 10 20 30 40 50
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 3 3⍴⍳9
    ⎕ ← ⍋a[;0]
    ⎕ ← a[⍋a[;0];]
    a =  abcde                    "     "
    ⎕ ← ⍋a
    ⎕ ← a[⍋a]
    a = 3 3⍴ abcdefghi                  "         "
    ⎕ ← ⍋a
    ⎕ ← a[⍋a;]
}
      fn1()
4 3 2 1 0
10 20 30 40 50
0 1 2 3 4
10 20 30 40 50
0 1 2
 0 1 2
 3 4 5
 6 7 8
0 1 2 3 4
```

```
abcde
0 1 2
abc
def
ghi


function fn2() {
    a =  abcde                    "     "
    c =  edcba                    "     "
    ⎕ ← c ⍋ a
    ⎕ ← a[c⍋a]
    a = 1 2 3 4 5
    c = 5 4 3 2 1
    ⎕ ← c ⍋ a
    ⎕ ← a[c⍋a]
    a = 3 3⍴⍳9
    c = 9 8 7 6 5 4 3 2 1 0
    ⎕ ← c ⍋ a
    ⎕ ← a[c⍋a;]
    a = (1 2 3) ( test ) (3 4 5)               "    "
    c = (3 4 5) ( test ) (1 2 3)               "    "
    ⎕ ← c ⍋ a
    ⎕ ← a[c⍋a]
}
      fn2()
4 3 2 1 0
edcba
4 3 2 1 0
5 4 3 2 1
2 1 0
 6 7 8
 3 4 5
 0 1 2
2 1 0
 3 4 5  test  1 2 3
```

The Greater Than function can act as either a monadic or dyadic primitive.

```
result ← expr1 > expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

`Dependent state:  ⎕CT`

The Greater Than function returns 1 if *expr1* is greater than *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "greater than" relational operator.

User-defined types can contain cross language overloads to the > operator.

Example

```
function fn() {
    ⎕ ← 10 > 12
    ⎕ ← 10 > 9 10 11
    ⎕ ← 10 > 5+3 3⍴⍳9
    ⎕ ← 1 2 3 > 1 2 3
    ⎕ ← 1 2 3 > 1+1 2 3
    ⎕ ← 1 2 3 > 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) > 3 3⍴10 11
}
      fn()
0
1 0 0
 1 1 1
 1 1 0
 0 0 0
0 0 0
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

The Greater Than or Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≥ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Greater Than or Equal function returns 1 if *expr1* is greater than, or equal to, *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "greater than or equal" relational operator.

User-defined types can contain cross language overloads to the ≥ operator. If ≥ is overloaded, ≤ must also be overloaded.

Example

```
function fn() {
    ⎕ ← 10 ≥ 12
    ⎕ ← 10 ≥ 9 10 11
    ⎕ ← 10 ≥ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≥ 1 2 3
    ⎕ ← 1 2 3 ≥ 1+1 2 3
    ⎕ ← 1 2 3 ≥ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≥ 3 3⍴10 11
}
      fn()
0
1 1 0
 1 1 1
 1 1 1
 0 0 0
1 1 1
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

The IndexOf function can act as either a monadic or dyadic primitive.

```
result ← expr1 ɩ expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

`Dependent state: ⎕IO, ⎕CT`

The IndexOf function returns the index of the first occurrence of expr1 in expr2. If expr2 does not contain expr1, the returned index is one plus the number of elements in expr2.

The IndexOf function is similar in use to the IndexOf method found on many objects in .Net, with the exception of returning one plus the number of elements in the argument data, instead of a -1.

Example

```
function fn() {
    ⎕ ←  hello world  ɩ  hello world   "            "   "            "
    ⎕ ← 1 2 3 ɩ 10 20 30 1 40 2 50 3 1 2 3
    ⎕ ← (ɩ10) ɩ 1 4 20
    ⎕ ← 0 1 2 3 ɩ 3 3ρɩ9
    ⎕ ← 1 ɩ 3 2 1 3 2 1
}
      fn()
0 1 2 2 4 5 6 4 8 2 10
3 3 3 0 3 1 3 2 0 1 2
1 4 10
 0 1 2
 3 4 4
 4 4 4
1 1 0 1 1 0
```

The Inner Product function can act as either a monadic or dyadic primitive.

```
result ← expr1 operatorexpr1 . operatorexpr2 expr2
```

Where:

result
An expression.

expr1
An expression.

operatorexpr1
An operator expression.

operatorexpr2
An operator expression.

expr2
An expression.

Remarks

The Inner Product function is a specialized short hand construct for successively calling operators in a pre defined order.

The Inner Product function creates its result by first calling the function specified by *operatorexpr2* as though that function had been called dyadically with *expr1* and *expr2*, and then takes the result of that operation, and uses it as the right operand to the reduce version of *operatorexpr1*.

Example

```
function fn() {
    ⎕ ← (3 3ρι9) ∧.≈ 0 1 2
    ⎕ ← (3 5ρ hellowhatsupdoc ) ∧.≈  whats  "                "      "     "
    ⎕ ← 1 2 3 +.× 1 2 3
    ⎕ ← 10+.×(1 2 3) (4 5 6)
}
      fn()
1 0 0
0 1 0
14
 50 70 90
```

The Interval function can act as either a monadic or dyadic primitive.

```
result ← ⍳ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

```
Dependent state:  I O
```

The Interval function produces an integer vector from one (1) to *expr1*, or if `I O` is zero (0), from zero (0) to (*expr1* - 1).

Example

```
function fn() {
    ⎕ ← ⍳10
    ⎕ ← 1+⍳10
    ⎕ ← 3+3×⍳10
}
      fn()
0 1 2 3 4 5 6 7 8 9
1 2 3 4 5 6 7 8 9 10
3 6 9 12 15 18 21 24 27 30
```

The Laminate function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍪ expr2
```

Where:

result
An expression.
expr1
An expression.
expr2
An expression.

Remarks

`Implicit argument: ⎕IO`

Catenates *expr1* with *expr2* along the first axis, unless another axis is provided.

Scalar expressions are expanded to conform with the non scalar expression.

Array expressions which differ by a rank of 1 are expanded to be conformable with the higher rank expression. Arrays must match in primary dimensions.

Example

```
function fn() {
    a ← 1 2 3 ⍪ 1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 2 3 ⍪ 1 3⍴1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 2 3 ⍪ 3 3⍴⍳9
    ⎕ ← a
    ⎕ ← ⍴a
    a ← 1 ⍪ 1 3⍴1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a ←  abc  ⍪ 2 3⍴ efghij             "   "        "       "
    ⎕ ← a
    ⎕ ← ⍴a
}
      fn()
1 2 3 1 2 3
6
 1 2 3
 1 2 3
2 3
 1 2 3
 0 1 2
 3 4 5
 6 7 8
4 3
 1 1 1
 1 2 3
2 3
abc
efg
hij
3 3
```

The Less Than function can act as either a monadic or dyadic primitive.

```
result ← expr1 < expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

`Dependent state:  ⎕CT`

The Less Than function returns 1 if *expr1* is less than *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "less than" relational operator.

User-defined types can contain cross language overloads to the < operator.

Example

```
function fn() {
    ⎕ ← 10 < 12
    ⎕ ← 10 < 9 10 11
    ⎕ ← 10 < 5+3 3⍴⍳9
    ⎕ ← 1 2 3 < 1 2 3
    ⎕ ← 1 2 3 < 1+1 2 3
    ⎕ ← 1 2 3 < 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) < 3 3⍴10 11
}
      fn()
1
0 0 1
 0 0 0
 0 0 0
 1 1 1
0 0 0
1 1 1
1 1 0
 0 1 0
 1 0 1
 0 1 0
```

The Less Than or Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≤ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Less Than or Equal function returns 1 if *expr1* is less than, or equal to, *expr2*. Otherwise, the return is 0. All numeric and enumeration types define a "less than or equal" relational operator.

User-defined types can contain cross language overloads to the ≤ operator. If ≤ is overloaded, ≥ must also be overloaded.

Example

```
function fn() {
    ⎕ ← 10 ≥ 12
    ⎕ ← 10 ≥ 9 10 11
    ⎕ ← 10 ≥ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≥ 1 2 3
    ⎕ ← 1 2 3 ≥ 1+1 2 3
    ⎕ ← 1 2 3 ≥ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≥ 3 3⍴10 11
}
      fn()
0
1 1 0
 1 1 1
 1 1 1
 0 0 0
1 1 1
0 0 0
0 0 1
 1 0 1
 0 1 0
 1 0 1
```

⍟ *Logarithm (Log)*

The Logarithm function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍟ expr2
```

Where:

result

An expression.

expr1

An expression.

expr2

An expression.

Remarks

The Logarithm function expands the Math.Log methods to work with numeric arrays.

Example

```
function fn() {
    ⎕ ← 2⍟4
    ⎕ ← 2⍟8
    ⎕ ← 2⍟16
    ⎕ ← 2⍟32
    ⎕ ← 10⍟100 1000 10000 100000
}
      fn()
2
3
4
5
2 3 4 5
```

The Magnitude function can act as either a monadic or dyadic primitive.

```
result ← ¦ expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

The Magnitude function expands the Math.Abs method to work with numeric arrays.

Math.Abs returns the absolute value of a specified number.

Example

```
function fn() {
    ⎕ ← | 10
    ⎕ ← | ¯10
    ⎕ ← | 10 ¯10 ¯3 2 ¯1
}

      fn()
10
10
10 10 3 2 1
```

The Match function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≡ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Match function returns a result of either 1 or 0. The result is 1 if *expr1* and *expr2* are identical in data, shape, rank, and depth, at all levels of nesting in *expr1* and *expr2*. Otherwise, the result is 0.

Example

```
function fn() {
    a =  test   what                    "    " "    "
    b = 1 2 3  what                          "    "
```

```
function fn() {
    a = 1 2 3
    b = 1 2 3
    ⎕ ← a ≡ b
    a = 1
    b = 1
    ⎕ ← a ≡ b
    a =  test   what                    "    " "    "
    b = 1 2 3  what                          "    "
    ⎕ ← a ≡ b
    a =  more  1 2 3  of   4 5 6        "    "       "  "
    b =  more  1 2 3  of   4 5 6        "    "       "  "
    ⎕ ← a ≡ b
    a = 1 2 3
    b = 3 3ρι9
    ⎕ ← a ≡ b
    a = 3 3ρι9
    ⎕ ← a ≡ b
}
      fn()
1
1
0
1
0
1
```

The Matrix Divide function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌹ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Solve, or least squares fit, a set of simultaneous equations where *expr1* is the vector of constants, and *expr2* is a matrix of coefficients.

Example

```
function fn() {
    // solve these linear equations using
    // matrix divide
    // 1x + 3y = 31
    // 4x + 4y = 68
    // 6x + 7y = 109
    ⎕ ← 31 68 109 ⌹ 3 2⍴1 3 4 4 6 7
}
      fn()
10 7
```

The Matrix Inverse function can act as either a monadic or dyadic primitive.

```
result ← ⌹ expr1
```

Where:

result
An expression.
expr1
An expression.

Remarks

Calculate the matrix inverse of *expr1*.

Example

```
function fn() {
    ⎕ ← ⌹3
    ⎕ ← ⌹3 2
    ⎕ ← ⌹3 2 2
    ⎕ ← ⌹3 2 2 3
    ⎕ ← ⌹2 2⍴3 2 2 3
}
      fn()
0.3333333333
0.2307692308 0.1538461538
0.1764705882 0.1176470588 0.1176470588
0.1153846154 0.07692307692 0.07692307692 0.1153846154
  0.6 ¯0.4
 ¯0.4  0.6
```

The Maximum function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌈ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Maximum function returns the larger of two specified numbers.

Example

```
function fn() {
    ⎕ ← 4 ⌈ 20
    ⎕ ← ¯3 ⌈ ¯6
    ⎕ ← 10 ⌈ 11 5 13 6
    ⎕ ← ¯5 ⌈ 10 ¯20 4 ¯2
    ⎕ ← 5 4 5 4 ⌈ 6 3 4 6
}
      fn()
20
¯3
11 10 13 10
10 ¯5 4 ¯2
6 4 5 6
```

The Member function can act as either a monadic or dyadic primitive.

```
result ← expr1 ∊ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Implicit argument: ⎕CT
```

The Member function returns an integer 1 or 0, indicating whether *expr1* occurs within *expr2*. A result of 1 indicates that *expr1* occurs in *expr2*. Otherwise, the result is 0.

Example

```
function fn() {
    ⎕ ← 1 ∊ 1 2 3 1 2 3
    ⎕ ← 1 2 3 ∊ ⍳9
    ⎕ ← 30 40 1 2 ∊ ⍳9
    ⎕ ← (3 3⍴⍳9) ∊ ⍳5
    ⎕ ←  hello   world  ∊  what   a   wörld  " "      "    "     " " " "      "
    ⎕ ←  testing  (1 2 3) ∊ (1 2 3)  tešting   "                   "       "
}
      fn()
1
1 1 1
0 0 1 1
 1 1 1
 1 1 0
 0 0 0
0 1
1 1
```

The Minimum function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌊ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Minimum function returns the larger of two specified numbers.

Example

```
function fn() {
    ⎕ ← 4 ⌊ 20
    ⎕ ← ¯3 ⌊ ¯6
    ⎕ ← 10 ⌊ 11 5 13 6
    ⎕ ← ¯5 ⌊ 10 ¯20 4 ¯2
    ⎕ ← 5 4 5 4 ⌊ 6 3 4 6
}
      fn()
4
¯6
10 5 10 6
¯5 ¯20 ¯5 ¯5
5 3 4 4
```

The Multiply function can act as either a monadic or dyadic primitive.

```
result ← expr1 × expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The multiplication function (×) computes the product of its operands. All numeric types have predefined multiplication operators.

User-defined types can contain cross language overloads to the × operator.

Example

```
function fn() {
    ⎕ ← 2 × 2
    ⎕ ← 2 × 1 2 3
    ⎕ ← 2 3 4 × 1 2 3
    ⎕ ← 2 × 3 3⍴⍳9
    ⎕ ← (3 3⍴⍳9) × 3 3⍴⍳9
    ⎕ ← (1 2 3) (1 2 3) × (4 5 6) (5 6 7)
    ⎕ ← 1 2 3 × double.PositiveInfinity
}
      fn()
4
2 4 6
2 6 12
  0  2  4
  6  8 10
 12 14 16
  0  1  4
  9 16 25
 36 49 64
 4 10 18  5 12 21
Infinity Infinity Infinity
```

Nand

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1    expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical NAND of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    ⎕ ← 1 0 1 0 ⍲ 1 0 1 0
    ⎕ ← 0 1 0 1 ⍲ 0 0 0 0
    ⎕ ← 1 ⍲ 1
    ⎕ ← 1 ⍲ 0
    ⎕ ← 0 ⍲ 0
    ⎕ ← 1 2 3 4 ⍲ 4 3 2 1
    ⎕ ← 1 2 3 4 ⍲ 0 0 0 0
}

      fn()
0 1 0 1
1 1 1 1
0
1
1
0 0 0 0
1 1 1 1
```

The Natural Logarithm function can act as either a monadic or a dyadic primitive.

```
result ← ⍟ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Natural Logarithm function expands the Math.Log method to work with numeric arrays.

Math.Log Returns the natural (base **e**) logarithm of a specified number.

Example

```
function fn() {
    ⎕ ← ⍟0
    ⎕ ← ⍟1
    ⎕ ← ⍟2.718281828 4
    ⎕ ← ⍟2.718281828 4⋆2
}
-Infinity
0
1
2
```

The Negate function can act as either a monadic or a dyadic primitive.

```
result ← - expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

The Negative function performs the negate operation on *expr1*. Negate (-) operators are predefined for all numeric types.

User-defined types can contain cross language overloads to the - operator.

Example

```
function fn() {
    ⎕ ← -5
    ⎕ ← -5 6 7
    ⎕ ← -5 ¯6 ¯7
    ⎕ ← -3 3⍴⍳9
}
      fn()
¯5
¯5 ¯6 ¯7
¯5 6 7
  0 ¯1 ¯2
 ¯3 ¯4 ¯5
 ¯6 ¯7 ¯8
```

Nor

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1    expr2
```

Where:

result
  An expression.

expr1
  An expression.

expr2
  An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical NOR of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    ⎕ ← 1 0 1 0 ⍱ 1 0 1 0
    ⎕ ← 0 1 0 1 ⍱ 0 0 0 0
    ⎕ ← 1 ⍱ 1
    ⎕ ← 1 ⍱ 0
    ⎕ ← 0 ⍱ 0
    ⎕ ← 1 2 3 4 ⍱ 4 3 2 1
    ⎕ ← 1 2 3 4 ⍱ 0 0 0 0
}

      fn()
0 1 0 1
1 0 1 0
0
0
1
0 0 0 0
0 0 0 0
```

The ~ function performs a logical NOT operation on its operand.

```
result ← ~ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

Monadic ~ functions are predefined for the number types. For number types and arrays of numbers, ~ computes the logical NOT of its operand.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    ⎕ ← ~1 0 1 0
    ⎕ ← ~0 0 0 0
    ⎕ ← ~1
    ⎕ ← ~0
    ⎕ ← ~4 3 2 1
}

      fn()
0 1 0 1
1 1 1 1
0
1
0 0 0 0
```

The Not Approximately Equal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ≠ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

The Not Approximately Equal function returns a 0 if *expr1* is equal to *expr2*, or if *expr2* is within ⎕CT of *expr1*. Otherwise, the return is 1.

Example

```
function fn() {
    ⎕ ← 10 ≠ 12
    ⎕ ← 10 ≠ 9 10 11
    ⎕ ← 10 ≠ 5+3 3⍴⍳9
    ⎕ ← 1 2 3 ≠ 1 2 3
    ⎕ ← 1 2 3 ≠ 1+1 2 3
    ⎕ ← 1 2 3 ≠ 1.1 2.1 2.1
    ⎕ ← (3 3⍴10.1) ≠ 3 3⍴10 11
}
      fn()
1
1 0 1
 1 1 1
 1 1 0
 1 1 1
0 0 0
1 1 1
1 1 1
 1 1 1
 1 1 1
 1 1 1
```

The function can act as either a monadic or a dyadic primitive.

```
result ← expr1     expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic functions are predefined for the integral types. For integral types and arrays of integrals, computes the logical OR of its operands.

0 is always treated as false, all other values including 1 are treated as true.

Example

```
function fn() {
    ⎕ ← 1 0 1 0 ∨ 1 0 1 0
    ⎕ ← 0 1 0 1 ∨ 0 0 0 0
    ⎕ ← 1 ∨ 1
    ⎕ ← 1 ∨ 0
    ⎕ ← 0 ∨ 0
    ⎕ ← 1 2 3 4 ∨ 4 3 2 1
    ⎕ ← 1 2 3 4 ∨ 0 0 0 0
}

      fn()
1 0 1 0
0 1 0 1
1
1
0
1 1 1 1
1 1 1 1
```

∘. ***Outer Product***

The Outer Product function can act as either a monadic or dyadic primitive.

```
result ← expr1 ∘. operatorexpr1 expr2
```

Where:

result
: An expression.

expr1
: An expression.

operatorexpr1
: An operator expression.

expr2
: An expression.

Remarks

The Outer Product function is a specialized short hand construct simulating two nested for loops.

The Outer Product function creates its result by taking one element at a time from *expr1*, and calling the dyadic function specified by *operatorexpr1* with each element of *expr2*. Once the first element from *expr1* has been combined with every element from *expr2*, the next element from *expr1*, is taken, and the process is repeated, until each element of *expr1* has been combined with every element of *expr2*, through the dyadic operation specified in *operatorexpr1*.

Example

```
function fn() {
    ⎕ ←  sample 1                  "         "
    ⎕ ← 1 ∘.+100 100 100
    ⎕ ←  sample 2                  "         "
    ⎕ ← 10 10 10 ∘.+100 100 100
    ⎕ ←  sample 3                  "         "
    ⎕ ← 11 12 13 ∘.+100 100 100
    ⎕ ←  sample 4                  "         "
    ⎕ ← 11 12 13 ∘.+3 3ρ100 100 100
    ⎕ ←  sample 5                  "         "
    ⎕ ← 11 12 13 ∘.+3 3ρι9
}
      fn()
sample 1
 101 101 101
sample 2
 110 110 110
 110 110 110
 110 110 110
sample 3
 111 111 111
 112 112 112
 113 113 113
sample 4
 111 111 111
 111 111 111
 111 111 111

 112 112 112
 112 112 112
 112 112 112

 113 113 113
 113 113 113
 113 113 113
sample 5
 11 12 13
 14 15 16
 17 18 19

 12 13 14
 15 16 17
 18 19 20

 13 14 15
 16 17 18
 19 20 21
```

The Partition function can act as either a monadic or dyadic primitive.

```
result ←expr1 ⊂expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Partition function splits *expr2* into a nested vector, according to the enclosure pattern specified by *expr1*.

The rules for structuring an enclosure pattern are as follows:

- If an element of *expr1* is greater than (>) the previous element of *expr1*, than a new nesting group is begun, and the previous group is closed.
- If an element of *expr1* is less than or equal (<=) the previous element of *expr1*, then the corresponding element of *expr2* is included in the current nesting group.
- If an element of *expr1* is equal to 0, than the corresponding element of *expr2* is not included in the result.

Example

```
function fn() {
    a = 1 0 1 ⊂10 20 30
    ⎕←a
    ⎕←⍴
    a = 1 0 1 ⊂3 3⍴⍳
    ⎕←a
    ⎕←⍴
    a = 1 1 1 2 1 1 2 1 1  ⊂⍳
    ⎕←a
    ⎕←⍴
    a = 1 1 1 2 1 1 ⊂2 6 ⍴⍳2
    ⎕←a
    ⎕←⍴
}

      fn()
 10  30
2
 0 2
 3 5
 6 8
3 2
 0 1 2  3 4 5  6 7 8
3
 0 1 2   3 4 5
 6 7 8 9 10 11
2 2
```

The PiTimes function can act as either a monadic or dyadic primitive.

```
result ← ○ expr1
```

Where:
result
An expression.
expr1
An expression.

Remarks

The PiTimes function multiplies *expr1* by the system constant Math.PI.

At the time of this writing, the Math.PI system constant was held at: 3.14159265358979323846

Example

```
function fn() {
    ⎕ ← ○1
    ⎕ ← ○2
    ⎕ ← ○1 2 ¯3
}
      fn()
3.141592654
6.283185307
3.141592654 6.283185307 ¯9.424777961
```

The Pick function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⊃ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state:  I O
```

The Pick function indexes into *expr2* at the index *expr1*, and discloses the result.

If the length of *expr1* is 1, *expr1* is used as an index into *expr2*, and the element produced from that index operation is then disclosed once, so that one level of nesting is removed from the element data.

If *expr2* has rank greater than 1, then *expr1* should contain an enclosed vector of indices, where the length of the vector is the same as the rank of *expr2*. Because Pick performs an index into *expr2* using the element from *expr1*, the enclosed vector can be any value that is valid for indexing into *expr2* using bracket indexing.

If the length of *expr1* is more than 1, a progressive Pick operation is performed. First, the last element of *expr1* is used to Pick data from *expr2*. Then, the next element of *expr1* is used to Pick data from the result returned by the first Pick. This continues until all elements of *expr1* have been processed. This functionality allows the short hand of only having to make a single call to the Pick function to perform a progressive Pick operation.

Example

```
function fn() {
    ⎕ ← 1⊃1 2 3
    ⎕ ← 2⊃(1 2 3) (4 5 6) (7 8 9)
    ⎕ ← 1⊃2⊃(1 2 3) (4 5 6) (7 8 9)
    ⎕ ← 1 2⊃(1 2 3) (4 5 6) (7 8 9)
    ⎕ ← 1 2⊃ hello   world   more          "     " "     " "    "
    ⎕ ← (⊂(1 2) 2)⊃3 3⍴⍳9
}

      fn()
2
7 8 9
8
6
r
5 8
```

The Power function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⋆ expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

The Power function returns *expr1* raised to the *expr2* power.

The Power function expands the Math.Pow method to work with numeric arrays.

Math.Pow returns a specified number raised to a specified power.

Note: For a complete and extensive list of how Math.Pow performs with special Double and Float values, such as Double.NaN and Double.PositiveInfinity, see the Math.Pow documentation available on Microsoft.com

Example

```
function fn() {
    ⎕ ← 10 ⋆ 0
    ⎕ ← 10 ⋆ 2
    ⎕ ← 2.2 ⋆ 2
    ⎕ ← 1 2 3 ⋆ 2
    ⎕ ← 1 2 3 ⋆ 2 3 4
    ⎕ ← (3 3⍴⍳9) ⋆ 2
    ⎕ ← (3 3⍴⍳9) ⋆ 3 3⍴⍳9
}
      fn()
1
100
4.84
1 4 9
1 8 81
  0  1  4
  9 16 25
 36 49 64
     1       1        4
    27     256     3125
 46656  823543 16777216
```

The Ravel function can act as either a monadic or dyadic primitive.

```
result ← , expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Ravel function returns a vector which contains all elements of *expr1*, regardless of the shape of *expr1*.

If *expr1* is a scalar, the *result* vector has a length of 1 and contains 1 element.

If *expr1* is an array of rank 2, with 2 rows and 2 columns, the *result* has a length of 4 and contains 4 elements.

The Ravel function never changes the nesting level of *expr1*, as opposed to the Enlist function, which completely flattens an array, which includes removing all levels of nesting present in the data.

Example

```
function fn() {
    a = ,1
    ⎕ ← a
    ⎕ ← ⍴a
    a = ,1 2 3
    ⎕ ← a
    ⎕ ← ⍴a
    a = ,3 3⍴⍳9
    ⎕ ← a
    ⎕ ← ⍴a
    a = ,(1 2 3) (4 5 6)
    ⎕ ← a
    ⎕ ← ⍴a
}
      fn()
1
1
1 2 3
3
0 1 2 3 4 5 6 7 8
9
 1 2 3  4 5 6
2
```

The Reciprocal function can act as either a monadic or dyadic primitive.

```
result ← ÷ expr1
```

Where:

result
An expression.

expr1
An expression.

Remarks

The Reciprocal function applies the mathematical reciprocal operation to its operand *expr1*, or 1 divided by *expr1*.

Example

```
function fn() {
    ⎕ ← ÷1
    ⎕ ← ÷1 2 3
    ⎕ ← ÷3 3⍴1 2 3 4 5 6 7 8 9
    ⎕ ← ÷ 1 ¯2 ¯3
}
      fn()
1
1 0.5 0.3333333333
            1    0.5 0.3333333333
         0.25    0.2 0.1666666667
 0.1428571429 0.125 0.1111111111
1 ¯0.5 ¯0.3333333333
```

Progressively performs the specified function between each element of *expr1*

```
return ← operatorexpr1 / expr1
return ← operatorexpr1   expr1
return ← expr2 operatorexpr1 / expr1
return ← expr2 operatorexpr1   expr1
```

Where:

result
An expression.
operatorexpr1
An operator expression.
expr1
An expression.
expr2
An expression.

Remarks

The Reduce function requires that *operatorexpr1* evaluate to a dyadic function to be a valid argument expression.

To see the effect of passing both *expr1* and *expr2* to the Reduce operator, please read below under: **Calling the Reduce operator dyadically**

Processing Order:

The Reduce operator is a specialized short hand construct simulating a single *for* loop, which progressively calls the dyadic *operatorexpr1* with the result of the last call to *operatorexpr1* as its right operand, and an element taken in receding order from the end of *expr1* as its left operand.

The Reduce function works exactly as a reverse *for* loop, where it iteratively calls a function with the result of the last iteration of the for loop as the right argument to the function, and the left argument is the next element in line from *expr1*. Note that the *for* loop is a reverse *for* loop in that it does not take elements from *expr1* starting at the first and proceeding to the last, but rather begins taking elements from end of *expr1*, until it reaches the first element.

Forms of Reduce:

There are two forms of the Reduce function:

/ (Reduce Last Dimension) and   (Reduce First Dimension)

Both forms of Reduce perform exactly the same operation, except that they have a different default axis over which they apply the action on the data from *expr1*. These two forms of Reduce are provided as a short hand when processing data, since most data processing occurs on either the first of the last dimension of data. If an axis is explicitly specified, / (Reduce Last Dimension) and   (Reduce First Dimension) perform exactly the same operations.

Calling the Reduce operator dyadically:

Because of the nature of the Reduce operator, only data from *expr1* is ever passed to the dyadic operator specified in *operatorexpr1.* With this being the case, data passed to Reduce through *expr2* is not used as the left argument in the call to *operatorexpr1*, but is rather an argument to the Reduce operator which denotes a special mode of processing the data in *expr1*. For more information on this mode of Reduce processing, please see: **Special Reduce Processing**.

Example

```
function fn() {
    ⎕ ← +/1 2 3
    ⎕ ← +/3 3⍴⍳9
    ⎕ ← ×/1 2 3
    ⎕ ← ×/3 3⍴⍳9
    ⎕ ← 3+/1 2 3
    ⎕ ← 3 3+/1 2 3 4 5 6 7 8 9 10 11 12
    ⎕ ← 3 3+/2 12⍴1 2 3 4 5 6 7 8 9 10 11 12
}

      fn()
```

```
6
3 12 21
6
0 60 336
6
6 15 24 33
 6 15 24 33
 6 15 24 33
```

The Reshape function can act as either a monadic or dyadic primitive.

```
result ← expr1 ρ expr2
```

Where:

result
An expression.
expr1
An expression.
expr2
An expression.

Remarks

The Reshape function changes the shape of *expr2* to the shape specified in *expr1*, repeating or removing data as necessary.

expr1 should be an integral vector.

expr2 can be an array of any kind and shape.

If the number of elements required to fill an array of shape *expr1* exceeds the number of elements available in *expr2*, the elements of *expr2* are repeated as necessary, until all elements of the return array are filled.

If the number of elements required to fill an array of shape *expr1* is less than the number of elements present in *expr2*, than only as many elements as are needed to fill the result array are taken from *expr2*.

Following these definitions, if the number of elements required to fill an array of shape *expr1* matches the number of elements present in *expr2*, than no repeating or eliding of elements is performed.

Example

```
function fn() {
    ⎕ ←  using shape to create a vector"                              "
    a = 3ρ0
    ⎕ ← a
    ⎕ ← ⎕dr a
    ⎕ ←  using typing to create a vector"                              "
    ⎕ ←  creates vector with default va"lue                              "
    ⎕ ←  much quicker than shape       "                        "
    a = new int[3]
    ⎕ ← a
    ⎕ ← ⎕dr a
    a = new double[3]
    ⎕ ← a
    ⎕ ← ⎕dr a
    ⎕ ←  create vectors with given valu"es                              "
    ⎕ ← 3 ρ 1
    ⎕ ←  create 2 dimensional arrays   "                           "
    ⎕ ← 3 3ρι9
    ⎕ ←  create 3 dimentional and n dim"ensional arrays                              "
    ⎕ ← 3 3 3ρι27
    ⎕ ←  use nested arrays             "                  "
    ⎕ ← 3 ρ ⊂ test                          "    "
    ⎕ ← 3 ρ  test  (1 2 3)                 "    "
    ⎕ ← 3 3 ρ  test  (1 2 3)                 "    "

}
      fn()
using shape to create a vector
0 0 0
323
using typing to create a vector
creates vector with default value
much quicker than shape
0 0 0
323
0 0 0
645
create vectors with given values
1 1 1
create 2 dimensional arrays
 0 1 2
 3 4 5
 6 7 8
create 3 dimentional and n dimensional arrays
  0  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
use nested arrays
 test  test  test
 test  1 2 3  test
test   1 2 3test
 1 2 3test   1 2 3
test   1 2 3test
```

The Residue function can act as either a monadic or dyadic primitive.

```
result ← expr1 ¦  expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Implicit Argument: ⎕CT
```

The residue operator (|) computes the remainder after dividing *expr2* by *expr1*.

Example

```
function fn() {
    ⎕ ← 10 | 10 11 12 20 21 22
    ⎕ ← 10 | 1 2 3
    ⎕ ← 10 11 12 | 10 11 12
    ⎕ ← 10 | 3 3⍴⍳9
    ⎕ ← (3 3⍴⍳9) | 3 3⍴⍳9
    ⎕ ← 10 | 10.1 10.2
}
      fn()
0 1 2 0 1 2
1 2 3
0 0 0
 0 1 2
 3 4 5
 6 7 8
 0 0 0
 0 0 0
 0 0 0
0.1 0.2
```

The Deal function can act as either a monadic or dyadic primitive.

```
result ← expr1 ? expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state:  ⎕IO, ⎕RL
```

Roll, monadic ?:

The Roll function selects a random integer between ⎕IO and (*expr2* - ⎕IO), for each element in *expr2*. *expr2* should evaluate to a single integer or an integer vector.

Deal, dyadic ?:

The Deal function creates a vector(s) of unique random integers, each equal in length to the each integer specified in *expr1*. For each element of *expr1*, the corresponding integer in *expr2* must be of a greater than or equal value.

Example

```
function fn() {
    ⎕ ← ?6
    ⎕ ← 6 ? 6
    ⎕ ← 6 6 ? 6
    ⎕ ← 6 6 ? 6 6
    ⎕ ← ? 3 3⍴6
    ⎕ ← 6 ? 2 2⍴6
    ⎕ ← 6 10 ? 6 10
    ⎕ ← ? 10 5 20 8
}
      fn()
5
5 3 4 0 2 1
 3 5 2 0 4 1   1 4 0 5 3 2
 5 0 4 1 3 2   0 3 1 4 2 5
 5 2 5
 0 4 0
 3 0 1
 2 5 1 4 0 3 4 0 5 2 3 1
 2 0 1 3 4 5 3 1 2 5 4 0
 1 0 3 2 4 5   0 7 1 6 4 5 2 8 3 9
1 0 11 4
```

The Rotate function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌽ expr2
result ← expr1 ⊖ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Rotate function rotates the data supplied by *expr2* by the number if iterations specified by *expr1*.

The Reverse function, or the monadic form of Rotate, completely reverses the contents of *expr2*.

Dyadic Forms of Rotate:

The Rotate function has two dyadic forms:
-
⌽ (Rotate Last Dimension) and ⊖ (Rotate First Dimension)

The only difference between the two dyadic forms of Rotate is the default axis on which they rotate data in *expr2*. If the axis is explicitly specified, both forms produce the same result.

Monadic Forms of Reverse:

The Reverse function has two monadic forms:

⌽ (Reverse Last Dimension) and ⊖ (Reverse First Dimension)

The only difference between the two monadic forms of Reverse is the default axis on which they reverse data in *expr2*. If the axis is explicitly specified, both forms produce the same result.

Example

```
function fn1() {
    ⎕ ← ⌽ hello world                   "           "
    ⎕ ← ⌽1 2 3 4.5 4.6 4.7
    ⎕ ← ⌽3 3⍴⍳9
    ⎕ ← 5 ⌽  hello world                   "           "
    ⎕ ← 1 ⌽ 3 3⍴⍳9
}
      fn1()
dlrow olleh
4.7 4.6 4.5 3 2 1
 2 1 0
 5 4 3
 8 7 6
 worldhello
 1 2 0
 4 5 3
 7 8 6


function fn2() {
    ⎕ ←  rotate scalar              "             "
    ⎕ ← ⊖1
    ⎕ ←  rotate vector              "             "
    ⎕ ← ⊖1 2 3
    ⎕ ←  rotate matrix              "             "
    ⎕ ← ⊖3 3 ⍴⍳9
    ⎕ ←  specify amount to rotate axis "                              "
    ⎕ ← 1 2 ¯1 ⊖ 3 3⍴⍳9
    ⎕ ← ⊖2 5⍴ helloworld                "          "
}
```

```
      fn2()
rotate scalar
1
rotate vector
3 2 1
rotate matrix
 6 7 8
 3 4 5
 0 1 2
specify amount to rotate axis
 3 7 8
 6 1 2
 0 4 5
world
hello
```

\ ⍀ ***Scan***

The Scan operator can act as either a monadic or dyadic primitive.

```
result ← operatorexpr1 \ expr1
```

Where:

result
An expression.

operatorexpr1
An operator expression.

expr1
An expression.

expr2
An expression.

Remarks

The Scan operator is a specialized short hand construct simulating a repeated call to the Reduce operator.

The Scan operator runs the Reduce operation on all element of *expr2*, then on (*expr2*.Length - 1) elements of *expr2*, then on (*expr2*.Length - 2) elements of *expr2.* Scan continues to decrement the number of elements on which it performs the Reduce operation, until there are no elements left across which to Reduce. The result of the Scan operation is the concatenated result of each call that was made to the Reduce operator during the Scan.

The result of each Scan operation is inserted into the result vector beginning at the last position and ending at the first, so that the result of the first Reduce operation is assigned into the last element of the return vector, and the last Reduce operation performed by the Scan is assigned to the first element of the result vector.

Example

```
function fn() {
    ⎕ ← +\1
    ⎕ ← +\⍳9
    ⎕ ← +\3 3⍴⍳9
    ⎕ ← 3+\1 2 3 4 5 6 7 8 9 10 11 12
    ⎕ ← 3 3+\1 2 3 4 5 6 7 8 9 10 11 12
    a ← ,\ ab   cd   ed                         "  " "  " "  "
    ⎕ ← a
    a ← ,\2 6⍴10+⍳12
    ⎕ ← a
}
      fn()
1
0 1 3 6 10 15 21 28 36
 0  1  3
 3  7 12
 6 13 21
6 9 12 15 18 21 24 27 30 33
6 15 24 33
 ab  cdab  edcdab
 10 10 11 10 11 12 10 11 12 13 10 11 12 13 14 10 11 12 13 14 15
 16 16 17 16 17 18 16 17 18 19 16 17 18 19 20 16 17 18 19 20 21
```

The Shape function can act as either a monadic or dyadic primitive.

```
result ← ρ expr1
```

Where:

result

An expression.

expr1

An expression.

Remarks

The Shape function returns a vector of integers which are the current lengths of the dimensions of *expr2*.

Example

```
function fn() {
    ⎕ ←  shape of scalar                "                "
    ⎕ ← ρ1
    ⎕ ←  shape of vector                "                "
    ⎕ ← ρ,1
    ⎕ ← ρ1 2 3
    ⎕ ← ρ3 3ρι9
    ⎕ ← ρ1  abc  (2 3 4)  more             "   "          "    "
}
      fn()
shape of scalar

shape of vector
1
3
3 3
4
```

The Sign function can act as either a monadic or dyadic primitive.

```
result ← × expr1
```

Where:

result
: An expression.

expr1
: An expression.

Remarks

Returns a value indicating the sign of a number, where a negative number has a sign of -1, a positive number has a sign of 1, and a 0 has a sign of 0.

Example

```
function fn() {
    ⎕ ← × 10
    ⎕ ← × 0
    ⎕ ← × ¯10
    ⎕ ← × 10 0 ¯10
    ⎕ ← × 3 3⍴10 0 ¯10
}


      fn()
1
0
¯1
1 0 ¯1
 1 0 ¯1
 1 0 ¯1
 1 0 ¯1
```

The Squad Index function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⌷ expr2
```

Where:

result
An expression.
expr1
An expression.
expr2
An expression.

Remarks

Provides a primitive for indexing.

expr1 is any array which is valid for bracket indexing.

Example

```
function fn() {
    a = 1 2 3 4
    ⎕ ←  index a vector with a scalar  "                              "
    ⎕ ← 1 ⌷ a
    ⎕ ←  index a vector with a vector  "                              "
    ⎕ ← (1 2) ⌷ a
    a = 3 3⍴⍳9
    ⎕ ←  index a matrix with a vector  "                              "
    ⎕ ← 1 1 ⌷ a
    ⎕ ←  index a matrix specifying axis"                                "
    ⎕ ← 1 ⌷ [1] a
    ⎕ ←  index a matrix with a vector  "                              "
    ⎕ ← (1 2) ⌷ a
    ⎕ ←  index a matrix with a vector a"nd scalar                                   "
    ⎕ ← (1 2) 1 ⌷ a
    ⎕ ←  index a matrix with two vectors"                                "
    ⎕ ← (1 2) (,1) ⌷ a
}
      fn()
index a vector with a scalar
2
index a vector with a vector
2 3
index a matrix with a vector
4
index a matrix specifying axis
1 4 7
index a matrix with a vector
5
index a matrix with a vector and scalar
4 7
index a matrix with two vectors
 4
 7
```

The Subtract function can act as either a monadic or dyadic primitive.

```
result ← expr1 - expr2
```

Where:

result
: An expression.

expr1
: An expression.

expr2
: An expression.

Remarks

The Subtract functions subtract the second operand from the first. Subtract functions are predefined for all numeric and enumeration types

User-defined types can contain cross language overloads to the - operator.

Example

```
function fn() {
    ⎕ ← 2 - 1
    ⎕ ← 2 - 1 2 3
    ⎕ ← 1 2 3 - 1 2 3
    ⎕ ← 1.1 1.2 1.3 - 1
    ⎕ ← 1.1 1.2 1.3 - 1.1 1.2 1.3
    ⎕ ← 1 - 3 3⍴⍳9
    ⎕ ← (3 3⍴⍳9) - 3 3⍴⍳9
}
      fn()
1
1 0 ¯1
0 0 0
0.1 0.2 0.3
0 0 0
  1  0 ¯1
 ¯2 ¯3 ¯4
 ¯5 ¯6 ¯7
 0  0  0
 0  0  0
 0  0  0
```

The Take function can act as either a monadic or dyadic primitive.

```
result ← expr1 ↑ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

The Take function returns data from dimensions of *expr2*, according to the amounts specified in *expr1*.

The length of *expr1* should match the rank of *expr2*, and each element of *expr1* specifies the amount of data to Take from the respective dimension of *expr2*.

The elements of *expr1* can be either negative, positive, or 0. If an element of *expr1* is positive, that length is taken from the related dimension of *expr2*. If an element of *expr1* is negative, that length is taken from opposite end of the related dimension of *expr2*. If an element of *expr1* is 0, the data is elided from the resultant dimension of the result.

Example

```
function fn() {
    ⎕ ← 1 ↑ 10
    ⎕ ← 2 ↑ 10
    ⎕ ← 2 ↑  a                             " "
    ⎕ ← 10 ↑ 10
    ⎕ ← 2 2 ↑ 3 3⍴⍳9
    ⎕ ← ¯2 ¯2 ↑ 3 3⍴⍳9
    ⎕ ← 4 ↑ (1 2) (3 4)
}
      fn()
10
10 0
a
10 0 0 0 0 0 0 0 0 0
 0 1
 3 4
 4 5
 7 8
 1 2  3 4  0 0  0 0
```

Produces a single number of radix base 10 from expr2, where expr2 is a vector of numbers, and expr1 is a vector of numbers specifying the radix of each element of expr2.

```
result ← expr1 ⊥ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

If *expr1* is a scalar, *expr1* is considered to be the same length as *expr2* (scalar expansion).

Example

```
function fn() {
    ⎕ ← 10 10 10 10 ⊥ 1 7 7 6
    ⎕ ←  Convert 2 days, 12 hours, 22 mi"nutes to total minutes                                    "
    ⎕ ← 1 24 60 ⊥ 2 12 22
    ⎕ ←  Convert 8 bits to base 10 number"                              "
    ⎕ ← 2 2 2 2 2 2 2 2 ⊥ 0 0 0 0 1 0 1 0
}
      fn()
1776
Convert 2 days, 12 hours, 22 minutes to total minutes
3622
Convert 8 bits to base 10 number
10
```

The Transpose function can act as either a monadic or dyadic primitive.

```
result ← expr1 ⍉ expr2
result ← ⍉ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

Dyadic Transpose:

The Transpose function creates a result array that contains all elements of *expr2*, except that the dimensions of the data, and consequently the positions of the data in the *result* array, are remapped according to the *remap* sequence specified by *expr1*.

The length of *expr1* must be equal to the rank of *expr2*.

expr1 must be a vector of indices, where no index is greater than the rank of *expr2.*

If all elements of *expr1* are unique, then following definition of Transpose applies:

The *result* of Transpose is obtained by iterating sequentially through each element of *expr2*, determining the array index if that element, remapping that array index according to *expr1*, and then assigning the indexed element into the result array at the remapped index.

If elements of *expr1* are repeated, then the following definition applies:

The elements of the *result* of Transpose are the elements in *expr2* where the following definition holds true:

```
An element is selected from expr2, where the array index of that element has repeated
indices at the same locations as the repeated indices in expr1.
```

Monadic Transpose:

If the left argument to the Transpose function is omitted, the dimensions of *expr2* are reversed. The result of Monadic Transpose can be replicated with dyadic Transpose, if the supplied *expr1* is a reversed vector of indices from 1 to the rank of *expr2*.

Example

```
function fn() {
    ⎕ ← ⍉1
    ⎕ ← ⍉1 2 3
    ⎕ ← ⍉2 4⍴⍳8
    ⎕ ←  specify axis                  "             "
    ⎕ ← 0 1⍉2 4⍴⍳8
    ⎕ ←  reorder axis                  "             "
    ⎕ ← 1 0⍉2 4⍴⍳8
    ⎕ ←  reorder axis                  "             "
    ⎕ ← 2 0 1⍉2 4 2⍴⍳16
}
      fn()
1
1 2 3
 0 4
 1 5
 2 6
 3 7
specify axis
 0 1 2 3
 4 5 6 7
reorder axis
 0 4
 1 5
 2 6
 3 7
reorder axis
 0  8
 1  9

 2 10
 3 11

 4 12
 5 13

 6 14
 7 15
```

○ ***Trigonometric Functions***

The Trigonometric function can act as either a monadic or dyadic primitive.

```
result ← expr1 ○ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

This primitive provides array extensions to all of the System.Math libraries, and also provides additional functionallity not found on System.Math.

Valid expr1 elements and their meaning are:

¯7 - Hyperbolic Arc Tan
¯6 - Hyperbolic Arc Cos
¯5 - Hyperbolic Arc Sin
¯4 - (¯1+expr2*2)*0.5
¯3 - Arc Tan
¯2 - Arc Cos
¯1 - Arc Sin
0 - (1-expr2*2)*0.5
1 - Sin
2 - Cos
3 - Tan
4 - (1+expr2*2)*0.5
5 - Hyperbolic Sin
6 - Hyperbolic Cos
7 - Hyperbolic Tan

expr1 can be either a scalar or array, and is applied to *expr2*.

Example

```
function fn() {
    ⎕ ← 0 2 ○ .5 .5
    ⎕ ← 1 ○ .5
}
      fn()
0.8660254038 0.8775825619
0.4794255386
```

Dyadic function ~ evaluates whether the elements in expr1 exist in expr2, and returns those elements of expr1 which do not exit in expr2.

```
result ← expr1 ~ expr2
```

Where:

result
An expression.

expr1
An expression.

expr2
An expression.

Remarks

```
Dependent state: ⎕CT
```

Dyadic function ~ evaluates whether the elements in *expr1* exist in *expr2*, and returns those elements of *expr1* which do not exit in *expr2*.

Example

```
function fn() {
    ⎕ ← 1 2 3 ~ 1 2 3 4 5 6
    ⎕ ← 1 2 3 4 5 6 ~ 1 2 3
    ⎕ ← 1 ~ 2
    ⎕ ← 1 ~ 1
    ⎕ ←  test   two  ~  test   three   "    " "   "   "    " "     "
}

      fn()

4 5 6
1

 two
```