VIRTUE - A DIFFERENT APPROACH TO (PARALLEL) PROGRAMMING

Throughout the development of the calculating machines into computers the major technical problem to be solved was the reduction of the amount of components and their bulk. This led inevitably to the use of serial processing. Though the nature itself is infinitly parallelised, the technical problems of computer development led also to the development of such software tools and programming languages which mirrored the serial nature of computors, so the serialisation of parallel natural processes is performed by humans, the programmers. Modern day developments of scientific and every-day needs for computing power has led to the introduction of multiprocessors, GPUs, clusters, grids and clouds of computers, as to ascertain enough processing speed for very complex algorithms. However, the programming languages that support these developments are still heavily based in serial programming, and the multi-computer environment is accessed only through programmed interfaces, where all the burden of parallelising now well known serial algorithms is again on the programmers.

Virtue is a programming language which proposes a different approach, by keeping the inherent parallel structure of natural algorithms, and doing the parallel processing by itself, if it is algorithmically possible. Virtue is a syntactically very simple, yet semantically extremely complex language, in which, for example, the source text of the algorithm for Conway's “Game of Life” necessitates only 10 language tokens (6 words, 14 numbers in 3 vectors and 1 delimiter).

Serial vs. Parallel Processing

Virtue Language is designed to keep as many logically parallel algorithmic structures expressed in such a way that the parallelism is obvious. However, an algorithm is, per definitionem, a *series* of operations, possibly intermixed with decision making.

Virtue tries to keep consistency inside the two sets of operations, the:

- serial operations
- parallel operations

However, it should be noted, that the whole language can be implemented on a purely serial computer, but can also be implemented on parallel data processing equipment (multiprocessor, vector, cluster, grid...). The consistency of semantical distinction between parallel (i.e. parallelizable) and serial (non-parallelizable) instructions in an program on many levels does not a-priory dictate the implementation of Virtue. Although multidimensional logical values are parallel constructs, it would be counter-productive to parallelise a logical operation on 2, 4 or 8 individual values (or wouldn't it?). However operations on conformable multi-dimensional arrays are also parallel constructs, but it would be very advisable to implement them in parallel on several independent processing units.

One of the consequences of the distinction between serial and parallel operations is that for example inside FUNCTIONs which are EXECUTEd, any global variable can be changed (using SET or ASSIGN). However, inside FUNCTIONs which are EACHed or MASKed, SET and ASSIGN to any variable except the variables explicitly defined in the FUNCTION header (for example: VAR $a $bbb $c FUNCTION ... ;) is strictly prohibited. In other words serially EXECUTEd FUNCTIONs may have side-effects, in parallely EACHed (MASKed etc.) FUNCTIONs side-effects are forbidden.

Such a strict distinction is necessary, as there is no a-priory order in parallel operations, and there is no communication possible between parallely executing operations (often used in APL jargon "scalar" operations).

Multidimensionality

Virtue enables multidimensional processing on several levels.

- Multidimensional arrays, from 1 to n dimensions, where n is caped only by the amount of memory of the Virtue processing equipment (which may be implemented on single computors, but also on clusters and even grids)

- Multidimensional numeric values, that is numbers of 1 (REAL - R), 2 (COMPLEX - C), 3 or 4 (QUATERNION - Q) and 5, 6, 7 or 8 dimensions (OCTONIONS - O)

- Multidimensional fuzzy logic/probability values of 1 (LOGIC REAL), 2 (LOGIC COMPLEX), etc. up to 8 dimensions (LOGIC OCTONIONS). All logic values are directly interchangable with numeric values, and the general convention (and therefor also the conversion rule) is that they live in a non-signed unity space, i.e. in the space [0.0,1.0] of numerical values.

Virtue multidimensionality consistency is supported by the ENCAPSULATE and DECAPSULATE operations, which convert [1,8]-dimensional vectors into R, C, O or Q numbers and vice versa; by the RESHAPE and other indexing/transforming operations, which use multidimensional numbers as [1,8]-dimensional indexes for arrays of up to 8 dimensions, and by INTERVAL, which uses multidimensional numbers for production of index arrays.

The operation family of REDUCE is generically *Serial*, as there is a specific known order prescribed by their definition. (1 2 3 4) SUBTRACT REDUCE gives the same result as 1 2 3 4 SUBTRACT SUBTRACT SUBTRACT. This RPN execution order is given in common notation as 1-(2-(3-4)), and this order is for non-commutative operations (like SUBTRACT, DIVIDE, or e.g. QUATERNION MULTIPLY etc.) theoretically not changable. Although REDUCE type operations may be partially parallelized in the implementation, with great care and with certain inter-computor (inter-processor) communication, their *semantics* is *serial*. Therefore FUNCTION ...; REDUCE constructs may use global variables inside FUNCTIONs (see Serial vs. Parallel Processing).

Multidimensional Fuzzy Logic

Virtue uses multidimensional fuzzy logic.

The **Monadic LOGICAL** converts (n <= 0.0 --> 0.0; n >= 1.0 --> 1.0; else n), in each dimension of the (possibly) multidimensional logic value. Therefore Virtue implements fuzzy logic. Values > 0.0 and < 1.0 are interpreted logically using Łukasiewicz logic t-norm. Wherever in a Virtue programme decisions are made based on LOGIC values, a

positive (or negative) decision will be made based on the probabilities expressed in the fuzzy logic value, i.e. LOGIC 1.0 is certainly yes, 0.0 certainly not, while other values will produce YES/NO decisionmaking results on the base of random selection based on the probability. For example the LOGIC value 0.5 will take the YES and NO decision branch in a Virtue programme with equal, randomly distributed statistical probability. The LOGIC value 0.7 will take the YES branch in average 70% of the cases. The LOGIC value 0.001 will almost always take the NO branch, but still, statistically (!), it would once in a thousand decisions take the YES branch.

The **Monadic BOOLEAN** converts a (multidimensional) numeric value into a (multidimensional fuzzy) logic value, and then believes that all LOGIC values > 0.5 are #TRUE, all < 0.5 are #FALSE. However, the idecidable 0.5 is processed as a probability, so the result of Monadic BOOLEAN for a fuzzy logic value of 0.5 will be #TRUE or #FALSE statistically equally probably. All LOGIC operations on BOOLEANs give BOOLEAN results. BOOLEANs are generically INTEGERs 0 and 1 (#FALSE and #TRUE).

The **Monadic PROBABLE** converts LOGICAL values into BOOLEAN, but taking into account the probability as the fuzzyness of LOGICAL. The result for each (multidimensional) value is randomly taken based on each probability value/dimension. A long series of PROBABLE is statistically correct in random choices based on the probability.

Multidimensional logic is implemented in 1 up to 8 dimensions, i.e. with REALs (1 dimension or scalar), COMPLEX (2 dimensions), QUATERNION (4 dimensions) and OCTONION (8 dimensions). To be able to define 3, 5, 6 and 7 dimensional fuzzy logic values, the unneeded right hand dimensions (i.e. the k-part of QUATERNION or the m.n.o-part, n.o-part or o-part of OCTONION may be filled with NaN, therefore enabling non-processing of this dimension (which is specifically important when calculating CONTRADICTION). For example, a 3 dimensional fuzzy logic value may be 0.1i1j0.5kNaN, a 5 dimensional example is 1i0j0k1l1mNaNnNaNoNaN. The value of a Not a Number (NaN) is consistently NaN, except when multiplied by 0 and divided by itself (any object taken 0 times is 0, any object goes exactly 1 times into itself - division NaN by NaN gives 1).

ASCII and INTEGER LOGIC conversion always yields BOOLEAN values
BOOLEAN is defined for LOGIC 1.0 and 0.0 as #TRUE and #FALSE

SERIAL DECISION INSTRUCTIONS and multidimensional logic

Virtue has serial decision instructions, i.e. instructions which are used inline of a serial processing algorithm (even though the algorithm itself may use many parallel processing instructions, i.e. using multidimensional numbers and arrays). These decision making instructions (the IF, CHECK, IF_YES, IF_NO, JUMP) have three major programme flow altering (i.e. decision) functions in Virtue, depending on the decision-making data provided:

- Decisions based on Truth values
- Decisions based on Probability values
- Decisions based on Contradiction values

When using IF, CHECK, IF_YES, IF_NO, JUMP serial processing logic (decision) instructions there are several possibilities (this is true for each individual IF_YES and IF_NO, as well as any individual decision based on the same data):

1. *Truth*: The checked value is single-dimensional (i.e. an ASCII, INTEGER or REAL) and can be converted to BOOLEAN: A simple YES/NO decision is taken

2. *Probability*: The checked value is single-dimensional fuzzy value: The YES decision is taken statistically (using comparison of random value with the LOGICAL probability value) more often as the value approaches 1.0; the NO decision statistically more often as the value approaches 0.0

3. *Contradiction*: The checked value is multi-dimensional (i.e. a COMPLEX, QUATERNION or OCTONION). In this case the below explained "measure of non-contradiction" is used on the multidimensional LOGIC value to get a single-dimensional quantity, necessary for serial processing (serial processing is single-dimensional processing!). If the Measure of Non-Contradiction is 0.0 or 1.0, then a simple YES/NO decision is taken

4. *Contradiction As Probability*: The checked value is multi-dimensional, and processed according to [3. *Contradiction*] above. However the resulting single-dimensional LOGIC value is not BOOLEAN i.e. is > 0.0 or < 1.0. The Non-Contradiction is resolved by using statistical probability decision making as in [2. Probability]. The decision will be YES the more statistically often the more non-contradictory the multi-dimensional logical value is.

Beware when using IF_YES and IF_NO constructs with non-boolean (i.e. fuzzy) logic and SCALARs, ADDRESSes or VARIABLEs (For example:
0.7 CHECK 2 IF_YES 3 IF_NO. This operation sequence may leave none, one or even two

numbers on the stack. Therefore, if you do not know exactly (!) what you are doing (i.e. this opportunity is left open for special programming algorithms. The Niladic STACKSIZE may be used to determine the number of items on the stack.)

Degree of Contradiction

IF and CHECK, i. e. SERIAL PROCESSING LOGIC INSTRUCTIONS, use the "Degree of Contradiction", as defined in "Multidimensional Logic: A model for Paraconsistent Logic" by Carlos Gershenson (**http://cogprints.org/1479/0/mdl.html**):

>>Let a value of truth be given by the vector (x,y). We can define the degree of contradiction C with:

$C(x,y)=|(x+y)-1|$

C can be seen as "how far do you get from fuzzy logic". If C is 0, then the point belongs to the fuzzy line, there's no contradiction, and it is contained by fuzzy logic. As C increases, you go farther from the fuzzy line, until you reach total contradiction. With C, we can see "how contradictory" a proposition is.<<

SERIAL PROCESSING LOGIC INSTRUCTIONS (IF, CHECK, IF_YES, IF_NO, JUMP) use the Contradiction as their fuzzy logic input (due to the fact that multidimensional logic variables pressupose multidimensional processing, not serial processing!). Based on the value of the NOT Contradiction (1 - C) the scalar thruth value is defined. This is the "measure of non-contradiction".

Example:

If we use two-dimensional logic values (complex logic values), then for example 0i1 or 1i0 will have a CONTRADICTION of 0, i.e. they are fully expressable in singledimensional fuzzy logic. Therefore the IF_YES condition is met, there is no contradiction.

All logic operations are consistent with Lukasiewitcz t-norm, and multidimensional operations with the above reference.

The **Monadic CONTRADICTION** converts a multidimensional logic value into the amount of Contradiction from one-dimensional logic, i. e. "how far are we from fuzzy logic, how contradictory is the proposition". In a manner of speaking it is the LOGIC

equivalent of the Monadic MAGNITUDE, which gives the scalar length of a (multidimensional) vector value.

For Complex (2-dimensional) Logic:
AiB CONTRADICTION --> abs((A+B)-1)
For Quaternion (3- and 4-dimensional) Logic:
AiBjC CONTRADICTION --> abs((A+B+C)-2)
AiBjCkD CONTRADICTION --> abs((A+B+C+D)-3)
etc.

Masks

The Triadic MASK has two possible modifiers: WRAP and REFLECT. The MASK (with or without modifiers) will never recurse into substructures, as this is not consistently implementable. For example:

1	(2 3 (4 5))	(6 7)	
(8 9)	10		11
12	((13 (14 15)) 16 (17 18))	(19)	

can obviously not be recursed by MASK into substructures in a consistent way. The FUNCTION which is MASKed will get an argument array of the shape of the mask applied, with substructures of the original array being the substructures of the FUNCTION argument array. However, the scalar MULTIPLY used by the MASK operation will descend into the substructures. Therefore

2 (3 3) RESHAPE FUNCTION ... ; MASK

applied to the example array will, for the first place [index (1 1) or 1i1] give the following argument to the FUNCTION:

0	0	0
0	2	(4 6 (8 10))
0	(16 18)	20

MASK, WRAP MASK and REFLECT MASK

MASK will beyond the edges of the array take (make) empty (' ' or 0) elements.

WRAP MASK will beyond the edges take the dimensionally oposite elements. It wraps diagonally around the masked array.

REFLECT MASK will in the same situation reflect the existing elements. It reflects of straigth lines; in corners, i.e. diagonally, it reflects the reflections, i.e. areas from outside the boarders of the MASKed array will first reflect the content (inside the boarders) and then rereflect it in the same way to fill up the argument array.

Example:

1 2 3
4 5 6
7 8 9

1 (3 3) RESHAPE FUNCTION ... ; MASK will at the first position, where number 1 is centered, give the FUNCTION the following argument as input:

0 0 0
0 1 2
0 4 5

and for the last:

5 6 0
8 9 0
0 0 0

1 (3 3) RESHAPE FUNCTION ... ; WRAP MASK will at the same position give FUNCTION the argument:

9 8 7
3 1 2
6 4 5

and for the last:

5 6 4
8 9 7

2 3 1

1 (3 3) RESHAPE FUNCTION ... ; REFLECT MASK will give for the same position the FUNCTION input:
1 1 2
1 1 2
4 4 5

and for the last:

5 6 6
8 9 9
8 9 9

1 (5 5) RESHAPE FUNCTION ... ; REFLECT MASK will give for the same position the FUNCTION input:

5 4 4 5 6
2 1 1 2 3
2 1 1 2 3
5 4 4 5 6
8 7 7 8 9

On large MASKed arrays the REFLECTion and WRAPing is more obvious! REFLECTion could be used for bumping certain structures off the walls, WRAPing will make a n-dimensional spherical space, and structures will continue wrapping around from bottom to top, left to right and diagonally "through" the walls.

Stack Processing

Niladic STACKSIZE may be used to determine the number of items on the stack, counting from the last frame pointer, i.e. not including the possible FUNCTION VARIABLEs. The amount returned is the same which would be cleaned by a CLEAN n-adic. The result of STACKSIZE, being on the TOS, is though not counted!
STACKSIZE is practical with RESTACK, probabilistic scalar, variable or address stacking (e.g.: 12.7 IF_YES) and with functions leaving an undeterministic number of results on the stack.

VIRTUE vs C programming

In this chapter we will present several example algorithms for (mostly) basic simple programmes written in Virtue and written in C.

REMARK:
All examples in both Virtue and C are written between the '---------'... lines, and are supposed to be saved in their respective named files. Programme source files in C have commonly the extension ".c", and programme sources in Virtue will be saved with the extension ".virt". As opposed to C programmes, for which the C compilers mostly insist on ending in ".c", Virtue does not have any such imposition, i.e. all file names are equally allowed, the convention of using ".virt" is practical to distinguish different files in a directory.
After the first programme example in both Virtue and C operating system commands necessary to execute the programme are given between the '=========='... lines. These operating system commands are given for Unix-type (Unix, Solaris, Darwin, UWIN, cygwin...) shells. In further examples these instructions are not given, as only filenames change. When changes in the C compilation command have to be made (as e.g. when using mathematic library routines), these instructions are given again. For Virtue it is supposed that interactive processing will be used, so the programmes may be saved in files, but it is not necessary. Still, for longer programmes we strongly suggest saving them in files, as later referencing, reuse of algorithms and programme changes are much easier in saved files, than writing the sentences again interactively.

EXAMPLE 1: Hello world!

The first example is commonly a prime programme, we want the computer to write a simple sentence on the terminal screen. Commonly this is called the 'Hello world' programme, and prints one line saying 'Hello world!'.

The programme which will do that in C is:

```
---------------------------------------

#include <stdio>

main()
{
     printf ("Hello world!\n");
}

---------------------------------------
```

To get the computer to actualy print out this text, we have to save the above programme text in a file, let's call it "hello.c". After that we have to compile, and then execute the programme:

```
=======================================

cc -o hello-c hello.c
hello-c
```

```
=======================================

```

To get the same result in Virtue, the following sentence is enough:

```
---------------------------------------

'Hello world!'.

---------------------------------------
```

Let's save this sentence in the file 'hello.virt'. To compile/interpret this sentence, i.e. to get it actually printed, we would write the following shell command:

```
=======================================

Virtue -q < hello.virt

=======================================
```

The -q flag to the Virtue processor keeps it 'quiet', that is, it will not print any greetings or prompts, except, naturally errors, if encountered. Virtue is normally used both as a "batch" processor - as in the above example, or, more commonly, as an interactive environment.

If using Virtue interactively, we would not use the -q flag, and the processor would print version information and a prompt. Remark: below we will cite just the input the user has to type, ignoring the possible computer outputs.

Executing the above Virtue programme (Hello world) in the interactive mode is shown below:

```
=======================================

Virtue
'hello.virt' READ.

=======================================
```

However, as the programme hello.virt is extremely simple, you could just type the following:

```
=======================================

Virtue
'Hello world!'.

=======================================
```

The major difference between the "batch" processing mode, and the interactive mode is that you can continue using anything 'READ' by applying other algorithms to the read data, and using the read functions (procedures, programmes, subroutines, sentences... whatever your naming preference is - in Virtue the programmes are composed of one or more sentences, and the functions are defined as FUNCTIONs, NILADICs, MONADICs, DYADICs etc - this will be elaborated on later) in later sentences.

As opposed to Virtue, C has no interactive mode, it is a pure write->compile->execute language. Therefore, you can not use the C programme

above as a part of another programme, but have to change the source file, compile it and than execute.

EXAMPLE 2: Number of words in a text line

The 2nd example deals with a simple task of reading in an arbitrary length (up to the limit of computer memory) string of characters ending with a NewLine (Enter), saving it in a variable for later further processing, and counting the number of words in this text line. For the sake of simpleness, we will presuppose that all words are delimited with exactly one space. The number of words in a sentece is one more than the sum of spaces in this sentence.

The Virtue programme to get the desired result (saving the input text line into the Virtue variable _text):

```
----------------------------------------

QUOTEDINPUT @text ASSIGN
' ' IDENTICAL SUM 1 ADD.

----------------------------------------
```

The QOUTEDINPUT will take input text exactly as it is, and put it on the top of the stack (do not forget that Virtue is a stack based processor, and that operators always have to follow the arguments, i.e. '1 2 ADD'). @_text puts the address of the variable _text onto the stack (note that in Virtue the variable names start with an underscore "_"!), and ASSIGN puts the value form TopOfStack-1 (the QOUTEDINPUTed text line) into the variable addressed by the TopOfStack (the address of _text, i.e. @_text). However, ASSIGN leaves the value put into the variable (the text) on the TopOfStack, so we do not have to get the value from the _text variable on the next line. Mark that the whole above programme of two lines is actually one sentence (ending with "."), and there is no reason this sentence could not be differently formated into lines. Every Virtue programmer will have their own way of writing readable programmes.

As we actually assigned the value of the inputed text line to a variable (or, if you prefer, associated the text with the variable name), this short programme has no sense in batch mode. So, we would use the Virtue processor interactively (see Example 1).

Now we will write a programme in C, which does exactly the same - input a string of characters from the user, end the input when NewLine (Enter) is pressed, save it in the variable "text" (in C we could use also the name "_text", as in Virtue, however in C variables which start with an underscore "_" are uncommonly used, as many include files and libraries use this not to influence the user naming space) and finally printing out the number of words in this text line.

As opposed to using Virtue in interactive mode, as discussed in previous paragraphs, where saving the value of the text line in an variable would have sense, as it can be used later during the interactive session, in the following C programme there is no sense to use the used algorithm to save the text line in a variable, as, when compiled and executed, this saved variable is not used. However, as we will expand this example in Example 3, and have stated in the problem that we do want to save the variable for

possible further processing, the programme does keep the whole text in the variable "text".

```
#include <stdio.h>
#include <malloc.h>

main ()
{
     long int i = 0;
     long int words = 1;
     char * text = (char *) malloc (1);

     while ((text[i++] = getchar()) != '\n')
          if (!(text = realloc (text, i + 1)))
          {
               printf ("Error - out of memory!");
               exit();
          }
     text[i] = 0;

     for (i = 0; i < strlen(text); i++)
          if (text[i] == ' ')
               words++;

     printf ("%d\n", words);

     /*** use text here further ***/

     free (text);
}
```

There are no comments on the algorithms used in the above programme, which was chosen for the brevity of expression. Although the above examples are simple enough not to have to be heavily commented, you can obviously see that the C programme is getting slightly complex, so it is very advisable that, when programming, you never forget to put meaningfull comments, and enough of them. If something you wrote is obvious to you in that moment, it does not mean that it will obvious, or even understandable, to somebody else reading your programme, and it happens quite often that even to you something you fully understood (and consequently did not comment) after some time is not easily comprehensible. Therefore, again, it is important to put comments on what you intended to do and how you intended to do it in any nontrivial programme.

The same remark about commenting is valid for any programming language (and in human texts we use parentheses and footnotes for the purpose of commenting). Programming in Virtue is no different, only the simplicity of programmes, as could already be seen from only the two examples given so far, makes commenting less frequent. In later examples you will see important commenting in Virtue programmes.

EXAMPLE 3: Addition to Example 2, count the occurences of individual letters

Now, as we have the inputed text in the variable _text, we will write a Virtue sentence to count all the occurences of letters 'A'..'Z' and 'a'..'z' in this text, and write them out as a line of numbers:

```
---------------------------------------

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
              MONADIC text IDENTICAL SUM;
EACH.

---------------------------------------
```

This sentence can be inputed in the Virtue interactive mode directly after finishing the Example 2.

As opposed to that, the C programme can not be expanded differently than adding new statements into the existing frame from Example 2, and then recompiling it. Therefore we cite the whole (Example 2 and Example 3) programme as one:

```
---------------------------------------

#include <stdio.h>
#include <malloc.h>

main ()
{
     long int letters[256];
     long int i = 0;
     long int words = 1;
     char * text = (char *) malloc (1);
     char * alphabet =
"ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz";

     for (i = 0; i < 256; letters[i++] = 0);

     while ((text[i++] = getchar()) != '\n')
          if (!(text = realloc (text, i + 1)))
          {
               printf ("Error - out of memory!");
               exit();
          }
     text[i] = 0;

     for (i = 0; i < strlen(text); i++)
     {
          if (text[i] == ' ')
               words++;
          letters[text[i]]++;
     }

     printf ("%d\n", words);

     for (i = 0; i < strlen(alphabet); i++)
          printf ("%ld ", text[i]);

     printf ("\n");
```

```
    free (text);
}
```

Just to directly compare it, the full Virtue programme giving the same results is:

```
QUOTEDINPUT @text ASSIGN

' ' IDENTICAL SUM 1 ADD

'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
    MONADIC text IDENTICAL SUM;
EACH.
```

A statistical comparison of language complexity for the same task:

- The C programme consists of 38 lines, 106 words and 646 characters.
- The Virtue programme consists of 7 lines, 15 words and 144 characters.

Remark: the amount of lines is not a valid reference, as both the C and the Virtue programme could have been written in one line. However, the number of lines indicates the length of a programme in the respective language written in such a way that it is cut (with empty lines) in reasonable logical chunks, and statements are cut into several lines, and use tabulation, to provide a visual clue to the structure of each of the abovementioned logical chunks.

Here it is important to note that the above C programme was integrated with the C programme from Example 2 in an optimized way. The amount of words is counted in the same loop as the individual characters, and not in two separated consecutive loops with the same index range.

In the Virtue programme no such type of optimisation is done. The philosophycal principles of these two languages (C and Virtue) differ in that sense quite a lot.

However, for optimisation purposes the C programme counts all possible letters, punctuation marks and special characters which may have occured in the input text line, and uses the alphabet only for outputing some of these 256 characters (the char type in C is 8-bit!).

The Virtue programme counts only those characters which are the letters explicitly defined, and, as opposed to C, is not character encoding set size dependant. Implementations of Virtue could work with 6-bit, 7-bit, 8-bit, 16-bit or even larger character sets, and yet the Virtue programme would not have to be changed. This C programme would not work with Unicode, for example, but would have to be changed.

EXAMPLE 4: The multi-dimensional hypersphere

In the 4th example we will do a little character graphics. We want to make a programme which will output a circle of characters "1" in a plane of "0"s. Furthermore, we want our programme to be able to work in a

space of any number of dimensions, from 1 dimensional vector (where the 'circle' will actually be a line of ones with zeros in front an at the end), over a 2-dimensional array (where we will see a real (naturally pixelized) full circle of 1's, over a 3-dimensional space (and we want to print out the 3-dimensional full sphere as a series of 2-dimensional slices (planes), up to, finally, a 8-dimensional hypersphere (also outputed in dimensional slices).

The following Virtue programme is defined as a function which takes three arguments - the space dimensions, the position of the circle/sphere/hypersphere centre, and the circle/sphere/hypersphere radius.

The arguments for the space dimensions and the centre point have to be inputed in the correspondingly dimensional numbers, i.e. a line would result in inputing integer real numbers (i.e. 21 10), a circle inputing integer complex numbers (i.e. 21i21 10i10), a 3-dimensional sphere inputing integer 3-dimensional quaternions (i.e. 21i21j21 10i10j10), 4-dimensional with quaternions (21i21j21k21 10i10j10k10), and 5-, 6-, 7- and 8-dimensional hyperspheres using octonions (i.e. 21i21j21k21l21m21n21o21 10i10j10k10l10m10n10o10).

You should be aware, however, that the 8-dimensional space of 21i21j21k21l21m21n21o21 has arround 38G positions (38,000,000,000), each having to be able to save an octonion value. That is 8 floating point numbers per position, and Virtue uses 64-bit floating point numbers - defined as "double" in C. Regarding this huge space requirement for a 8-dimensional space with just 21 positions in each dimension it is obvious that there is presently very few computers which would allow such huge memory spaces to be actively used in a calculation!

```
"radius" "centre" "space"
ARGS 3 FUNCTION
               INTERVAL "makes the n-dimensional index space from
                         the 'space' argument"
          SUBTRACT  "the space from the centre position.
                     A 0 will be in this position"
          MAGNITUDE "calculate the magnitude of the indices."
     NOTLESS "put 1's in all positions whose magnitude is inside the radius"
;
@sphere SET.
```

The comments in Virtue are put between the double quotes ("). The comments may be put anywhere (except inside the Virtue words), either inside functions, saved programmes, or during normal interactive work.

Virtue functions are commonly saved in variables begining with a dot (.). This is not enforced, but is common programming practice.

Instead of "ARGS 3 FUNCTION" just "TRIADIC" could be written.

So, without comments:

```
TRIADIC
     SPACE CENTRE MAGNITUDE NOTLESS;
@sphere SET.
```

An example 2-dimensional circle:
==========

```
9 11i11 21i21 sphere EXECUTE.
```

==========

An example 3-dimensional sphere:
==========

```
"Take a radius of" 9 "from a centre at" 11i11j11 "in a space of" 21i21j21
"and make a" sphere! .
```

==========

The "!" is a shorthand notation for the EXECUTE operation.

An example 5-dimensional hypersphere (much too large for rendering):
==========

```
9 11i11j11k11l11 21i21j21k21l21 sphere!.
```

==========

As you noticed, we did not change the defined function ".sphere" in no way whatsoever to get a circle, a sphere and a hypersphere. Although in the three above invocations we used rectangular spaces, there is no reason why we would not ask for something as:
==========

```
91.32 184i122j240 323i242j480 sphere!.
```

==========
i.e a 3-dimensional sphere of radius 91.32 in a space of x 323, y 242 and z 480 places and with the centre at x 184, y 122 and z 240. Virtue does not mind.

A C programme to enable the same functionality provided by the "sphere" Virtue function would necessitate implementation of complex, quaternion and octonion mathematics, very complex pointer arithmetic (addressing of multidimensional arrays), and allocation/deallocation of arrays. The print statements would have to take care of the space rows, columns and planes, to print out a reasonably obvious result.

Due to this quite complicated problem, we will here present a much simplified C programme, which will print only 3-dimensional spheres of radius 9, with the centre at [11][11][11] inside a space of [21][21][21]. 3-dimensions were chosen as to show quaternion arithmetic (in 3-dimensions) and printout control. A generic programme like the Virtue "sphere" function would necessitate a much much more complex C programme.

EXAMPLE 5: Lukasiewicz fuzzy logic implication

```
--------------------------------------
Virtue
DYADIC
      :LOGICAL; DIP LOGICAL
      +
      1 SWAP -
      1 MIN
LOGICAL; OPERATOR @IMPLICATION SET
"18 tokens"
"ready to use:".

INPUT IMPLICATION.
--------------------------------------

--------------------------------------
cat > implication.c
double IMPLICATION (left, right)
      double left;
      double right;
{
      double impl;
      if (right < 0.0) right = 0.0;
      if (right > 1.0) right = 1.0;
      if (left < 0.0) left = 0.0;
      if (left > 1.0) left = 1.0;
      impl = 1 - left + right;
      return (impl < 1)? impl : 1;
}
/* 77 tokens */
/* when used link the produced object file or include the source in the
compilation */
^D

cat > imply.c
#include "implication.c"
#include <stdio.h>

main ()
{
      double left, right, result;
      scanf ("%lf %lf", &left, &right);
      result = IMPLICATION (left, right);
      if (result == 1.0)
            printf ("T");
      else if (result == 0.0)
            printf ("F");
      else
            printf ("%lf", result);
      printf ("\n");
}
^D
cc imply.c -o imply
imply
--------------------------------------
```

(A Very Personal) Introduction

Once upon a time, and it was late year 1974, I had the opportunity, happily taken, to start learning how to program in BASIC on the (then very modern) HP 2000F Timesharing system. From that time on two of my main interests are Compter Science and Cybernetics. Very soon I was quite good in the Art of Programming.

The oldest computer I ever programmed was an “outer programme”, that is patch-panel wired, relay based computer developed by my friend and teacher Branimir Makanec back in 1962. This computer (called “The Digital Simulator”) was specifically targeted towards Computer Aided Learning.

The second oldest compter I used, programmed and serviced was an Eurocomp LGP 21, a fully transistorised small office computer, with a Flexowriter and punched tape input/ output. The main memory was on a rotating chromed plate, and the trick to get programmes running as quickly as possible was to calculate the time the platter needs to rotate to the next instruction.

BUGS and necessities

```
5 .. 0i1 * ENCLOSE 5 .. +.
```

RESHAPE shall work with positive multidimensional numbers as arguments in addition to positive integer vectors.

Ideas how to programme something

Column major index space:

```
5i5 .. .
```

Row major index space:

```
5 .. 0i1 * ENCLOSE 5 .. + ENLIST (5 5) RESHAPE.
```

MONADIC DECAPSULATE

Zorislav Shoyat

A (very) preliminary

VIRTUE

Language Reference Manual

Zagreb, 13/10/2014

Virtue objects (data and spaces)

Data

Virtue keeps a lot of information on the content of the data it works on. In addition to the data on how a specific space looks like, its dimensions, rank, depth, subspaces etc., each individual element of a space, a scalar, can be of different type, and also has specific subtypes.

The Virtue main data types are: ASCII, INTEGER, REAL, COMPLEX, QUATERNION and OCTONION.

The Virtue subtypes are: BOOLEAN, LOGICAL, BINARY, NUMBER, CARTESIAN, POLAR.

Binary data

Beware that BINARY characters, integers and reals (floating points) are BINARY in the exactest sense of the word. This means that all boolean/logical operations will treat them as pure binary data. This is very powerfull, as for example floating point numbers can be manipulated directly in binary form. However, the BINARY is not provided for this kind of (possible) trickery. Binary data is important for e.g. pixel data manipulation, or specific character manipulations.

There are three datatypes which may be typecasted into BINARY. The ASCII is always 8 bits, the INTEGER has, depending on the 32/64-bit computor implementation, 32 or 64 bits. And finally, BINARY real has normally 64 bits (ISO double), but Virtue may be compiled for 32 bit floating point (ISO float), in which mode certain processors work much faster (i.e. on the x86 the SSE instruction set etc.). So a BINARY REAL can have 64 or 32 bits.

However, in operations which are not boolean/logical, any BINARY number will behave as a normal number. This means that if you change INTEGER bits 31 and/or 32 (63 and/or 64), the non-converted, recasted INTEGER will have a negative value, or will be recasted into a REAL (not BINARY!). The same is true for REAL numbers, where BINARY shall be used with a lot of precausion! Meddling with BINARY INTEGER bits higher then bit 31 (starting from 1) is directly implementation dependent, as is any meddling with the REAL bits.[1]

On the other side, BINARY ASCII data is fully compatible with the 8-bit extended ascii character encoding: control characters {[0..31], 127}, interpunctions {[32, 47],[58, 64],[91, 96],[124, 126]}, numbers [48, 57], capital letters [65, 90] and small letters [97, 123], and font dependent area [128, 255]. Therefore any BINARY data operation on the ASCII will be performed inside this range and is guaranteed to generate a printable or control character. This can be very convenient for any kind of preparing data for display, be it a vector display like the xterm supported Tektronix 4014, or a binary data file.

1.The philosophy of the Virtue language is that all constructs are fully portable and mathematically/ logically interconnected. The BINARY data is the only exception from this general philosophical rule. As explained in text, BINARY ASCII is not really breaking the datatypes interconnectedness. The reason BINARY INTEGERs, and even “worse” BINARY REALs are introduced into Virtue is to enable some low-level binary work. USE WITH CAUTION!

Mathematical operations on BINARY INTEGERs and REALs are done in the conventional way, as if BINARY would be _GENERIC_. Boolean/logical operations are done binary.

Mathematical operations on ASCII (characters) are not possible. However, on BINARY ASCII simple mathematical operations are performed (e.g. negative, subtraction, addition, multiplication, division...). See each operation for applicability.

Binary data can be directly inputed, and is SAVEd and WRITEd (written) in the form 0#X, where X is a string of zeros and ones. Depending on the length of the digit string, it will be saved in ASCII if up to 8 bits, in INTEGER if up to 32 bits. Longer bitstrings are saved depending on the particular computor/implementation (32/64 bit INTEGERs, 32/64 bit REALs). So, for example 0#1011010 will be the ASCII character 'Z', whereas 0#101101000 will be the INTEGER number 360 (like in 360 degrees).

Additionally, in the same way as in the PostScript language, a number base can be written instead of the 0, i.e.: 10#27 3#1000, 2#11011 and 0#000011011 are the same integer, the same as 0#11011 NUMBER.

The BINARY character data will be SAVEd and WRITEd (written) as a character string (delimited by the character delimiter '), and the number of the above described form (0#X) will be written inside backquotes `, for example **'UU' BINARY '' WRITE** will write **'`0#01010101``0#01010101`'**.

Numbers

Multidimensional numbers, i.e. complex, quaternion and octonion numbers may be written shortened to the important parts. That is, the number 0i0j0k0l0m0n0o1 may be written as 0o1, or, for example, 1k2m3o4 is the number 1i0j0k2l0m3n0o4. All multidimensional numbers inputed by the user can also be preceded by 'r', for the 'real' part. That is, the number 1i2 can be inputed both as 1i2, and as r1i2. The abovementioned numbers can be written as r0o1, r1k2m3o4 etc.

General multidimensional numbers, i.e. numbers indicated as NUMBER subtype may be directly imputed by prefixing the multidimensional (i,j,k,...) part by #, that is #1i2j3 is the same as 1i2j3 NUMBER. The WRITE, SAVE etc. will write these numbers in such a way, so the subtype is preserved when reading them again. The same applies for memoisation, that is 1i2j3 is not the same value as #1i2j3.

Polar coordinates

Virtue recognizes polar coordinates. Their angular values are always in radians.

Due to the fact that Virtue spaces are multi-dimensional, polar coordinates are written in such a way that the "distance", i.e. the straight line coordinate, the radius, is the first (i.e. the one which would be the "real" part of a multidimensional number), and all the circular coordinates are in the "imaginary" parts of a multidimensional number. This is consistent with the conventional notation of polar coordinates as (r, φ), or, in multidimensional polar coordinates as $(r, \varphi_1, \varphi_2, ..., \varphi_n)$.

In Virtue, polar coordinates are written as classical multidimensional numbers, but instead of i...o, p..v are used. That is: (12, 3.14) would be written in Virtue as 12p3.14. A multidimensional polar coordinate, like (10, 1, 2, 3, 3, 2, 0, 1) would be written as 10p1q2r3s3t2u0v1. The unnecessary polar coordinates can be elided, as for any multidimensional number.

The same as for writing the multidimensional numbers in cartesian coordinate system, the polar numbers allow the 'radius' to be explicitly noted by a 'r' preceding the number, i.e. r12p3.14, r2v1 etc.

When doing mathematical operations on polar multidimensional numbers, as e.g. adding, subtracting..., the polar coordinate(s) (the polar multidimensional number) will be converted to the cartesian form (classic multidimensional numbes, which, in Virtue, always represent cartesian coordinates when working with/on spaces). If a monadic operation is done on polar numbers, the result will be polar. If a dyadic operation is done on arguments which are both polar, the result will be polar, otherwise the result will be of the subtype of the left argument. That is **`1p1 2i2 ADD`** will give a polar number, **`2i2 1p1 ADD`** will give a cartesian number.

Beware that no operations on polar coordinates will be done without their conversion to cartesian and back, which means that the standard cartesian space operations are done, except that the argument(s) to the operations are in polar form. There are certain mathematical operations which are done much faster in POLAR than in CARTESIAN form, and these optimisations are inbuilt in the Virtue interpreter. For example the MAGNITUDE operation on a polar number always gives the "real" part, i.e. the radius, whereas with cartesian coordinates Pythagora's formula for hypotenuse must be used. Therefore for an octonion a cartesian MAGNITUDE necessitates 8 multiplications, 7 additions and a square root, whereas in polar notation the result is just copied from the "radius" part of the number. Unfortunately, most of the mathematical multidimensional operations are defined only in cartesian form (e.g. addition, subtraction), and generally therefore keeping coordinates in polar form is more time-expensive than keeping them in cartesian form.

Scalars, Arrays and Spaces

All Virtue data is kept as Scalars, Arrays or Spaces.

A *scalar* is any individual datum, like a character: **`'a'`**; number: **`42`**, **`0i3.14`**, **`1k8`**, **`1k12n24`**; address: **`@somewhen`**; function: **`:;`**, **`ARGS 7 FUNCTION CLEAR;`**; empty space: **`#NIL`**, **`()`**, or even anything in a zero-dimensional array like **`(1)[0 0 0 0 0]`**, and a subspace: **`('Hello world')`**, **`((1 2 3) 4 (5 (6 7 ((8))) 9 (10)))`**.

Arrays are all data which are not scalars. They are kept in so called "column major" order, i.e. the first index is the leftmost one.

An array can be one-dimensional, therefore we often call it a *vector*:

```
(1 2 3 4 5)
```

it can be two dimensional, when it is often called a *matrix*: **`20 INTERVAL (5 4) RESHAPE.`** will give:

```
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
16 17 18 19 20
```

or it can be any-dimensional (see the following 5-dimensional example). Virtue does not impose any limit on the number of dimensions an array may have. However, beware that the amount of individual data grows with each dimension exponentially. For example, an 8-dimensional array of just 10 elements in each dimension has actually all together 100,000,000 (one hundred million, 100M) individual data elements. The same-dimensional (8) array of 20 elements per axis has already 25,600,000,000 (twenty five billion and six hundred million, 25.6G) individual elements. Now if we would ask Virtue to make such an index space of octonions with the following sentence: **`20i20j20k20l20m20n20o20 SPACE.`**, we would actually require from our compter(s) to make an array of 25.6 billion octonion numbers (and each octonion number consists of 8 numbers, each kept in the computer as an floating point number, commonly in 64 bits, i.e. 8 bytes). So 20 elements in each of 8 dimensions times 8 numbers per element times 8 bytes per number: **`20 8 ** 8 * 8 *.`**, is 1,638,400,000,000 bytes (that is one trillion six hundred thirty eight billion four hundred million bytes, i.e. around 1.5 TiB, terabytes), not counting any computing necessary overhead (Virtue has to know about each individual datum what it is!). Now try to multiply two of such arrays! Therefore yes, Virtue allows any number of dimensions in an array, but beware of the space consequences.

The order in which Virtue arrays are conceptually arranged can be seen in this example: **`(5 4 3 2) DUP PRODUCT SPACE SWAP RESHAPE.`** will give an array of 2 groups of 3 groups of 4 rows in 5 columns, i.e. 5 columns of 4 rows in 3 superrows in 2

hyperrows. As a four-dimensional space this array has the length of 5 in the x-dimension, length 4 in y-dimension, 3 elements in z-dimension and two items in t-dimension:

```
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


61 62 63 64 65
66 67 68 69 70
71 72 73 74 75
76 77 78 79 80

81 82 83 84 85
86 87 88 89 90
91 92 93 94 95
96 97 98 99 100

101 102 103 104 105
106 107 108 109 110
111 112 113 114 115
116 117 118 119 120
```

Regarding arrays as virtual representations of geometrical spaces, the first two dimensions, (5 4) in the above example, represent a plane. Contrary to normal geometrical notation where the complex (second dimension) axis looks upwards, in Virtue the second axis looks downwards. All higher dimensions are represented on the screen as two-dimensional slices, divided by dimension-increasing whitespace.

A *space* in a broader sense is anything in Virtue. Even a scalar is a kind of space. However, in a more specific sense, we can talk about a *space* to be an array with sub-structures. Virtue allows any complexity of any element of any array. So a character string may be combined with some numbers to form an array which is part of a vector of different complex and substructured arrays which is again a part of a multidimensional array which is actually a scalar (any substructure in Virtue is a scalar, and behaves as such).

Virtue memory

Names and variables

Virtue distinguishes two types of memory entities: the names and the variables.

The names are composed of any string of digits, letters, interpunctions and even spaces (for such purposes use the AT, GET and DO word family). They reside in different levels and different speed/size type name tables.[1] Virtue names have meanings.

The variables are exclusive to each individual invocation of a functional sentence, and are composed of the character **&** and a string of digits. They reside on the local function invocation stack, and are not available even to functional sub-sentences defined inside the functional sentence.

All names are kept in dictionaries.

Virtue can save all data in memory.

Memory assignment:

@_AAA, @.AAA, _AAA, .AAA not local ... search previous symbol table stack entries. If not found, put in global (lowest level) symbol table

@AAA, AAA local or global ... search previous symbol table stack entries. If no found, put in local (top) symbol table on stack

@%AAA, %AAA local label (top symbol table on stack)

Data for RESULT are always in the basic table.

Data for AT and GET are the same as @AAA, AAA, insert on local (top) symbol table stack entry.

1.There are two internal types of name tables: the Balanced Binary Tree and a huge Hash Table. The Balanced Binary Tree is very good for shorter names, as we usually use them, therefore a more compact name-space. The Hash Table is particularly usefull for the memoisation, as extremely long names can be formed by the RESULT verb, and for any use when a “randomised” large name-space is necessary. The NEW CONTEXT can explicity define which of the two algorithms shall be used.

search global, set global (e.g. XXX - the name is XXX)

search all, set global - (e.g _XXX - the name is XXX)

search local, set local - very local (e.g. .XXX, %XXX, the name is .XXX, %XXX)

search all, set local - common nested structure (e.g. $XXX, the name is XXX)

stack based - (e.g.. L1)

'.' and '%' are part of the name, they keep very local.

'', '$' and '_' are not part of the name, they are modifiers.

no mod: search global, insert global	- SGIG	name means 'The name'
'_': search all, insert global	- SAIG	_name means 'newest name'
'$': search all, insert local	- SAIL	$name means 'local name'
'.', '%': search local, insert local	- SLIL	.name means 'this name'

There are two types of behaviour of names, depending on their prefix. The names can either be fixed to a local or global context, or they can be dynamically scoped.

In dynamic scoping the name is searched always from the local name through layers of functional/context invocations down to the basic, global memory level. Everytime such a name is invoked, Virtue will search for it on all symbol table stack levels.

In fixed scoping the name (with it's possible prefix) is fixedly defined to be either local or global. Beware that local names are not local variables (see further in text). As they are fixed to a specific symbol table, these names can be already interpreted at the lexical level, so that they are assembled in the programme with their addresses, and are therefore much faster in execution than the dynamic names.

There are two type of behaviour of addresses. They may either place the name in a local or in a global symbol table, if the name was not found according to the name search principles explained above.

As the '.' prefix indicates a local context, it is not possible to use those memory names in user communication, except inside sentences. As soon the sentence is finished (executed), the SLIL variables are not existing any more.

All invocations of the same function data share the same memory. Therefore all name, .name, %name and $name are certainly preserved between completely unrelated invocations of the same function (as long as the original is kept). Only the &xx numeric stack variables are completely local to each invocation of a function. In other words, each assembled function, that is a function which can be manipulated as data, keeps with itself its own private memory, which is global for all invocations of the same function (data item). The "local" names ('.') have their own name space, which is always local, even when directly at

the user interaction level. So, in the example 1 @a ASSIGN .a ADD. Virtue will complain that .a is not defined, although the statement was inputed at the user interaction level. So for each individual sentence the “local” (search local, insert local) are completely independent of all other name dictionaries.

All RESULT invocations keep their data in the global memory, and all functional sentences on any level can use them.

If a variable address is saved somewhere, it will always point to the name in the symbol table from which it originally took it, independent of the context in which it is presently used.

Shorthand representation

Numbers:

NaN	**#X**
Inf	**#Z**
-Inf	**#-Z**
NUMBER	**#N**
BOOLEAN	**#B**
LOGICAL	**#L**
BINARY	**#b**

Virtue verbs (operators)

Input-Output

INPUT => <data>

QUADIN

get user input and execute it. Leave the result on stack.The user input is always finished when the "enter"/"return" key is pressed, and processing continues. Except for the noted differences, INPUT behaves in the same way as the Virtue prompt.

The user may type several Virtue sentences on input, where each sentence is terminated by the end-of-sentence word '.'. However, INPUT will not allow the user to use the data which is already on the stack, so it is not possible to do something like "2 INPUT" and then inputing a sentence "2 ADD". Every INPUT starts with a fresh stack frame, so the above will raise the error "800 - not enough arguments on stack for the reqested operation.", as "2" is not enough arguments for the "ADD" (which is dyadic).

INPUT will not allow the word "OFF" to exit the whole Virtue system.

The result of INPUT must always be exactly one (1) data element. If no result of the input is left on the stack, a #NIL is left on stack instead. INPUT must leave exactly one element, as otherwise the enveloping sentences (the Virtue programme) would be confused. If the user leaves more than one data element on the stack, the warning "700 - Warning: INPUT (QUADIN) returned more than 1 result on stack. Discarding all but the first (deepest) element." is emitted, and all but the first data element left on stack by the user is discarded.

TEXTINPUT => <data>

INPUTTEXT, QUOTEDINPUT, QOTEDIN, QUOTED, QUOTEQUAD

get user input as a quoted string. Leaves an ASCII vector on stack. The QUOTEDINPUT finishes with the "enter"/"return" keypress. If no text is given, an empty ASCII vector (''), which is equivalent to #NIL.

VARIABLES => <structured ascii>

VARS

make a structured ASCII space of all Symbol Table entries which start with '_'. Each name will be entered into the ASCII space as a scalar ASCII vector.

Note: there is no way to list the names of data saved without the '_' prefix. Data saved without this prefix is supposed to be kind of "constant" or "internal" to the user programme.

As defined, VARIABLES will give names of all Symbol Table entries starting with the character '_'. As this naming is only a convention, functions and OPERATORs can also be listed in the resulting ASCII space, depending on the naming conventions used by the user.

FUNCTIONS => <structured ascii>

FUNCS, FNS

make a structured ASCII space of all Symbol Table entries which start with '.'. Each name will be entered into the ASCII space as a scalar ASCII vector.

Note: there is no way to list the names of functions saved without the '.' prefix. Such FUNCTIONS (and OPERATORs) are supposed to be "internal" to the user programme.

As defined, FUNCTIONS will give names for all Symbol Table entries starting with a '.'. As naming functions to start with '.' is just a convention, FUNCTIONS may give names of data which are no functions, but start with this character.

<data> OUTPUT => <data>

TOS

output in "human readable" form the data on Top of Stack. This is the same type of output as done by Virtue itself at the end of a processed sentence. This output is formatted according to the dimensional view of the data, and no shape reference is outputed. Therefore this form is not suitable for direct reading by Virtue (as opposed to WRITE), due to the fact that some important structure size and form is not outputed.

OUTPUT leaves the data on stack.

<data> PRINT =>

QUADOUT

output in "human readable" form the data on Top of Stack. The printing format is identical to that of OUTPUT.

PRINT discards the printed data from stack.

'file' READ => <...>

read a file with the name specified by the ASCII vector on Top of Stack. READ reads and performs all the sentences in the file (each read as if INPUT was used on the file). When reaching the End of File, the input is returned to the user. The file must be positioned in the directory from which Virtue was started or below (deeper in the directory structure). If it is wanted that READ can open files in any directory available to the user, Virtue has to be started with the '-u' option ("unconstrain directories").

'file' TEXTREAD => 'character vector'

READTEXT

reads a file from begin to End of File as one single ASCII vector. The directory constraints are the same as for READ.

'file' SAVE =>

saves the whole session to a named file in human/Virtue readable form. First saves the whole content of the storage (i.e. variables, memoised data, functions etc.), then the whole stack. READing this file in a new session will restore the whole Virtue state. The directory constraints are the same as for READ. The data is outputed in full format, that is in the format the Virtue input processor understands, as it is in WRITE.

SAVE may be used inside any Virtue sentence on any recursive level.

'file' LOAD => <...>

loads a fresh saved session. 'x' LOAD behaves like *CLEAR WIPE 'x' READ*. Beware that inside a function, in recursion, the implicit CLEAR of stack in LOAD will clear only the topmost stack frame, i.e. the stack frame at the present recursive level. So, LOAD inside a function will clear all stack elements so far left on the stack by the function. However, it will WIPE clean *all* global variable memory, i.e. saved data (variables, functions, memoisations...) and reset the environment to the saved state. The directory constraints are the same as for READ.

'file' DUMPSTACK =>

DUMP

save everything on the stack to the file named. The stack and machine environment remain unchanged. The directory constraints are the same as for READ. The stack is dumped in full format, as in WRITE.

'file' REMEMBER =>

save everything in the variable memory (variables, functions, memoisations...) to the named file. The machine environment is left unchanged. The directory constraints are the same as for READ. The variable memory is outputed in full format, as in WRITE.

<data> 'file' WRITE => <data>

write the data on the stack to the file. The data is written, as opposed to normal Virtue output to the user, in full format, i.e. highest number precission available, and spaces are outputed in a vector format with the "[xxx]" shape definition at the end, in the same way as Virtue interpreter expects them to be inputed by the user. To get the user interface output of this precission and format, use a #NIL, or an empty character vector ('') for the file name.

<data> 'file' APPEND => <data>

append the data from the stack to the end of an (possibly) existing file. The data is written if full format, as for WRITE.

<data> 'file' TEXTWRITE => <data>

WRITETEXT

write the data from the stack to a file, but in a user-interaction format, the same as OUTPUT, not the full format of WRITE.

<data> 'file' TEXTAPPEND => <data>

APPENDTEXT

append the data from the stack to the end of an (possibly) existing file. The data is writen in the user-interaction format, the same as OUTPUT.

<structure> <file> OPEN => <file>

MSB OPEN

<structure><file>LSB OPEN => <file>

The OPEN verb opens a file (specified by the <left> argument) for reading data which shall be provided in the form of <dataspec>.

The <left> argument can be either a character string, or an already opened file.

If the <left> argument is a character string, a file so named will be opened. Beware that different operating systems differ in their file naming conventions. Try to use a kind of lowest common denominator.

If the <left> argument is a file “HANDLE”, the specified file will not be reopened, its position will not be changed, but a new specification for the data to be read will be used.

The <dataspec> (right) argument can be either a simple nonzero positive integer or a character string.

If the <dataspace> argument is a nonzero positive integer between 1 and 8 (on 64-bit computers) or 1 and 4 (on 32 bit computers), the OPEN will open a binary file. The specified number in the argument will be taken as the number of bytes to be read at once into one Virtue datum. Length 1 data will be read as the type _BINARY_ ASCII, and the sizes between 1 and 4/8 will be saved as _BINARY_ INTEGER. On a 32-bit integer 64-bit floating point (with the integral precision of 53-bits), the 5 and 6 byte reads will be read as REAL numbers without information loss. 7 and 8 byte reads will be converted to REAL numbers with certain information loss.

If the <dataspec> argument is a negative integer between -1 and -8, then _BINARY_ floating point data is read. The <dataspec> argument indicates the dimensionality of the numbers to be read. 1 is for REAL, 2 for COMPLEX, 3-4 for QUATERNION and 5-8 for OCTONION. Unused dimensions will be set to 0.0. For writing the opposite applies, and the unused dimensions will not be outputed to the file.

If the <dataspec> argument is a character string, “formated” reading/writing will be defined for the file. The specification of the “format” character string which is in the format accepted by the C language scanf/printf library functions. This format will be used both for reading and writing to the specified file. However, due to the fact that the scanf and printf formats are not identical (though for simple things very similar), The <dataspace> argument may also be a structure containing two character strings, the first for the input format (scanf type) and the other of the output format (printf type).

Please beware that when using the character string format specifier in <dataspec> in specific gray areas there may be difference between different operating systems. Do not use obscure format features.

If the file specified is not existing, it will be created with read/write permissions.

<file> CLOSE

closes an open file, or a structure of files at the operating system level. This does not mean that the file, if its file “HANDLE” is preserved in the workspace, can not be reopened without the OPEN command. A CLOSE acts as a flush (commiting all data to physical computer device), and as a REWIND. Therefore it may be used whenever necessary.

CLOSE will work on any type of space (vector, array or even subarrayed space) which consists only of FILEs. It will attempt to close them all.

As file HANDLEs can be internally manipulated as any other data, it is possible to have complex structures of any combination of data with HANDLEs. However, CLOSE will only work on those spaces which consist exclusively of file HANDLE data.

<file> <interval> DATAIN

DATAREAD

will read data from <file> in the form specified by OPEN and put it into the space defined by the <structure> argument of the OPEN command. The file pointer is left on the position after the last data read.

All data is read from files as binary in Network Byte Order. This means that the first byte of a multi-byte entity (like an INTEGER, which, depending on the host system architecture may have 4 or 8 bytes) is the most significant byte of the entity (e.g. number). This is called the Most Significant Byte (MSB) first. For example, the number .....,

If you need to read data from files which you know are in the other, Least Significant Byte (LSB) first, please specify the adverb LSB when opening (or reopening) the file. If you have a file of mixed byte order, you can always reopen the file with the other order (OPEN is for MSB, LSB OPEN for LSB) and seek back to the position in the file you were on.

As long as the file “HANDLE” is preserved in any place in Virtue, another DATAIN will automatically reopen the file as if performing a REWIND.

<left> <file> DATAOUT => <data>

DATAWRITE

will write the <left> argument to the file <file> in the format specified by the OPEN verb.

If the format specified in OPEN can not be applied to the data contained in <left>, the data will be written until the first element which can not be written.

All data is written to files as binary in Network Byte Order. This means that the first byte of a multi-byte entity (like an INTEGER, which, depending on the host system architecture may have 4 or 8 bytes) is the most significant byte of the entity (e.g. number). This is called the Most Significant Byte (MSB) first. For example, the number .....,

If you need to write data to files for which you know they have to be in the other, Least Significant Byte (LSB) first, please specify the adverb LSB when opening (or reopening) the file. If you have a file of mixed byte order, you can always reopen the file with the other order (OPEN is for MSB, LSB OPEN for LSB) and seek back to the position in the file you were on.

If writing in _BINARY_ format (the <dataspec> is numeric - see “<structure> <file> OPEN => <file>” on page 17), all data will be written in binary format, including all REAL, COMPLEX etc. numbers. Multidimensional numbers will be written without any indication of their “physical” boundaries. Beware that 64-bit floating point and 32-bit foating point architectures are not compatible if writing these numbers. The same applies to 32-bit and 64-bit integers on respective computer architectures.

<integer> <file> SEEK

will position the file pointer relatively to the present position by adding to it the INTEGER argument <integer>. Positive numbers will advance the file towards the end, negative will go backwards towards the begining.

If the positive <integer> argument would cause the file pointer to pass the end of file, the file pointer will be positioned at the end. If the negative <integer> would cause the file pointer to get before the begin of the file, the SEEK will behave like a REWIND.

If you need to get to a specific absolute address within a file, use REWIND before SEEK.

Beware that certain types of file which may be opened, like device files or pipes, may not permit seeking. The SEEK will complain.

<file> REWIND

rewinds the file. If it would be a tape-drive (not modern nowadays), it would actually be rewound. A disk file will just get its pointer at the begining.

REWIND will work on any type of space (vector, array or even subarrayed space) which consists only of FILEs. It will attempt to close them all.

As file HANDLEs can be internally manipulated as any other data, it is possible to have complex structures of any combination of data with HANDLEs. However, REWIND will only work on those spaces which consist exclusively of file HANDLE data.

Monadic scalar

RULES:.....

<left> CONJUGATE => <result>

the result of the mathematical *conjugation*. For INTEGER and REAL it is a no-operation. For multidimensional numbers CONJUGATE changes the sign of all dimensions except the first one (i.e. the sign of the imaginary parts of the number, the real part unchanged).

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> NEGATIVE => <result>

the result of *0.0 - <left>* , for all dimensions of the numbers. Makes all positive numbers negative, and negative positive.

works on: BINARY ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> SIGNUM => <integer: -1,0,1>

SIGN

gives the sign of the argument, negative (-1), zero (0) or positive (+1). For INTEGER and REAL numbers, the SIGNUM gives the sign of the number (positive/negative/zero). However, for COMPLEX numbers there are several ways of calculating the signum. For consistency the csgnz() method is used. The result is zero (0) if both the real and imaginary parts are 0. The result is positive (+1) if the real part is positive, or the real part is zero, and the imaginary part is positive. The result is negative (-1) if the real part is negative, or the real part is zero, and the imaginary part is negative.

The sign of the quaternion and octonion numbers is mathematically not yet strictly specified, and is generally not taken as something applicable. However, Virtue strives towards full orthogonality of single-/multi-dimensional numbers. Therefore the SIGNUM is extended to QUATERNIONs and OCTONIONs in the spirit of COMPLEX numbers. This is done by extending the principles of the csgnz() function.

Generically the SIGNUM operation on multidimensional numbers gives the positive (+1) result when the first (from left to right, i.e. from real towards imaginary parts) non-zero dimension of the number is positive, and the negative (-1) result if the first non-zero dimension of the number is negative. It returns zero (0) if all dimensions of the number are zero.

To keep consistency, the SIGNUM verb will convert a POLAR multidimensional number first to CARTESIAN, and than perform the calculation. This is due to the fact that the above functions (csgnz, csgnq, csgno) are defined in rectangular (cartesian) space.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> DIRECTION => <result>

gives the unit vector (dimensional number) of the argument. For INTEGER and REAL, one-dimensional numbers, the results can only be -1, 0, and 1, the same as for the SIGNUM function. For multidimensional numbers the result are the coordinates of the point in the multidimensional space which defines the direction towards the specified argument number, and which point is exactly one (1) dimensionless unit away from the zero point (0).

For an anydimensional number (trivially for one-dimensional) the DIRECTION is actually the defined argument (<left>) point divided by the length of the hypotenuse (distance) from the zero point of the space to the argument point. The sign of the individual dimensional result is the same as the sign of the corresponding argument element.

For _POLAR_ numbers the DIRECTION is the same for all “imaginary” parts, with r (the radius) being 1.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> RECIPROCAL => <result>

the reciprocal of the argument, i.e. *1.0 / <left>*, for any-dimensional argument.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> MAGNITUDE => <result>

ABSOLUTE, ABS

the magnitude of the argument. The magnitude is the any-dimensional hypotenuse, i.e. the distance from the zero point of the space to the argument (<left>) point. For INTEGER and REAL numbers this is trivially the ABSOLUTE value, i.e. *|<left>|*. For COMPLEX, QUATERNION and OCTONION numbers it is the square root of the sum of the squares of each dimension.

the MAGNITUDE multiplied by the DIRECTION of any argument is the argument itself, i.e.: *<argument> DUP DUP MAGNITUDE SWAP DIRECTION MULTIPLY IDENTICAL* is always true (except for numerical processing errors).

For _POLAR_ numbers the MAGNITUDE is always r, as in polar coordinates we have the radius, which is actually the cartesian hypotenuse to the defined point. Therefore in _POLAR_ coordinates it is quite obvious that the argument is identical to its MAGNITUDE multiplied by its DIRECTION.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> FLOOR => <integer result>

the floor, i.e. the biggest integer smaller than or equal the argument. For example, the floor of 1.1 is 1, the floor of -1.1 is -2.

For INTEGERs, as they are anyway integers, the operation is trivial - the result is the same as the argument.

For multidimensional numbers the floor is quite complex. For COMPLEX numbers: if the sum of the distances from the real number in the real part (r) to the corresponding integer of the real part (floor(r)), and the real number in the distance of the imaginary part (i) to its corresponding integer (floor(i) is less then 1.0 (i.e. *r - floor(r) + i - floor(i) < 1.0*) than the complex floor components are r = floor(r) and i = floor(i). Otherwise, if r - floor(r) is bigger then i - floor(i) then the resulting r = floor(r) + 1, and i = floor(i), else r = floor(r) and i = floor(i) + 1.

In other words, if the sum of the differences between the real number and its integer floor in all dimensions is smaller then 1, then all the dimensional parts of the result are floors of the of the corresponding argument parts, otherwise the result for the dimensional part with the biggest difference between the real number and its integer floor is 1 + floor(x).

This approach to COMPLEX, QUATERNION and OCTONION floor is the same as taken by Sam Sirlin for his apl compiler (from aplc-5.12).

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> CEILING => <integer result>

the ceiling, i.e. the smallest integer bigger than or equal to the argument. For example, the ceiling of 1.1 is 2, the ceiling of -1.1 is -1.

For INTEGERs, as they are anyway integers, the operation is trivial - the result is the same as the argument.

For multidimensional numbers CEILING is equal to: <left> NEGATIVE FLOOR NEGATIVE. See the description of FLOOR.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> TRUNCATE => <integer result>

INTEGER, TRUNC

The result of this verb is a the truncated integer value of the argument. As opposed to FLOOR and CEILING, integer is a truncating operation in all dimensions of the argument numbers. This means that COMPLEX, QUATERNION and OCTONION numbers will be truncated to integer arguments in all dimensions, as if they would be completely inependent. Truncation, the verb INTEGER is actually rounding towards 0.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> EXPONENTIAL => <result>

EXP, eX

the exponential function $e^{<left>}$.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> NATURALLOG => <result>

LOGE, LOGe, LN

the natural logarithm function $log_e<left>$.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> PITIMES => <result>

PI*

multiply <left> by pi (3.14159265358979323846 for double precision implementations). For the number pi itself, it is possible to use either 1 PITIMES, or directly the constant **#PI** (much faster, as it is a constant).

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> FACTORIAL => <result>

the factorial function.

For positive INTEGERs the factorial function is defined as the product of all positive integers up to <left>. It is equivalent to <left> INTERVAL PRODUCT. The Virtue FACTORIAL is precalculated for integers 0 to 12 (on 32 bit implementations) or 0 to 20 (on 64 bit implementations). [20 FACTORIAL is 2432902008176640000.] For bigger arguments the gamma(<left>+1) is calculated, as for non-integer singledimensional numbers.

For all negative INTEGERs the factorial function is undefined, and will raise the complaint (error): "36 - bad argument to factorial".

For all other singledimensional numerical values the result is the result of the gamma function on <left> + 1, i.e. *gamma(<left> + 1)*.

For COMPLEX numbers the FACTORIAL calculates the complex gamma function. The Virtue implementation of this function is a translation from FORTRAN to C of the complex gamma function taken from the "Collected Algorithms from CACM," algorithm 421, by Hirondo Kuki. The C translation was done by Thomas Glenn Smith for his cAPL interpreter.

Unfortunately, the algorithms for QUATERNION and OCTONION factorials (quaternionic gamma and octionionic gamma functions) are not implemented yet. The mathematical scholars are much investigating into them lately. Possible implementation will be done in a later stage of Virtue development.

A _POLAR_ COMPLEX number is first converted to rectangular (_CARTESIAN_) format, and than the complex gamma function is applied. The result of FACTORIAL is always sub-typed as _NUMBER_.

works on: INTEGER, REAL, COMPLEX

<left> SINUS => <result>

SIN

calculates the trigonometric function *sin<left>*. It is equivalent to #SIN CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallely different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> COSINUS => <result>

COS, COSIN

calculates the trigonometric function *cos<left>*. It is equivalent to #COS CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallely different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> TANGENS => <result>

TAN

calculates the trigonometric function *tan<left>*. It is equivalent to #TAN CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallely different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCSIN => <result>

ASIN

calculates the inverse (arc) trigonometric function *asin<left>*. It is equivalent to #ASIN CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCCOS => <result>

ACOS, ACOSIN

calculates the inverse (arc) trigonometric function *acos<left>*. It is equivalent to #ACOS CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCTAN => <result>

ATAN

calculates the inverse (arc) trigonometric function *atan<left>*. It is equivalent to #ATAN CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> SINH => <result>

calculates the hyperbolic trigonometric function *sinh<left>*. It is equivalent to #SINH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> COSINH => <result>

COSH

calculates the hyperbolic trigonometric function *cosh<left>*. It is equivalent to #COSH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> TANH => <result>

calculates the hyperbolic trigonometric function *tahh<left>*. It is equivalent to #TANH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCSINH => <result>

ASINH

calculates the inverse (arc) hyperbolic trigonometric function *asinh<left>*. It is equivalent to #ASINH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCCOSINH => <result>

ARCCOSH, ACOSINH, ACOSH

calculates the inverse (arc) hyperbolic trigonometric function *acosh<left>*. It is equivalent to #ACOSH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> ARCTANH => <result>

ATANH

calculates the inverse (arc) hyperbolic trigonometric function *atanh<left>*. It is equivalent to #ATANH CIRCULAR.

When more trigonometric functions have to be calculated for the same argument, or parallelly different functions for an argument array, please use the dyadic verb CIRCULAR.

Works on: INTEGER, REAL, COMPLEX

<left> SQUARE => <result>

SQR

calculates the square (second power) of the argument. It is equivalent to 2 POWER, that is <left>^{2}.

Presently does not work on QUATERNION and OCTONION data. This limitation will be lifted in future revisions/versions of Virtue.

Works on: INTEGER, REAL, COMPLEX

<left> CUBE => <result>

CBR

calculates the cube (third power) of the argument. It is equivalent to 3 POWER, that is <left>^{3}.

Presently does not work on QUATERNION and OCTONION data. This limitation will be lifted in future revisions/versions of Virtue.

Works on: INTEGER, REAL, COMPLEX

<left> SQRT => <result>

calculates the square root of the argument. It is equivalent to 0.5 POWER, i.e. <left>$^{1/2}$

Presently does not work on QUATERNION and OCTONION data. This limitation will be lifted in future revisions/versions of Virtue.

Works on: INTEGER, REAL, COMPLEX

<left> CBRT => <result>

calculates the cube root of the argument. It is equivalent to 1 3 / POWER, i.e <left>$^{1/3}$.

Presently does not work on QUATERNION and OCTONION data. This limitation will be lifted in future revisions/versions of Virtue.

Works on: INTEGER, REAL, COMPLEX

<left> POLAR => <polar result>

converts CARTESIAN coordinates to POLAR coordinate system. This has sense only on multidimensional numbers, for all others it is actually a slow NOOP.

The verb POLAR will convert all multidimensional types (even BOOLEAN, LOGICAL etc.) mathematically as converting from CARTESIAN, except POLAR, which stays as is. Furthermore, POLAR will not make any conversion when the data has the subtype NUMBER (or is GENERIC).

Both polar and cartesian multidimensional coordinates may be freely intermixed in any Virtue session. Virtue will take care of the necessary conversions when specific operations are mathematically not feasible or even not possible in polar form (as for example the addition and subtraction of multidimensional numbers are possible only in cartesian form). However, even when doing such operations on polar coordinates (as for example adding them), although the calculation itself will be done in cartesian form, the result will be in polar form. There are, furthermore, some operations on polar coordinates which are much more simple in polar form (as e.g. the multiplication, division, the magnitude or the direction). When having a polar argument these operations will automatically be performed in the faster polar form.

For dyadic operations of mixed polar and cartesian form the result will be in the form of the left (first) argument. So for example **`1p1 1i1 ADD`** will result in the polar number **`2.400738757p0.8742202771`**, whereas **`1i1 1p1 ADD`** will result in the cartesian number **`1.540302306i1.841470985`**, which are mathematically identical. When both arguments to a dyadic operation are polar, the result will be polar. **`1p1 2p2 ADD`** gives **`2.676043577p1.68012709`**.

POLAR is the inverse operation of CARTESIAN. So **`1i1 POLAR`** is **`1.414213562p0.7853981634`**. If you need to typecast a cartesian coordinate into a polar coordinate, as for example when you inputed a coordinate in cartesian form and actually wanted to input it in polar form, first recast the cartesian number into NUMBER: **`1i1 NUMBER POLAR`** gives **`1p1`**.

Polar coordinates are indicated, as already noted, by the infixes ‘p’, ‘q’, ‘r’... between the coordinate dimensions, and are both inputed and outputed in this form.

Works on: COMPLEX, QUATERNION, OCTONION

Ignores: INTEGER, REAL

<left> DEG => <left>

DEGREES

In Virtue all polar numbers can be expressed in degrees or radians. The DEG verb defines that the polar coordinates indicated (complex, quaternion, octonion) will be expressed in degrees. It is important to note that the DEG is a monadic operator, which means that it operates on the <left> argument. This means that the result will be the argument expressed in degrees. The DEG verb sets the _DEG_ subdetail in each multidimensional number in the argument. This flag, indicating that the number is to be expressed in degrees will persistently be kept throughout the life of that particular number. So, for example, **`1p0.7853981634 1p1.570796327 CATENATE DEG`** will give **`(d1p45 d1p90)`**. However **`1p0.7853981634 DEG 1p1.570796327 CATENATE`** will give the two element vector **`(d1p45 1p1.570796327)`**!

As already noted, multidimensional coordinates expressed (in input and output) in degrees will have the prefix ‘d’, and have to be polar (i.e. they are written with the ‘p’, ‘q’, ‘r’... infixes between coordinates.

Independent of the way the coordinates are expressed (degrees or radians), all Virtue operations will, actually, be done in radians (the purely mathematical form), but the user will not be avare of that.

Works on: COMPLEX, QUATERNION, OCTONION.

Ignores: INTEGER, REAL

<left> RAD => <left>

RADIANS

As in Virtue all polar numbers are expressed either in degrees or radians, the RAD verb will change their expression type to radians (i.e. it will clear the _DEG_ subdetail flag).

Note that, as opposed to for example mathematical calculators which enable either the immediate conversion of a number from degrees to radians, or, some other, set the general polar coordinate “style” to be in degrees or radians throughout the calculation session, in Virtue the RAD verb is a monadic operator, which means that it operates on the <left> argument. This means that the result will be the argument expressed in radians. The way a certain number is be expressed to the user will persistently be kept throughout the life of that particular number. So, for example, **`d1p45 d1p90 CATENATE RAD`** will give **`(1p0.7853981634 1p1.570796327)`**. However **`d1p45 RAD d1p90 CATENATE`** will give the two element vector **`(1p0.7853981634 d1p90)`**!

If not indicated by the ‘d’ prefix, all polar coordinate numbers are in radians.

Independent of the way the coordinates are expressed (radians or degrees), all Virtue operations will, actually, be done in radians (the purely mathematical form), but the user will not be avare of that.

Works on: COMPLEX, QUATERNION, OCTONION.

Ignores: INTEGER, REAL

<left> CARTESIAN => <cartesian result>

converts POLAR coordinates to CARTESIAN coordinate system. This has sense only on multidimensional numbers, for all others it is actually a slow NOOP.

The verb CARTESIAN will convert only POLAR numbers to the cartesian coordinate system. For all other multidimensional types (NUMBER, GENERIC, even BOOLEAN, LOGICAL etc.) there will not be any conversion, but the data will be typecasted into the CARTESIAN subtype.

Both polar and cartesian multidimensional coordinates may be freely intermixed in any Virtue session. Virtue will take care of the necessary conversions, and perform the operation in the form which is mathematically possible, or faster. Generally all of the

operations on multidimensional numbers are done in cartesian form, except as noted in the text about the verb POLAR. The cartesian form is mathematically identical with the numeric (non-coordinate) form of multidimensional numbers. Therefore, both _CARTESIAN_ and _NUMBER_ multidimensional numbers have the same mathematics.

For dyadic operations of mixed polar and cartesian form the result will be in the form of the left (first) argument. So for example **`1i1 1p1 ADD`** will result in the cartesian number **`1.540302306i1.841470985`**, whereas **`1p1 1i1 ADD`** will result in the polar number **`2.400738757p0.8742202771`**, which are mathematically identical. When both arguments to a dyadic operation are cartesian, the result will be cartesian. **`1i1 2i2 ADD`** gives **`3i3`**.

CARTESIAN is the inverse operation of POLAR. So **`1.414213562p0.7853981634 CARTESIAN`** is **`1i1`**. If you need to typecast a polar coordinate into a cartesian coordinate, as for example when you inputed a coordinate in polar form and actually wanted to input it in cartesian form, first recast the polar number into NUMBER: **`1p1 NUMBER CARTESIAN`** gives **`1i1`**.

Works on: COMPLEX, QUATERNION, OCTONION

Ignores: INTEGER, REAL

<left> NOT => <logical value>

~

the logical not. As Virtue logic is multivalued Lukasziwicz t-norm, NOT is actually calculated by *1* - *<logical left>*. <logical left> is the argument reduced to the interval [0, 1]. All values of zero or less are regarded as logically "absolutely false", values greater then 1 as logically "absolutely true", and values between 0 and 1 as logically "maybe", i.e. more or less true.

When using the NOT verb on “normalised” numbers in the range [0, 1] the verb NOT may be used as a shorthand for **`1 SWAP SUBTRACT`**, as it’s result is *1* - *<left>*.

The NOT function applied to ASCII gives 1 if the character was ' ' (space) or any control character, and 0 otherwise. (Space is regarded as boolean false, any other character as boolean true).

NOT applied to integer numbers (including INTEGER and integer valued real numbers) gives a boolean (true = 1 / false = 0) value.

Virtue has multidimensional logic. Each numerical part in a multidimensional number is regarded as an independent logical dimension.

For BINARY data NOT gives the binary not of each bit of the datum. Applied to INTEGERs gives one’s complement of the number. That is, 1 NEGATIVE is -1, 1 BINARY NOT is -2. However, as expected, 1 BINARY NEGATIVE is the same as 1 NEGATIVE, i.e. -1. For REAL data the result is implementation dependant (use with extreme caution).

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> BOOLEAN => <boolean value>

the operation BOOLEAN converts any value to the boolean logic type true = 1 / false = 0. Argument values less then 0.5 are converted to #FALSE, values greater then 0.5 to #TRUE. However, the intermediate value of exactly 0.5 (the undecided logical maybe), which in no way can be regarded either as true or as false, is processed stohastically, i.e. the BOOLEAN result will be fully random, either #TRUE or #FALSE, with equal statistical probability. When this case occurs, and BOOLEAN returns a random truth value, it sets the ISPROBABLE data property, as to prevent the memoisation (RESULT, REEVAL...) to store data based on probability. See more about this property at PROBABLY.

Applied to ASCII data, any control character and the space (' ') give #FALSE, and all other (printable) characters #TRUE. In output all boolean values are written as T (true - internally being number 1) or F (false - internally being number 0). T and F can also be used for input. However, if the character data (ASCII) was indicated to be BINARY, the result will be #FALSE only if the numerical value of the character was 0, otherwise #TRUE.

With multidimensional scalar argument BOOLEAN will convert each logical dimension to a boolean dimension.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> LOGICAL => <logical value>

converts any value to the multivalued logical value in the interval [0, 1]. All values of 1 or greater are regarded as 1 (absolutely true), values of 0 and less as 0 (absolutely false), and values inbetween left as is (maybe, probably, more or less true).

For ASCII and INTEGER data the result is always BOOLEAN, #TRUE or #FALSE. ASCII values of space (' ') and all control characters are converted to 0 (absolutely false), and all printable characters to 1 (absolutely true).

As logical values have a (theoretically) infinite number of possible states (restricted only by the precision of the computor internal representation), they are always outputed as REAL numbers in the range [0, 1]. There is no possibility to directly input values which have the _LOGICAL_ subtype. However the _LOGICAL_ values are directly mathematically compatible with all REAL numbers in the inclusive range between 0 and 1, and the user, if checking for their type with ISLOGICAL will always get the correct result.

Multidimensional scalar arguments are converted to multidimensional (independent) logical values.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> PROBABLY => <boolean value>

the operation PROBABLY stohastically converts a fuzzy (multivalued) logical value to a boolean value. For all argument values which are not 0 (absolutely false), converted to boolean #FALSE (i.e. left as 0), or 1 (absolutely true), converted to boolean #TRUE (i.e left as 1), the resulting boolean trueness (true/false) is based on the probability of the event. That means that an argument (converted to logical value) of 0.5 will have a 1:1 chance of being

resolved into #TRUE or #FALSE, an argument of 0.25 will have a 2:1 chance of being resolved to #FALSE, an argument of 0.75 a 2:1 chane of being resolved to #TRUE, etc.

For ASCII, INTEGER and integer valued real numers the result of PROBABLY is always #TRUE or #FALSE, and is exactly the same as if applying BOOLEAN or LOGICAL ooperations. ASCII values of space (' ') and all control characters are converted to 0 (#FALSE), and all printable characters to 1 (#TRUE).

Multidimensional scalar arguments are independently stohastically converted to multidimensional boolean values.

The operation PROBABLY will define the resulting data property as "ISPROBABLE". Operations which deal with memoisation (e.g. RESULT, REEVAL) will not memoise the function if its result was at any stage propagated to have been based on probabilistic processing. The "ISPROBABLE" property prevents memoisation of probabilistic results, as otherwise results based on once calculated probability would be stored permanently, which would cause a cascade of wrong/trivial/nonchanging results in further processing of memoised stohastic algorithms.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> BINARY => <binary value>

The verb BINARY indicates for all single-dimensional data (characters or numbers) that they are to be processed in binary form. This means that all the bits of the datum will be processed by boolean (and the boolean form of logical) operations as a bit-field, i.e. each bit in the data will be processed individually. This is radically different from data which is in not-binary form, where the logical operations will be performed on the whole datum as one individual boolean/logic value.

For example ‘ ‘ NOT will give true, and ‘Z’ NOT will give false, as ‘ ‘ is interpreted as #FALSE and ‘Z’ as #TRUE. As opposed to this, ‘ ‘ BOOLEAN NOT will give an 8-bit character, which is font dependent (e.g. in the font in which this text is written it will be ’ß’, hexadecimal DF, decimal 223), and ‘Z’ BOOLEAN NOT the character coded by hexadecimal A5, decimal 165 (in the font of this text ‘¥’).

When outputed in the “structured” form (as with the WRITE, SAVE etc.), binary data will be in the form 0#X (see “Binary data” on page 6). When outputed to the user they will be non-distinguishable from their non-binary form.

As an example of using BINARY, we could take a monochrome raster file (in this example without any header) and make a negative image by the following Virtue sentence:

```
'image.bw' TEXTREAD BINARY NOT 'negative.bw' TEXTWRITE.
```

works on: ASCII, INTEGER (, REAL)

<left> NUMBER => <numeric value>

This verb will indicate that the data is to be regarded as a simple numeric data, i.e. that it is neither _BINARY_ nor _POLAR_, _CARTESIAN_ etc. All of the inputed single-

dimensional numbers are defined with the subtype _NUMBER_, whereas inputed multi-dimensional numbers using the "imaginary part" form (i.e. using the infixes 'i', 'j', 'k'...) will be automatically regarded as _CARTESIAN_. The _NUMBER_ subtype, as with all Virtue subtype, will be kept with individual data (scalars) as long as possible.

For example, to cast a cartesian coordinate into a polar one (or vice versa) without the cartesian to polar conversion, as it would be done with the verb POLAR (or CARTESIAN), use NUMBER first. Therefore: **`1i1 NUMBER POLAR`** is **`1p1`**, **`1p1 NUMBER`** is **`1i1`** (subtype _NUMBER_), **`1p1 NUMBER CARTESIAN`** is again **`1i1`**, but of the subtype _CARTESIAN_.

Note: NUMBER will convert _BINARY_ ASCII (character) data to it's _NUMBER_ INTEGER form. However, NUMBER will not convert non-_BINARY_ characters. For that use the verb DEASCII.

works on: _BINARY_ ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> ISBOOLEAN => <boolean value>

the result of the ISBOOLEAN operation is #TRUE if the argument contains boolean logic values (true/false) in all dimensions. It will return #FALSE if any numeric value of any number component (part) is not 1 or 0.

ASCII characters are can never be 1 or 0, so ISBOOLEAN will always return #FALSE for character arguments.

It should be noted that in logic operations (NOT, OR, AND...) values outside the range [0,1] are silently converted into boolean values, but the operation ISBOOLEAN is provided to answer on the question if all argument (multidimensional) components are in fact already BOOLEAN.

ISBOOLEAN always returns a singledimensional boolean value.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> ISLOGICAL => <boolean value>

the result of ISLOGICAL operation is #TRUE if the argument contains multivalued logical values (in the range [0,1]) in all dimensions, based on 0 being absolutely fals, 1 being absolutely true, and values inbetween different amounts of trueness (> 0.5) or falseness (< 0.5). Any argument value which is outside of the range [0,1] is not a logical value, and will result in a #FALSE (0).

As ASCII characters are not numeric values, ISLOGICAL on an ASCII argument will always return #FALSE (0).

For INTEGER values ISLOGICAL behaves exactly as ISBOOLEAN, as boolean values are a two-valued subset of the multi-valued logicals.

It should be noted that in logical operations (NOT, OR, AND...) values outside the range [0,1] are silently converted into boolean values, but the operation ISLOGICAL is provided to explicitly state in its result if all the argument (multidimensional) values are in fact logical values.

ISLOGICAL always returns a singledimensional boolean value.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> ISPOLAR => <boolean value>

The result of the verb ISPOLAR is a boolean value #TRUE or #FALSE (expressed in output as T or F) for each individual numeric value in the argument which is in polar coordinates (complex, quaternion or octonion), independent of their expression in radians or degrees. The result of ISPOLAR for single-dimensional numbers (INTEGERs and REALs) is always false)

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION.

<left> ISCARTESIAN => <boolean value>

The result of the verb ISCARTESIAN is a boolean value #TRUE or #FALSE (expressed in output as T or F) for each individual numeric value in the argument which is in cartesian coordinates (complex, quaternion or octonion). The result of ISCARTESIAN for single-dimensional numbers (INTEGERs and REALs) is always false. If a number was not defined to be _CARTESIAN_ (as for example if "touched" by the verbs NUMBER, BINARY, BOOLEAN...) the result of ISCARTESIAN will be false.

Numbers inputed in the multidimensional form with the infixes 'i', 'j', 'k'... between the "imaginary" parts are pressuposed to be _CARTESIAN_, that is **`1i1 ISCARTESIAN`** will be T (true). The same applies also for numbers generated by the SPACE (INTERVAL) verb, as they are actually rectangular indices of the array, therefore they are obviously _CARTESIAN_.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION.

<left> ISBINARY => <boolean value>

Indicates that a particular datum is supposed to behave like a binary datum. Binary data can be characters or single-dimensional numbers. For data which may not be used as _BINARY_ (for example COMPLEX, QUATERNION, OCTONION) the result is always #FALSE.

works on: ASCII, INTEGER , REAL, COMPLEX, QUATERNION, OCTONION

<left> ISNUMBER => <boolean value>

Indicates that a particular datum is supposed to behave like a simple numerical datum. All single-dimensional numbers are generally of the _NUMBER_ subtype (if not asked to be of some other subtype). Multidimensional numbers are _NUMBER_s only if explicitly asked to be (by the verb NUMBER). Character data is never a number, even if _BINARY_. However, character data can be always "transcoded" into its numerical form, either by the expression BINARY NUMBER, or by the verb DEASCII.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> CONTRADICTION => <logical result>

Multidimensional logic accepts more than one value of truthness. It can handle contradictions and uncertainties. With COMPLEX logicals the line going from (0,1) to (1,0) represents non-contradiction, the "fuzzy line", i.e the values representable by single-dimensional logic. This is the line in the complex logical plane where the values of truthness complement each other (their sum is 1), and thus, there's no contradiction. Each dimension represents a condition. For example: true and false. So, when the value of truthness complements the value of falseness, there's no contradiction, and fuzzy logic can deal with these statements. So we can state that multidimensional logic is a meta-set of fuzzy logic. For example, the liar paradox: "This phrase is false" can not be represented in single-dimensional fuzzy logic. It is actually true and false. As a complex logic value it could be represented by r1i1 (e.g. true it is true - the r part, and true it is false - the i part).

The CONTRADICTION operation will flag such contradictions. 1i0 is not contradictory, 0i1 also, but 1i1 is contradictory.

Actually the result of CONTRADICTION is the degree of contradiction in the multidimensional value. For it can be seen as "how far do you get from fuzzy logic". If the result is 0, then the point belongs to the fuzzy line, there's no contradiction, and can be contained by single-dimensional fuzzy logic. As the contradiction increases, it is farther from the fuzzy line, until it reaches total contradiction. With CONTRADICTION, it is possible to see "how contradictory" a proposition is.

The two-dimensional contradiction is calculated as $C(r,i) = |(r+i) - 1|$. For the orthogonality of Virtue data types and operations, it is necessary to expand this two-dimensional definition to QUATERNIONs and OCTONIONs. Virtue allows also for three, five, six or seven dimensional values, but using quaternions and octonions we can not distinguish a non-used dimension from a false (0) in the used dimension. Therefore non-used dimensions must have a NaN in them. So for example 1i0j1kNaN is a 3-dimensional boolean (or logical) value, 1i0j1k0l1mNaNnNaNoNaN is a 5-dimensional logical (boolean) value. (Beware: the logical quaternions and octonions can be shortened by non-used dimensions only by NaNs in their rightmost dimensions. That is 1i0jNaNk1 is not a 3-dimensional logical value, but a 4-dimensional!)

The implemented formula for quaternion CONTRADICTION is $C(r,i,j,k) = |(r+i+j+k) - 2| / 2$, the formula for octonion CONTRADICTION is $C(r,i,j,k,l,m,n,o) = |(r+i+j+k+l+m+n+o) - 4| / 4$. Generally it is the “absolute value of (sum of all dimensions minus half the number of dimensions) divided by half the number of dimensions”.

The result of CONTRADICTION is a single-dimensional fuzzy logic value.

The result of CONTRADICTION for ASCII, INTEGER and REAL, i.e. single-dimensional data, is always 0, there is no contradiction in a single valued statement.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> ROLL => <result>

?

ROLL gives a random integer number in the range [1,<arg>] for positive arg, and [-1,<arg>] for negative arg. ROLL will roll a n-sided (<arg>-sided) dice in each dimension of multidimensional numbers independently, or, if you prefer, it will roll that many dices. As any dice has an integer number of sides, so ROLL will truncate (see the verb INTEGER) all non-integral numbers rounding them to nearest integers towards 0.

The argument 0 is a special case, as we want to throw a dice which has no sides at all. As this is a non-operation, the result of 0 ROLL will be a NaN (not-a-number, obviously, actually a NIL). For multidimensional operations a NaN will be the result in any dimension in which the argument truncates to zero.

For the ASCII type ROLL will give a random character between the space ' ' and the argument character.

The ROLL operation sets the ISPROBABLE data property, so that memoisation of functions using the ROLL is automatically not done. On the ISPROBABLE data property see more at PROBABLY, and the memoisation verbs RESULT and AT/GET/DO.

ROLL will completely disregard the input data subtype (i.e. POLAR), and the result is always PROBABLE NUMBER.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> RANDOM => <result>

??

RANDOM is similar to ROLL, but the result of RANDOM, as opposed to ROLL, is always non-integer. In a multidimensional number the value in each dimension is randomised independently. The result of RANDOM is a real-valued number in the interval (0,<arg>] for positive argument and [<arg>,0) for negative argument. The result of RANDOM on zero is always zero.

For ASCII RANDOM behaves exactly like ROLL, that is, a random character between the space ' ' and the argument character will be generated.

The RANDOM operation sets the ISPROBABLE data property, so that memoisation of functions using the RANDOM is automatically not done. On the ISPROBABLE data property see more at PROBABLY, and the memoisation verbs RESULT and AT/GET/DO.

RANDOM will completely disregard the input data subtype (i.e. POLAR), and the result is always PROBABLE NUMBER.

works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

Monadic structural

<left> ENASCII => <result>

Enascii is the opposite operation of DEASCII.

ENASCII is a form of ENCAPSULATE. It will take an integer valued INTEGER or REAL and convert it into the corresponding ASCII character. Due to the fact that ASCII is character, nonnumeric, data, the conversions to this data type have to be explicitly named. ENASCII is existing because ENCLAPSULATE can not know if the user wishes to encapsulate data into numbers or characters.

ENASCII will try to make an ASCII character from a scalar. If it is not possible (i.e. non-integers, negative numbers, complexes, quaternions, octonions...) it will silently just leave all data elements which can not be converted to characters.

As an example for DEASCII/ENASCII combination we could write the following Virtue sentence to change the color balance of Red, Green and Blue in a 8-bit per color RGB raster (headerless) file:

```
''image.rgb' TEXTREAD DEASCII DUP SHAPE (1.2 0.8 0.9) SWAP
RESHAPE MULTIPLY 0.5 ADD FLOOR 255 SMALLER ENASCII
''enhanced.rgb' TEXTWRITE.
```

Explanation: read the file by 8-bits (characters), convert them to numbers, take it's shape and reshape the color enhancements (1.2 0.8 0.9) to be the same size as the data[1], multiply each component by the desired factor, round the result (0.5 ADD FLOOR), cap it to maximum 255 and then convert it back to characters written in the new file.

works on: INTEGER, REAL

ignores: ASCII, INTEGER of values <0 or >255, (non integer valued) REAL, COMPLEX, QUATERNION, OCTONION

<left> ENCAPSULATE => <result>

Encapsulate is the opposite operation of DECAPSULATE.

ENCAPSULATE will "encapsulate" (multi)dimensional data into (multi)dimensional numbers, that is, it will turn vectors into numbers. SCALARs are left as they are, except integer valued REALs, which are converted to INTEGERs. Vectors of length 1 to 8 will be converted into corresponding multidimensional numbers, any vector longer than 8 elements will not be touched, i.e. will be ignored.

1 element vectors give REAL result, except if the REAL is integer valued. Then the result will be INTEGER.

2 element vectors give COMPLEX result, with the first element of the vector being the real, and the second element the imaginary part.

1. As the data is 3-element (RGB) per pixel as a linear vector, the reshape of the enhancements will also be a linear vector of a repeated enhancements for each 3-element pixel.

3 and 4 element vectors give a 3 or 4 dimensional QUATERNION. A 3 dimensional QUATERNION will have a zero (0) in its 'k' part. The first vector element will be transferred to the 'r' part, the second to the 'i', the third to the 'j'...

5, 6, 7 and 8 element vectors give a 5, 6, 7 or 8 dimensional OCTONION. In the same way as for QUATERNIONs, the unused number dimensions will be zero. The vector is translated from first to the last element into the natural order of OCTONIONs, the same as for COMPLEX and QUATERNION numbers.

A vector of _BOOLEAN_ data will be ENCAPSULATEd into an ASCII or and INTEGER. The encapsulation flows from right, i.e. the least significan digit to right, the most significant digit. Therefore a 6-element vector (T F F F F T) will be ENCAPSULATEd into the _BINARY_ ASCII '!', the same as the 8-element vector (F F T F F F F T), whereas the 9-element vector (F F F T F F F F T) will result in an INTEGER _BINARY_. This is compatible with the 0#XXX notation used for binary data. Beware that 32 and 64 bit implementations of Virtue are not compatible if the vector to be encapsulated is longer then 32 elements. PROBLEM: WHAT TO DO IN THIS CASE?

Encapsulate can not work on three types of data, and will behave like a no-operation on them (however, it will convert as much of the argument as possible):

- ASCII, COMPLEX, QUATERNION and OCTONION data. These data types obviously can not be encapsulated into complex, quaternion or octonion numbers. Virtue will leave the data as is.

- COMPLEXED STRUCTURED, i.e. spaces which contain anything else than only sub-spaces. In the case of such <left> argument, it will be left as is, and Virtue will print a warning message : "1003 - Warning: ENCAPSULATE can not work on COMLPLEXED STRUCTURED spaces. The data will not be converted.".

- multidimensional spaces which are not a simple collection of sub-spaces, whose lowest level are vectors between 1 and 8 elements. In the case that such a space is supplied to ENCAPSULATE, it will complain with the warning: ""1003 - Warning: ENCAPSULATE can not work on multidimensional spaces. The data will not be converted."

works on: INTEGER, REAL

ignores: ASCII, COMPLEX, QUATERNION, OCTONION

<left> DEASCII => <result>

Deascii is the the opposice operation of ENASCII.

Deascii is a form of DECAPSULATE, which works only on character data. It will convert all character data into INTEGER numeric values. The resulting values will be in accordance with the ASCII character set, between 0 and 255. It will silently ignore all other values in the argument, and just copy them to the result.

works on: ASCII

ignores: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> DECAPSULATE => <result>

Decapsulate is the opposite operation of ENCAPSULATE.

Decapsulate will take a character or a number and convert it to the appropriate number or vector. INTEGERs and REALs are decapsulated into a single element vector, COMPLEX numbers into a double element vector, except if the complex number has 0 in the imaginary dimension. Then a one element vector will be formed. QUATERNIONs are decapsulated into one ('i', 'j', 'k' are zero), two ('j' and 'k' are zero), three (if the 'k' part is zero) or four element vectors, OCTONIONS are decapsulated into one, two, three, four, five, six, seven or eigth element vectors. A one dimensional OCTONION has zeros in all imaginary parts, a five dimensional OCTONION would have zeros in the last 3 dimensional parts ('m', 'n' and 'o'), a six dimensional in the last 2 ('n' and 'o'), and a sevendimensional will have a zero in the last part ('o').

BINARY data (either of the basic type ASCII or INTEGER) is DECAPSULATEd into vectors of 8 or 32/64 _BOOLEAN_s. That is, **'A' BINARY DECAPSULATE** will give **(F T F F F F F T)**, i.e. binary 01000001 (hex. 41, dec. 65). As Virtue interpreter may be implemented on 32 and 64 bit integer architectures, the result of e.g. **65 BINARY DECAPSULATE** may be either 32 or 64 bit in length. E.g. the result on a 32 bit implementation will be **(F F F F F F F F F F F F F F F F F F F F F F F F T F F F F F T)**, i.e. binary 0000 0000 0000 0000 0000 0000 0100 0001.

Concording to the difference of the pair ENCAPSULATE (making numerical values) and ENASCII (making character data), DECAPSULATE works only on numeric data.

works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

ignores: ASCII

<left> SHAPE => <vector>

LENGTH

The result is the shape of the space given as argument. SHAPE gives the shape of the (multidimensional) array, but does not go into sub-structures. The sub-structures (subspaces, subarrays) of an array are regarded as single element data, i.e. scalars.

The shape of a scalar is #NIL (i.e. the scalar has no shape, as it is a datum, and not a space - the same as e.g. 'atom' in some languages, like LISP).

A special case is when the argument of SHAPE is an ADDRESS or LABEL. In this case SHAPE will give the name of the addressed variable or the name of the label. It is the "shape" of an address or label. The name will be returned in a character vector.

For all other arguments, SHAPE will always return a vector of INTEGERs. Virtue is a column major order processor of arrays. Therefore the elements of the resulting shape-vector will be ordered by a column major order. The length of each dimension of the (multidimensional) array is represented by one element in the resulting vector.

<left> RAVEL => <vector>

RAVEL will turn any (multidimensional) array into a (onedimensional) vector. It will not go into substructures (subarrays), as they are regarded as single data elements.

For non-structured arrays, RAVEL is equivalent to ENLIST. Structured arrays will be converted to a structured vector. If a whole substructured space shall be converted to a vector of all individual elements, use ENLIST.

<left> ENCLOSE => <one element vector>

this operation encloses a space within a single-element vector. This means that the result of enclose will be a substructure (subarray, subspace), which behaves like a single vector/array element. An enclosed substructure will behave as any other scalar. Scalar operators on enclosed substructures will apply the operation to all subarray elements for each element of the array(s) supplied. That is: **`(1 2 3) (1 2 3) +`** gives **`(2 4 6)`**. **`(1 2 3) (1 2 3) ENCLOSE +`** gives **`((2 3 4) (3 4 5) (4 5 6))`**.

<left> DISCLOSE => <one element vector>

this is the opposite operation from ENCLOSE. It works (presently) only on a single element vector whose element is an enclosed space. Disclose will take that space out on the top level, i.e. it will decrement the depth of the space by one.

DISCLOSE works only on one element structured vector. If an argument of another shape is given to DISCLOSE, Virtue will complain with an error "973 - DISCLOSE can not work on structured spaces - implementation restriction."

The implementation will be enhanced to implement full DISCLOSE funcitionality.

<left> FIRST => <result>

CAR

FIRST has two forms: the FIRST, and the FIRST ELEMENT.

FIRST gives the first "plane" of a space, i.e. it discards the rightmost dimension. For vectors FIRST gives a scalar first element. A FIRST on a two dimensional array will be a vector, on three dimensional array a twodimensional plane etc. The shape of the result is the same as the shape of the argument without it's last dimension. Therefore the FIRST element of a vector is a scalar. The complementary operation to FIRST is REST. If the FIRST element of a vector was a substructure (subspace), the result will be a scalar with the substructure (rank 0), but the DEPTH will be the same as the original depth of that element in the argument array.

On vectors FIRST is analogous to CAR in LISP, or, it actually behaves like a multidimensional CAR.

<left>FIRST ELEMENT => <result>

FIRSTONE

FIRST ELEMENT will give the first element of any (multidimensional) array, as a scalar. However, if the first element of the array was a substructure (subarray), it will be raised one level (like with DISCLOSE on a single element vector).

This means that, as opposed to FIRST, FIRSTONE will completely disregard the dimensionality of the argument array, and will DISCLOSE a subarray if in the first element of the argument.

FIRST ELEMENT is equivalent to RAVEL FIRST DISCLOSE.

FIRST ELEMENT behaves analogous to the monadic operator "first" in APL.

<left> REST => <result>

CDR

REST has two forms: the REST, and the REDUCING REST.

Both RESTs are complementary operations of FIRST. REST will give all the rest of the space except the first element as defined by the verb FIRST. The result of REST is always an array of the same dimensionality (the same RANK) as the original. The REST of a single dimensional (multidimensional) array is a NIL, but the dimensionality of that NIL is not 1 (as for the constant #NIL), but the same as the dimensionality of the argument, with its all shape elements equal to 0. For example (1 2 3 4 5 6)[3 2] REST gives (4 5 6)[3 1], the REST on this will give ()[0 0].

On vectors REST is analogous to CDR in LISP, or, it actually behaves like a multidimensional CDR.

<left> REDUCING REST => <result>

RREST

The REDUCING REST is also a complementary operation to FIRST, and will give all the rest of the space except the first element as defined for the verb FIRST. The result of REDUCING REST is always a space (rank >= 1), except for the REDUCING REST of a single element vector, when the result is a scalar. The shape of the REDUCING REST result is the same as the shape of the argument, but with it's last dimension number of elements reduced by one, as long as there are more planes in the last dimension. If the length of the last dimension would be 1, REDUCING REST discards this dimension, and the rank is reduced by one. That is why the RREST of a single element vector is a scalar. The REDUCING REST of a scalar is a #NIL, or an empty character vector '', depending on the existance of character data in the original argument. If there were any characters in the argument, the scalar array will be '', otherwise a #NIL.

On vectors REDUCING REST is analogous to CDR in LISP, or it can be regarded as a kind of multidimensional CDR.

The difference between REST and REDUCING REST can probably most easy be explained with boxes. Imagine you have a box with two rows of two boxes, and in each of them there are three things. In Virtue something like **`((1 2 3)(4 5 6)(7 8 9)(10 11 12))[2 2]`**. With **`REST`** you first discard the first row of boxes. What is left is a box with one row of two boxes with the items. Next time you apply **`REST`**, you get rid of the (now only) row of two boxes with their content. What is left is an empty box, but which still has empty space for empty boxes not holding things. In Virtue you will get a two-dimensional empty array.

Applying **`REDUCING REST`** to the above example: First you take out the first row of boxes, the same as with REST. You still have a box with two boxes with things. Next time you apply **`REDUCING REST`** you discard the big, now empty box, and get the smaller two boxes with three things each. Now **`REDUCING REST`** will discard the first of the two boxes. Next time **`REDUCING REST`** discards the second box, and there is nothing left. In Virtue you will get a simple #NIL.

`RREST RREST` on the above example is equivalent to **`REST FIRST REST`** (but faster).

<left> TAKE => <result>

Take works only on single-dimensional spaces (i.e. vectors).

<left> DROP => <result>

Take works only on single-dimensional spaces (i.e. vectors).

<left> MIRROR => <result>

The MIRROR mirrors all elements around the central point of the multi-dimensional space. The shape of the space stays the same. For a single-dimensional space, a vector, it is simply reversing the elements: **`5 INTERVAL MIRROR`** is **`(5 4 3 2 1)`**. A result of a two-dimensional MIRROR (or if you still prefer to call it REVERSE) would put the last element of both dimensions into the first element of both dimensions and vice versa. The same applies for all elements around the central point. For example **`3i2 SPACE MIRROR`** will give **`(3i2 2i2 1i2 3i1 2i1 1i1)[3 2]`**. In a space of (4 3 2) elements the element at (1 1 1) after the MIRROR will be the element formerly at (4 3 2), and at (4 3 2) will be the one from (1 1 1).

MIRROR on a vector is the same as REVERSE on the same vector.

Mirror has to physically move data, so it is very memory access intensive.

<left> REVERSE => <result>

REVERSEFIRSTAXIS

REVERSE reverses the order of elements in the first dimension of a space, i.e. along the fist axis, the first index. For example **`3i2 SPACE REVERSE`** will give **`(3i1 2i1 1i1 3i2 2i2 1i2)[3 2]`**.

REVERSE on a vector is the same as the MIRROR on the same vector.

<left> TRANSPOSE => <result>

Transposes the space, i.e. reverses the order of dimensions. A column-major index space will become a row-major and vice versa.

Transpose ignores scalars and vectors, as their transpositions are the same as originals (no dimensions to order).

Transpose works (presently) only on spaces of up to 8 dimensions.

Transpose has to physically move data, so it is very memory access intensive.

<left> INTERVAL => <space>

..

SPACE

INTERVAL is the main Virtue word which makes spaces. Generally it will make an interval vector/array defined by the argument. The argument to INTERVAL has only three possible forms, a nonstructured scalar (or one element vector), 2-element or 3-element vector. Depending on the three forms of argument INTERVAL makes an interval of consecutive numbers (not necessarily INTEGER):

(1) - integer valued scalar or one-element vector: make an array of consecutive integer valued numbers up to that number. A positive number will make the interval [1,<arg>], a 0 will make a #NIL, and a negative number will make the interval [-1,<arg>].

(2) - two-element vector. The interval will be started with the first argument element, and incremented by 1 up to the maximum of the second argument element. For example: **`(1.1 4.4) INTERVAL`** will give a four element vector **`(1.1 2.1 3.1 4.1)`**. If the first element of the argument vector is smaller than the second, a reverse flowing interval will be generated, with the last element value not less then the second argument element. For example: **`(4.4 1.1) INTERVAL`** will give a four element vector **`(4.4 3.4 2.4 1.4)`**.

(3) - three-element vector. The interval will be started with the first argument element (from), finished not greater than the second argument element (to), and will be incremented by the value specified by the third argument element (step). This is is similar to the **`FOR .. TO .. STEP ..`** statement in many computer languages, however the result is not a loop, but a vector/array of the specified elements. For example: **`(4.4 1.1 –0.4) INTERVAL`** will give the 9-element vector **`(4.4 4 3.6 3.2 2.8 2.4 2 1.6 1.2)`**. If the "from" is smaller then the "to", and the "step" is negative, or the "from" is greater then

"to", and the "step" is positive, Virtue will complain with the exception "158 - the specified INTERVAL would be infinite.".

INTERVAL is the most important Virtue constructor, as it actually constructs index spaces, spaces in which each element is the (multidimensional) address (index) of that element. This allows many algorithms to be written very efficiently, and run in parallel.

To allow such versality, INTERVAL works with all Virtue supported data-types: on ASCII, INTEGER, REAL, COMPLEX, QUATERNION and OCTONION values.

- ASCII: make an string of characters "to", or "from" "to". In the form of one element ASCII argument, INTERVAL will make a consecutive (theoretically computer implementation dependant, but generally actually ASCII) sequence of characters form space (' ') to the argument character. Virtue will not make intervals into control characters, i.e. those whose ASCII value is less then the space (' '). INTERVAL on character data can not have a "step" (quite obviously). Reverse intervals are supported. For example: **`('z' 'a') INTERVAL`** will give the character vector **`'zyxwvutsrqponmlkjihgfedcba'`**.

- INTEGER, REAL: the behaviour of INTERVAL on INTEGER and REAL arguments is given in the general explanation of different interval specifications.

- COMPLEX, QUATERNION, OCTONION: To allow full manipulation with index spaces, Virtue, regarding those numbers as multidimensional numbers, makes multidimensional index arrays when using them. An INTERVAL argument with COMPLEX numbers will produce a two-dimensional array, QUATERNION numbers will produce a 3- (if the k part is zero) or a 4-dimensional array, and OCTONIONS will produce 5-, 6-, 7- or 8- dimensional arrays. So, for example, to generate some 3 dimensional index space we would use three-dimensional QUATERNIONS: **`(-1i-1j-1 1i1j1) INTERVAL`** would give the following space:

```
-1i-1j-1  0i-1j-1    1i-1j-1
-1i 0j-1  0i 0j-1    1i 0j-1
-1i 1j-1  0i1j-1     1i 1j-1

-1i-1     0i-1       1i-1
-1        0          1
-1i1      0i 1       1i 1

-1i-1j 1  0i-1j 1    1i-1j 1
-1i 0j 1  0i 0j 1    1i 0j 1
-1i 1j 1  0i 1j 1    1i 1j 1
```

As another example, a 5-dimensional sphere of 1's inside 0's, in the interval of 10 points around zero, and with the radius of 8, would be specified in Virtue as following: **`(-10i-10j-10k-10l-10 10i10j10k10l10) SPACE MAGNITUDE 8 NOTGREATER`**.

It is important to note that all numbers in Virtue are effectively "downgraded" to their lowest possible representation. That is, an OCTONION with zeros in all imaginary dimensions is regarded as a REAL (or even INTEGER), or an QUATERNION with zeros in the j and k dimension is actually a COMPLEX number.

Though beware: all arguments in the interval definition vector have to be of the same dimensionality. Therefore (0 10i10) INTERVAL shall be written as (0i0 10i10) INTERVAL, otherwise an error exception "157 - all arguments to INTERVAL shall be of the same

dimensionality." will be raised. This is a temporary restriction, to be lifted in later versions/ releases of Virtue.

No argument to INTERVAL may be a structured space (depth > 1). Othervise Virtue will raise the exception "154 - the argument to INTERVAL must be a scalar or a 1-, 2- or 3-element vector."

<left> ENLIST => <space>

ENLIST makes a single-dimensional vector of *all* elements contained in any space. This means that all the subarrays (substructures) are element by element enlisted in the final result, according to their position in the original space. If the applied space is a simple, non-structured (i.e. depth == 1) (multidimensional) array, ENLIST behaves exactly like RAVEL.

<left> DEPTH => <integer scalar>

gives the depth of the argument. The depth of a space is actually the amount of enclosed substructures. The depth of a scalar is zero, of a non-structured array 1, and each deeper level of sub-structuring is counted. The DEPTH will give the depth of the deepest sub-structure. For example: **`(((1 2 3)) (4 5) 6) DEPTH`** is **`3`**, **`((4 5) 6) DEPTH is 2, (6) DEPTH`** is **`1`** and **`6 DEPTH`** is **`0`**.

<left> RANK => <integer scalar>

RANK is equal to the dimensionality of the argument array (space). It is equal to the SHAPE SHAPE (i.e. the shape of the shape). Scalars have RANK 0, vectors 1, 2-dimensional arrays 2 etc.

Time

<left> SLEEP =>

Asks Virtue to do nothing for as close as possible to the amount of seconds specified in the argument. The sleeping interval can be specified down to microseconds, by using a real number. For example: 3.600222 SLEEP will try to sleep exactly 3 seconds, 600 miliseconds and 222 microseconds. Beware that the precision of sleep can, due to OS and Virtue implementations, not be guaranteed.

The argument to SLEEP must be a scalar or a single element vector. It shall be either INTEGER or REAL.

TIME => <scalar real>

The TIME werb will give as a result the UTC time of day, as a Julian date with the precission up to microseconds. Julian days are very practical due to the fact that they can be

subtracted to give a Julian day difference. **`TIME 0.5 + FLOOR 7 MODULO`** will always give the proper day of the week (0 is Monday, 6 is Sunday), throughout our know history. (FLOOR is used above to get rid of the sub-day time, i.e. the hours, minutes, seconds... microseconds.)

The Julian date keeps time in days, starting at noon. To get the seconds multiply the TIME with 86400, the number of seconds in a day.

The precision of the TIME is very much dependant on the particular way the Operating Systems keeps time. NTP synchronized computers will mostly be within +/- 128 miliseconds from the internationally standardised UTC. A small error is introduced also by Virtue, as there is processing to do until the first next verb which can work on the given time. This delay is directly dependant on the speed of the computer used, and indirectly on the tasking behaviour of the given system.

About the Julian day see more e.g. at https://en.wikipedia.org/wiki/Julian_day.

Dyadic scalar

RULES:.....

<left> <right> ADD => <result>

+

Mathematical addition.

For complex, quaternion and octionion numbers, the addition is a straightforward operation, and is performed by adding the appropriate dimensions of both arguments. Geometrically, the operation extends the position (coordinate) in each dimension by the amount of the other argument, producing vector addition.

<left> <right> SUBTRACT => <result>

-

CENTRE, CENTER

Mathematical subtraction. The synonyms CENTRE and CENTER are provided to be used whenever a vector/array/space of indices (see INTERVAL) has to be centred around 0. Namely, it is obvious that the subtraction of a number from any linearily increasing number vector is actually identical to the notion of centering this vector around 0.

For example, the synonym CENTRE could be helpful in understanding the following Virtue sentence: **`3i3 SPACE 2i2 CENTRE.`**, giving the following matrix:

```
-1i-1  0i-1  1i-1

-1     0     1

-1i1   0i1   1i1
```

For complex, quaternion and octionion numbers, the subtraction is a straightforward operation, and is performed by subtracting the appropriate dimensions of both arguments. Geometrically, the operation contracts the position (coordinate) in each dimension by the amount of the other argument, producing vector subtraction.

<left> <right> MULTIPLY => <result>

*****

TIMES

Mathematical multiplication.

For integers, reals and complex numbers the multiplication is commutative, i.e. **`1i1 2i2 *`** is the same as **`2i2 1i1 *`**.

For quaternions and octonions the multiplication is non-commutative. There are two mathematical ways to perform quaternion and octonion multiplication: left to right and right to left. In Virtue the adopted algorithm is left to right, which is compatible to the general left to right philosophy of Virtue, and is also the traditional approach.

Therefore, the multiplication **`1i2j3k4 4i3j2k1 *`** gives the result **`–12i6j24k12`**, and the multiplication **`4i3j2k1 1i2j3k4 *`** results in **`–12i16j4k22`**.

As multiplication of multidimensional numbers is a very complex operation, for quaternions requiring 16 floating point multiplications and 12 additions/subtractions, and for octonions involving 64 multiplications and 56 additions/subtractions, Virtue will, before performing any mathematical operations, check if a shorter multiplication algorithm can be used. Therefore a multiplication of a single-dimensional real with an octonion will necessitate only 8 multiplications, and a multiplication of two three-dimensional quaternions will necessitate only 9 multiplications and 5 additions. As already defined, the dimensionality of numbers in Virtue is defined by the zeros in last (therefore not used) dimensions.

<left> <right> DIVIDE => <result>

/

Mathematical division.

There are two mathematical ways to perform quaternion and octonion division: left to right and right to left. In Virtue the adopted algorithm is left to right, which is compatible to the general left to right philosophy of Virtue, and is also the traditional approach in mathematics. Therefore **`1i2j3k4 4i3j2k1 DIVIDE`** results in the number **`0.6666666667i0.3333333333j0k0.6666666667`**.

As division of multidimensional numbers is a very complex operation, for quaternions requiring 20 floating point multiplications, 18 additions/subtractions, 4 divisions and a square root, and for octonions involving 72 multiplications, 63 additions/subtractions, 8 divisions and a square root, Virtue will, before performing any mathematical operations, check if a shorter division algorithm can be used. Therefore a division of a single-dimensional real with an octonion will necessitate only 16 multiplications, 14 additions/subtractions, 8 divisions

and a square root, whereas the division of an octionion with a real will necessitate just 8 divisions and nothing more. As already defined, the dimensionality of numbers in Virtue is defined by the zeros in the last (therefore not used) dimension(s).

<left> <right> RESIDUE => <result>

MODULO, MOD

The MODULO word gives the residue after integer division of the <left> argument with the <right> argument. So for example `8 7 MOD` will give `1`, `2.5 1.2 MOD` will give `0.1`.

It is important to note that for quaternions and octonions the multiplication order is important, therefore the formula for RESIDUE must be as follows:

*Left - floor(Left / Right) * Right*

Quaternion and octonion RESIDUE is not implemented yet!

Works on: INTEGER, REAL, COMPLEX

<left> <right> MINIMUM => <result>

SMALLER, MIN

The result of MINIMUM is the smaller of the two arguments. MINIMUM works on all types of data, except addresses.

For ASCII characters, the result will be the alphabetically (i.e. code dependent order) earlier positioned character. That is `'a' 'z' SMALLER` will give `'a'`.

For single-dimensional numbers (integers and reals) the smaller of the two is returned.

For multi-dimensional numbers (complex, quaternion and octonion) the ordering of numbers for an operation such as minimum (or any other comparison) has to be uniquely defined for the purposes of consistency. It may be generally accepted that a possible way to compare multidimensional numbers is to compare their magnitudes, i.e. the distance from point 0 of the appropriate space to the coordinates indicated by the number. Therefore all numbers lying on the surface of the distance defined sphere and further are “larger” than the numbers inside this sphere. From this follows that the MINIMUM werb is not discriminating individual multidimensional numbers lying anywhere on the mentioned sphere, or numbers pointing in different directions.

However, to keep the basic ordering of the spaces, MINIMUM will use the SIGNUM to distinguish between the “positive” and “negative” “side” of the space. See the description of the verb SIGNUM for the definition for multidimensional numbers.

Works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> <right> MAXIMUM => <result>

LARGER, MAX

The result of MAXIMUM is the larger of the two arguments. MAXIMUM works on all types of data, except addresses.

For ASCII characters, the result will be the alphabetically (i.e. code dependent order) later positioned character. That is `'a' 'z' LARGER` will give `'z'`.

For single-dimensional numbers (integers and reals) the larger of the two is returned.

For multi-dimensional numbers (complex, quaternion and octonion) the ordering of numbers for an operation such as maximum (or any other comparison) has to be uniquely defined for the purposes of consistency. It may be generally accepted that a possible way to compare multidimensional numbers is to compare their magnitudes, i.e. the distance from point 0 of the appropriate space to the coordinates indicated by the number. Therefore all numbers lying on the surface of the distance defined sphere and further are "larger" than the numbers inside this sphere. From this follows that the MAXIMUM werb is not discriminating individual multidimensional numbers lying anywhere on the mentioned sphere, or numbers pointing in different directions.

However, to keep the basic ordering of the spaces, MAXIMUM will use the SIGNUM to distinguish between the "positive" and "negative" "side" of the space. See the description of the verb SIGNUM for the definition for multidimensional numbers.

Works on: ASCII, INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> <right> POWER => <result>

******

POW

The <left> argument raised to the power given by the <right> argument.

Works on: INTEGER, REAL, COMPLEX, QUATERNION, OCTONION

<left> <right> LOGARITHM => <result>

LOG

<left> <right> CIRCULAR => <result>

TRIG, TRIGONOMETRIC

The CIRCULAR verb is different from all the other scalar dyadic operators, as its <right> argument indicates the trigonometric function to be performed as a monadic

operation. In all other respects the CIRCULAR is a scalar dyadic operator, i.e. all argument and result shape rules apply as with other scalar dyadics. The verb CIRCULAR is inspired by APL, and the numeric values used for different functions are compatible with the APL2 values.

The <right> argument, being the operation indicator, has to be integer in the appropriate range. To ease the use of the <right> argument, all individual trigonometric functions have special mnemonics, as for example #SIN, #TAN, #ATANH etc. The full list of these is given under constants.

The verb CIRCULAR is very convenient when several trigonometric operations have to be done on number(s), or different operations done on individual elements. So for example: **`(¯1 0 1) (#SIN #COS #TAN) CIRCULAR`** gives the sine of -1, cosine of 0 and tangens of 1: **`¯0.8414709848 1 1.557407725`**; and **`(¯1 0 1) ((#SIN #COS #TAN)) CIRCULAR`** will give the sines, cosines and tangenses of each of the three argument numbers: **`( ¯0.8414709848 0.5403023059 ¯1.557407725 ) ( 0 1 0 ) ( 0.8414709848 0.54023059 1.557407725 )`**.

The following trigonometric functions are defined:

#ATANH, -7: Hyperbolic Arcus (inverse) Tangens

#ACOSH, -6: Hyperbolic Arcus (inverse) Cosinus

#ASINH, -5: Hyperbolic Arcus (inverse) Sinus

#SQRTX2m1, -4: Square root of X times X minus 1

#ATAN, -3: Arcus (inverse) Tangens

#ACOS, -2: Arcus (inverse) Cosinus

#ASIN, -1: Arcus (inverse) Sinus

#SQRT1mX2, 0: Square root of 1 minus X times X

#SIN, 1: Sinus

#COS, 2: Cosinus

#TAN, 3: Tangens

#SQRTX2p1, 4: Square root of X times X plus 1

#SINH, 5: Hyperbolic Sinus

#COSH, 6: Hyperbolic Cosinus

#TANH, 7: Hyperbolic Tangens

<left> <right> BINOMIAL => <result>

<left> <right> AND => <result>

^

<left> <right> OR => <result>

v

Logical OR.

Beware: the Virtue symbol for the logical operation OR is the letter “v”!

<left> <right> NAND => <result>

~^

<left> <right> NOR => <result>

~v

<left> <right> STRONGAND => <result>

!^

<left> <right> STRONGOR => <result>

!v

<left> <right> STRONGNAND => <result>

!~^

<left> <right> STRONGNOR => <result>

!~v

<left> <right> IDENTICAL => <result>

==

The result is _BOOLEAN_ yes if the respective elements of <left> and <right> are identical, which means that for any floating point numbers the result will depend on their exact match. Otherwise, INTEGERs are compared with REALs on their mathematical value.

This operation is different from EQUIVALENT (EQUAL, =).

For example: **`#PI SQRT SQUARE #PI ==`** (IDENTICAL) will result in a **`F`**, whereas **`#PI SQRT SQUARE #PI =`** (EQUIVALENT) will result in a **`1`** for all input/output purposes (although the real value returned is _LOGICAL_ almost 1: precisely *1 - 4.44089209850063e-16*).

IDENTICAL is much faster then EQUIVALENT, but may be used only if it is obvious that the floating point data will not be manipulated by higher mathematical functions.

The result of IDENTICAL is always _BOOLEAN_.

<left> <right> NOTIDENTICAL => <result>

##, ~==

The result is _BOOLEAN_ yes if the respective elements of <left> and <right> are not identical, which means that for any floating point numbers the result will depend on their exact mismatch. Otherwise, INTEGERs are compared with REALs on their mathematical value.

This operation is different from NOTEQUIVALENT (NOTEQUAL, <>, #, ~=).

For example: **`#PI SQRT SQUARE #PI NOTIDENTICAL`** will result in a **`T`**, whereas **`#PI SQRT SQUARE #PI NOTEQUIVALENT`** will result in a **`0`** for all input/output purposes (although the real value returned is _LOGICAL_ almost 0: precisely *4.44089209850063e-16*).

NOTIDENTICAL is much faster then NOTEQUIVALENT, but may be used only if it is obvious that the floating point data will not be manipulated by higher mathematical functions.

The result of NOTIDENTICAL is always _BOOLEAN_.

<left> <right> LESS => <result>

<

<left> <right> NOTGREATER => <result>

=<, <=, ~>

<left> <right> EQUIVALENT => <result>

=

EQUAL

This verb compares the <left> with the <right> argument taking into account the imprecissions of floating point mathematical calculations in digital computers. This is achieved by using Comparison Tolerance (#CT), a value which defines the floating point calculations precision.

Therefore, although slower then IDENTICAL (as each value has to be compared to upper and lower limits of Comparison Tolerance), EQUIVALENT will properly compare two floating point numbers, disregarding the possibly calculation introduced (precision) error. The user can always change the Comparison Tolerance (see #CT), and therefore directly influence the precision to which two numbers will be regarded as EQUIVALENT.

When comparing two floating point numbers, the result of the comparison will be 0, i.e. false, if the two numbers differ more than allowed by Comparison Tolerance, and will be 1-(difference between the numbers) for those which compare as equivalent. In other words, the result of EQUIVALENT will either be false, almost true or true. It will be true if the numbers were IDENTICAL. The "almost true" value, calculated by subracting the effective difference of the two numbers compared from "true" (1), actually shows the present error.

Therefore both **`#PI SQRT SQUARE #PI NOTIDENTCAL`** and **`#PI SQRT SQUARE #PI EQUIVALENT`** will result in a **`1`** for all input/output purposes (although the real value returned by EQUAL is _LOGICAL_ almost 1: precisely *1 - 4.44089209850063e-16*).

BEWARE: As already mentioned, the result of EQUIVALENT is a _LOGICAL_ value, showing the exact imprecission of the comparison. That means that the sentence **`#PI SQRT SQUARE #PI EQUIVALENT CHECK 'Equal' IF_YES 'Not equal' IF_NO.`** has a 1:2.251799814e+15 chance to answer 'Not equal'! If the user wishes to have always the answer 'Equal', it is necessary to convert the _LOGICAL_ value into _BOOLEAN_ using the monadic BOOLEAN, i.e. ... **`EQUIVALENT BOOLEAN`** ... **`CHECK`** ... .

The result of EQUIVALENT is _BOOLEAN_ for ASCII and INTEGER, and is always _LOGICAL_ for any combinations involving at least one floating point number.

<left> <right> NOTLESS => <result>

=>, >=, ~<

<left> <right> GREATER => <result>

>

<left> <right> NOTEQUVALENT => <result>

<>, #, ~=

NOTEQUAL

This verb compares the <left> with the <right> argument taking into account the imprecissions of floating point mathematical calculations in digital computers. This is achieved by using Comparison Tolerance (#CT), a value which defines the floating point calculations precision.

Therefore, although slower then NOTIDENTICAL (as each value has to be compared to upper and lower limits of Comparison Tolerance), NOTEQUIVALENT will properly compare two floating point numbers, disregarding the possibly calculation introduced (precision) error. The user can always change the Comparison Tolerance (see #CT), and therefore directly influence the precision to which two numbers will be regarded as NOTEQUIVALENT.

When comparing two floating point numbers, the result of the comparison will be 1, i.e. true, if the two numbers differ more than allowed by Comparison Tolerance, and will be the difference between the numbers for those which compare as equivalent. In other words, the result of NOTEQUIVALENT will either be true, almost false or false. It will be true if the numbers were NOTIDENTICAL. The "almost false" value, being the effective difference of the two numbers inside the Comparison Tolerance, actually shows the present error.

Therefore **`#PI SQRT SQUARE #PI NOTIDENTCAL`** will result in a **0** for all input/output purposes (although the real value returned by EQUAL is _LOGICAL_ almost 0: precisely *4.44089209850063e-16*, i.e. the actual difference of the numbers).

BEWARE: As already mentioned, the result of NOTEQUIVALENT is a _LOGICAL_ value, showing the exact imprecission of the comparison. That means that the sentence **`#PI SQRT SQUARE #PI NOTEQUIVALENT CHECK 'Not equal' IF_YES 'Equal' IF_NO.`** has a 1:2.251799814e+15 chance to answer 'Equal'! If the user wishes to have always the answer 'Not equal', it is necessary to convert the _LOGICAL_ value into _BOOLEAN_ using the monadic BOOLEAN, i.e. ... **`NOTEQUIVALENT BOOLEAN`** ... **`CHECK`** ... .

The result of NOTEQUIVALENT is _BOOLEAN_ for ASCII and INTEGER, and is always _LOGICAL_ for any combinations involving at least one floating point number.

Structural (non-scalar) dyadics

<left> <right> RESHAPE => <result>

reshapes any vector or array (remember that substructures are scalars) given as the <left> argument into the shape indicated by the <right> argument. The number of dimensions of the resulting array is equal to the number of <right> elements, and the values of individual elements define the lengths (sizes) of each dimension. In Virtue the order of dimensions, to be compatible with the multidimensional numbers is "column first", i.e. the leftmost index is the first incremented and indicates the first dimension (see also e.g. "<left> INTERVAL => <space>" on page 33).

The right argument, indicating the shape, can have any number of elements, as long as they are zero or positive integer[1]. A #NIL can also be the <right> argument to RESHAPE. In that case, as #NIL consistently means 'nothing' in Virtue, the <left> argument will be left as is. However, to indicate that it shall be even less structured than it was before, it is assigned to be a SET.

A direct reshape can be obtained in input (and "structural" output of WRITE etc.) as a "shaped space" by appending the shape vector to a specified vector, by writing "(x)[y]" where x and y are vectors. The elements of x may be any valid data type which can be specified inside a vector, that is a #NIL, or one or more elements (characters or numbers). The y has to be a vector of one or more positive integers (including 0). A shaped space indicated by the absence of y, that is "(x)[]" will be regarded as a SET (the same as reshaping by #NIL). The shaped spaces may be nested.

<left> <right> CATENATE => <result>

,

CAT, CONS

Catenating a SCALAR with #NIL will make a single element vector of the scalar. This is compatible with the Lisp definition of (CONS 'x ()). Otherwise, CATENATE will not catenate #NILs into a vector. If you want to get #NILs into a vector, ENCLOSE them first, or write them already in a subvector like **(())** .

In other words, **#NIL #NIL CATENATE** gives #NIL, ***<vector>* #NIL CATENATE** gives <vector>, **#NIL *<vector>* CATENATE** gives the <vector>, **#NIL *<scalar>* CATENATE** gives <scalar>, but ***<scalar>* #NIL CATENATE** gives a one element vector of the scalar. See also the description of #NIL.

1.Multidimensional numbers which are effectively integers are automatically recognised as such by Virtue, and used as real INTEGERs. It is important to note that such internal operations are completely transparent to the user. The user will percieve only integer, or single dimensional, etc. values independently of the way they are internally kept.

<left> <right> DEAL => <result>

!??

<left> <right> MATCH => <result>

<left> <right> EXACT MATCH => <result>

Reductions

<left> <operator> REDUCE => <result>

//

The verb REDUCE will make a reduction of a space along the first axis, i.e. along the leftmost index, the column. The <operator> specified with REDUCE has to be a dyadic scalar verb, like ADD, MULTIPLY, STRONGOR, POWER... However, the user shall take into consideration that not all reductions using scalar dyadics will make a lot of sense.

The most commonly used reductions have their own specific names, see in further text SUM, PRODUCT etc. All specially named reductions (except PRODUCT on quaternion and octonion numbers) are using commutative operations, and can therefore be executed in parallel with minimal communication.

There is a special case in reductions using non-commutative operations. A left to right reduction by a non-commutative operation is easy to implement, in the same way as a commutative operation, as sub-reductions may be freely done in parallel (that would be an "infix" reduction from left to rith). However, this type of reduction using non-commutative operations actually has not a lot of sense. A much more algorithmically interesting and useful reduction is the "postfix" reduction, which gives a result which is the same as by applying postfix operators on a stack, i.e. **`(1 2 3 4) SUBTRACT REDUCE`** will give the result **`−3`**, the same as the Virtue sentence **`1 2 3 4 − − −`**, that is, in infix mathematical notation *1-(2-(3-4))*. This type of reduction is used for any non-commutative operation, as for example SUBTRACT, DIVIDE, or quaternion and octonion MULTIPLY etc. Beware that algorithmically these reductions can not be parallelised, so their execution will always be strictly serial, although possibly on different processors or computers.

Some of the reductions, like e.g. *NOTLESS REDUCE*, will have a very limited usefulness, or none at all. This particular sentence will give a _BOOLEAN_ result which will be a TRUE only if (e.g.) the first element of the vector is bigger or equal to 1, i.e. it will be, independent of the length of the vector equal to **`FIRST 1 NOTLESS`**, except when reducing a two element vector, where the result is the same as NOTLESS on two arguments.

<left> SUM => <result>

+//

ADD REDUCE

The verb SUM has the meaning of **ADD REDUCE**. It will make a reduction of a space along the first axis, i.e. along the leftmost index, the column, by adding all the elements of that leftmost axis.

As addition is always commutative for any-dimensional numbers, the SUM can be calculated by parallel execution of serial summations.

<left> PRODUCT => <result>

***//**

MULTIPLY REDUCE

The verb PRODUCT has the meaning of **MULTIPLY REDUCE**. It will make a reduction of a space along the first axis, i.e. along the leftmost index, the column, by multiplying all the elements of that leftmost axis.

Multiplication is a commutative operation for integers, reals and complex numbers. However, it is not commutative for quaternion and octotion numbers. Therefore the PRODUCT for one- and two-dimensional numbers can be calculated by parallel execution of serial multiplications, i.e. from left towards the right, from lower index towards the higher. However, for higher-dimensional numbers, the quaternions and octonions, multiplication is non-commutative, therefore the PRODUCT verb is executed strictly serially from right, the last axis index, towards left, the first axis index. See also the verb REDUCE.

<left> ALL => <result>

^//

AND REDUCE

<left> ANY => <result>

v//

OR REDUCE

<left> SMALLEST => <result>

MINIMUM REDUCE, MIN REDUCE

<left> LARGEST => <result>

MIXIMUM REDUCE, MAX REDUCE

<left> <operator> REDUCEROWS => <result>

/-/

REDUCELASTAXIS

will use the <operator> to reduce the last axis of an array. The last axis is actually the "rows" axis.

<left> SUMROWS => <result>

+/-/

SUMLASTAXIS, ADD REDUCELASTAXIS, ADD REDUCEROWS

<left> PRODUCTROWS => <result>

***/-/**

PRODUCTLASTAXIS, MULTIPLY REDUCELASTAXIS, MULTIPLY REDUCEROWS

<left> ALLROWS => <result>

^/-/

ALLLASTAXIS, AND REDUCELASTAXIS, AND REDUCEROWS

<left> ANYROW => <result>

v/-/

ANYLASTAXIS, OR REDUCELASTAXIS, OR REDUCEROWS

<left> SMALLESTROW => <result>

SMALLESTLASTAXIS, MINIMUM REDUCELASTAXIS, MINIMUM REDUCEROWS

will give the smallest element in each particular row. This means that the (multidimensional) array will be reduced by one, last, dimension.

<left> LARGESTROW => <result>

LARGESTLASTAXIS, MAXIMUM REDUCELASTAXIS, MAXIMUM REDUCEROWS

will give the largest element of each particular row.

<left> <integer right> COMPRESS => <result>

COMPRESSCOLUMNS

The COMPRESS operation will compress elements along the first axis, that is the leftmost index, according to the integer vector <right>. The element of the first axis will be retained if the appropriate <right> element is True, and will be elided if it is False.

This means that all the columns indicated by False in the <right> argument will be deleted.

The <right> argument must be INTEGER only vector, and has to have the same number of elements as is the number of columns (the first axis length) in the <left> space.

For example: **`25 INTERVAL (5 5) RESHAPE (1 0 1 1 0) COMPRESS.`** will give:

```
1 3 4
6 8 9
11 13 14
16 18 19
21 23 24
```

<left> <integer right> COMPRESSROWS => <result>

COMPRESSLASTAXIS

The COMPRESSROWS operation will compress elements along the last axis, that is the rightmost index, according to the integer vector <right>. The element of the last axis will be retained if the appropriate <right> element is True, and will be elided if it is False.

This means that all the rows indicated by False in the <right> argument will be deleted.

The <right> argument must be INTEGER only vector, and has to have the same number of elements as is the number of rows (the last axis length) in the <left> space.

For example: **`25 INTERVAL (5 5) RESHAPE (1 0 1 1 0) COMPRESSROWS.`** will give:

```
1 2 3 4 5
11 12 13 14 15
16 17 18 19 20
```

<left> <right> REPLICATE => <result>

/../

HAS TO BE FINISHED.

Function definition

FUNCTION

:

NEADIC

ARGS <x> FUNCTION

VARS <x> FUNCTION

ARGS <x> VARS <y> FUNCTION

VARS <x> ARGS <y> FUNCTION

NILADIC

ARGS 0 FUNCTION

VARS <x> NILADIC

VARS <x> ARGS 0 FUNCTION, ARGS 0 VARS <x> FUNCTION

MONADIC

ARGS 1 FUNCTION

VARS <x> MONADIC

VARS <x> ARGS 1 FUNCTION, ARGS 1 VARS <x> FUNCTION

DYADIC

ARGS 2 FUNCTION

VARS <x> DYADIC

VARS <x> ARGS 2 FUNCTION, ARGS 2 VARS <x> FUNCTION

TRIADIC

ARGS 3 FUNCTION

VARS <x> TRIADIC

VARS <x> ARGS 3 FUNCTION, ARGS 3 VARS <x> FUNCTION

RETURN => <function results>

RETURN will return from a functional sentence wherever it may be in that sentence. It will return as many data as are on the stack before RETURN.

YIELD => <function results>

BEWARE: YIELD is in early testing phase.

Allows continuations. When recalled, the functional sentence will continue after the YIELD. All elements on the function stack (as with RETURN) will be returned to the caller, and the function stack will be left empty for the continuation. A functional sentence with YIELD (or YIELDTO) may be regarded as a kind of “co-function” or “co-routine”.

YIELD is similar to RETURN, as it will return from the function, but it will preserve the position last executed. When invoking the function again, it will continue where it stopped, i.e. just after the YIELD. The function will always at any invokation take the number of arguments defined (ARGS). This has to be taken in account when writing the continuation after YIELD!. In a way, it has to be regarded as if the continuation after YIELD is the beginning of the function.

Beware that between consecutive invocations of the function stopped by YIELD all local variables (VARS, indicated by &<num>, i.e. &1, &2...), which are kept on stack, lost. However, all the local names (indicated by .<name>) are preserved. The context stack is not preserved if the function’s continuation was called from another context. Therefore changed meanings of names used in the continuation after YIELD can be used as parameters for the function continuation.

Be very carefull with continuations, specifically keep track of the stack if you use functions of different number of arguments. Beware again that the stack of the yielding

function will not be restored, but that the YIELD will, the same as RETURN, return as many data as are on the stack before YIELD.

<function> YIELDTO => <function results>

BEWARE: YIELDTO is in early testing phase.

The functionality of YIELDTO is similar as that of YIELD, only the <function results> will be returned to a specified function. That specified function will then continue either from the begining (if it was first called, or does not have YIELDs), or just after it's own YIELD. A functional sentence with YIELD (or YIELDTO) may be regarded as a kind of "co-function" or "co-routine".

As opposed to YIELD, which behaves like a normal RETURN, returning from the functional sentence to its caller, YIELDTO is actually a Go To. The other function will take over the return address, and therefore return to the caller of the function invocing YIELDTO. However, this function will keep the continuation after the YIELDTO, and any invocation of it will continue after that verb.

Be very carefull with continuations!

Beware that with YIELDTO you actually do something which C language would call a "longjmp", i.e. that you enter a completely independent functional sentence, perhaps in the middle of it (after its YIELD or YIELDTO verb). So anything you have on the stack in this function (including local variables &<num>) will be inherited by the function you transfer the control to.

the stack of the yielding function will not be restored, but that the YIELD will, the same as RETURN, return as many data as are on the stack before YIELD.

FUNCTIONEND => <function results>

;

Applying functions to data

<(arguments)> <function> EXECUTE => <result of execution>

!

EXEC

asdf

Virtue allows full recursion of any combination of functions, multiple recursions in one function, cross recursions etc. However, the depth of recursion is, unfortunately, very closely related to the Operating System on which Virtue is being used. Some operating

systems will allow very deep recursions (e.g. SunOS 4.1.1 from 1987), but some will start making segmentation faults after just ten(s) of recursion depths (e.g. FreeBSD, even the newest version). Future versions/revisions of Virtue will constantly strive to relieve this OS problem.

<(arguments)> <function> <count> LOOP => <result of looping>

The LOOP verb in Virtue is special as it's execution is guaranteed to be strictly serial, i.e. each call of the argument function will be certainly executed one after the other, as the loop counter changes. This is an important feature for algorithms which depend on the seriality of execution, as for example when getting input, or updating a global variable, or when the output of a previous iteration of the function is the input for its next iteration.

<left> <function> EACH => <result>

Beware: EACH, as opposed to LOOP may be executed in parallel on several processors, and therefore the execution sequence may be completely "random". This is important to know, as for example if the function is changing global variables based on its work, the end result in the global variable can be the result of any of the executions. If the algorithm needs strictly serial processing, use LOOP, if possible. Beware, however, that both the arguments and the result of EACH and LOOP are quite different.

<left> <mask> <function> MASK => <result>

The Triadic MASK has two possible modifiers: WRAP and REFLECT. The MASK (with or without modifiers) will never recurse into substructures, as this is not consistently implementable. For example:

```
1                (2  3 (4 5))                (6 7)
(8 9)            10                          11
12               ((13 (14 15)) 16 (17 18))   (19)
```

can obviously not be recursed by MASK into substructures in a consistent way. The FUNCTION which is MASKed will get an argument array of the shape of the mask applied, with substructures of the original array being the substructures of the FUNCTION argument array. However, the scalar MULTIPLY used by the MASK operation will descend into the substructures. Therefore

```
2 (3 3) RESHAPE FUNCTION ... ; MASK
```

applied to the example array will, for the first place [index (1 1) or 1i1] give the following argument to the FUNCTION:

```
0          0          0
0          2          (4 6 (8 10))
0          (16 18)    20
```

MASK will beyond the edges of the array take (make) empty (' ' or 0) elements.

Example:

```
1 2 3
4 5 6
7 8 9
```

1 (3 3) RESHAPE FUNCTION ... ; MASK will at the first position, where number 1 is centered, give the FUNCTION the following argument as input:

```
0 0 0
0 1 2
0 4 5
```

and for the last:

```
5 6 0
8 9 0
0 0 0
```

<left> <right> WRAP MASK => <result>

WRAP MASK will beyond the edges take the dimensionally oposite elements. It wraps diagonally around the masked array.

Example:

```
1 2 3
4 5 6
7 8 9
```

1 (3 3) RESHAPE FUNCTION ... ; WRAP MASK will at the same position give FUNCTION the argument:

```
9 8 7
3 1 2
6 4 5
```

and for the last:

```
5 6 4
8 9 7
2 3 1
```

<left> <right> REFLECT MASK => <result>

REFLECT MASK will in the same situation reflect the existing elements. It reflects of straigth lines; in corners, i.e. diagonally, it reflects the reflections, i.e. areas from outside the boarders of the MASKed array will first reflect the content (inside the boarders) and then rereflect it in the same way to fill up the argument array.

Example:

```
1 2 3
4 5 6
7 8 9
```

1 (3 3) RESHAPE FUNCTION ... ; REFLECT MASK will give for the same position the FUNCTION input:

```
1 1 2
1 1 2
4 4 5
```

and for the last:

```
5 6 6
8 9 9
8 9 9
```

1 (5 5) RESHAPE FUNCTION ... ; REFLECT MASK will give for the same position the FUNCTION input:

```
5 4 4 5 6
2 1 1 2 3
2 1 1 2 3
5 4 4 5 6
8 7 7 8 9
```

On large MASKed arrays the REFLECTion and WRAPing is more obvious! REFLECTion could be used for bumping certain structures off the walls, WRAPing will make a n-dimensional spherical space, and structures will continue wrapping around from bottom to top, left to right and diagonally "through" the walls.

Manipulating functions

<function> OPERATOR => <operator>

The OPERATOR verb will transform a FUNCTION, which can be manipulated on the stack before being explicity EXECUTed (or EACHed, LOOPed etc.), into an operator, a verb that is immediately executed as it is encountered in the text. That is, a saved operator (a function defined as an operator) will be immediately executed as soon as it is recalled from memory. A saved (non-operator) function will be put on stack when it is recalled from memory.

The consequence of immediate execution of an operator function is that it can not be used in any construct needing a function, as it behaves as all inbuilt Virtue verbs.

For example, consider the following sentence, saving the operator function in memory: **NILADIC 3 PITIMES SQRT; OPERATOR @Sqrt3pi SET.** After defining this new operator, the one word sentence **Sqrt3pi.** will give the result 3.069980124.

However, if it is necessary to use a function already defined as an operator on the stack (that is for EACH, LOOP etc.), the function can be retrieved onto the stack by the verb GET. So, for the above example, we can write **`@Sqrt3pi GET.`**, or even **`'Sqrt3pi' GET.`** The result will be the original: **`NILADIC 3 PITIMES SQRT; OPERATOR`**.

works on: functions

<left> <function> RESULT => <result>

The verb RESULT will apply the function to its <left> argument(s), but before doing so it will check if the same function was already applied to the same argument(s). In that case the RESULT of the previous function application will be left on stack. If there is not yet a result of the application of this specific function to this specific argument(s), the RESULT will evaluate the function (in the same manner as EXECUTE does), return the result on the stack and save the same result in memory for further usage.

RESULT is a verb wich allows semi-automatic “memoisation”. It is fully automatic, but the user has to request it where appropriate by using the RESULT verb.

Due to the fact that the result of the <function> may not be always the same for the same argument (<left>), as, for example in the case of using random numbers inside the function, printing out results inside the function and other similar cases, Virtue will in such cases fully automatically skip the memoisation part and the RESULT will behave exactly the same as EXECUTE. To be able to do that, the internal information Virtue keeps about all processable elements (including functions) includes also several internal flags about if the function writes to global variables, reads or writes to the user interface, uses stochastic processing etc.

The real text of the function as it is on the stack is used. Therefore differently named functions with the same algorithmic content will automatically be regarded as same, independent of their names and independent of their internal comments.

It is important to note that there are cases in which a direct execution of a function will be faster than its memoised execution. This is true for example if the <left> argument(s) are big spaces, and the <function> does not do much processing. To enable the memoisation Virtue converts both the argument(s) and the function into an address represenatation, and than consults a fast hash based memory if there is already something remembered under this composed name. Naturally, if the composition of the name (which has to take into account each individual element of a space as well as its general structure, and the function) takes so long time that it would be longer than the time necessary to actually execute the function, than the use of memoisation (the verb RESULT) has no sense. This decision is left to the user/ instructor (programmer).

However, the use of memoisation can often drastically shorten the overal execution time. In certain cases, as for example the usage of a simple recursive Fibonacci series calculation, the usage of the verb RESULT can speed up the execution almost unbelievably. So for example, without memoisation the double-recursive Fibonacci altogithm [fib(0 or 1) = 1; fib(n) = fib(n-2) + fib(n-1)] requires for fib(30) on a modern 2 GHz Intel/AMD processor over 20 seconds. In the same time (21 seconds) using the memoisation with the verb RESULT, a 20 MHz Motorola 68020 processor (e.g. a Sun 3/60 workstation from 1987) will calculate all the first 1200 (!!!) Fibonacci numbers (around the 1200th fibonacci number the 64-bit IEEE

floating point representation gets into infinity). A subsequent request for all the first 1200 fibonacci numbers on this 20MHz processor will be finished in less than 14 seconds.

To repeat: Calculating First 30 Fibonacci numbers without memoisation on 2000 MHz (2 GHz) processor takes the same amount of time as calculating First 1200 Fibonacci numbers with memoisation on 20 MHz processor using the same simple recursive algorithm! (Functions do not have to be in any way adapted to use the RESULT, i.e. memoisation, on them.)

<left> <function> REEVAL => <result>

There are occasions when a REEVALuation of the memoised result (see RESULT) is necessary. This could be for example if using a global constant in the memoised function, which constant had to be changed. This type of usage can not be automatically detected by Virtue without significantly degrading the usability of memoisation, therefore the REEVAL verb is provided for the user to cope with such special situations.

<ascii left> ASSEMBLE => <function(s)>

The ASSEMBLE verb will take any ASCII (character) vector (string) and convert it to one or more functions, which will be left on the stack, with the last on top of stack. Actually it assembles the sentences in the string into executable functions on stack.

If the character string contains only one Virtue sentence, with or without the sentence-finishing '**.**', this sentence will be put on stack as a function. This means that a sentence which begins with a FUNCTION will be translated to internal form and transferred to the stack as an executable function. Therefore the result of ASSEMBLE and EXECUTE on a string containing only a function will be exactly the same. That is: **'MONADIC NOT;' ASSEMBLE** is synonimous to **'MONADIC NOT;' EXECUTE**. Though, opposed to EXECUTE, ASSEMBLE will ignore all written after the FUNCTIONEND, i.e. after '**;**'. That is: **'MONADIC NOT; 1 2' ASSEMBLE.** will result only in the function **MONADIC NOT;** on stack! EXECUTE will make three entries on stack.

However, if the sentence does not start with a FUNCTION, ASSEMBLE will make a NEADIC wrapper function around everything in each sentence encountered. So, for example **'10 INTERVAL' ASSEMBLE** will result in the function **: 10 INTERVAL;** on the stack.

As any character string (which, naturally, has to be understandable in the Virtue language and ortography, i.e. properly formed) can contain several individual sentences, those Virtue sentences will be converted to individual FUNCTIONs, one per sentence, which are then put on the stack. As the order of sentences in the character string is from left to right, and the stack is last-in-first-out (or, the same, first-in-last-out), this means that the last sentence in the character string will be on top of the stack.

For example:

'1 2 3 . 10 RESHAPE DYADIC ADD;. ROLL. MONADIC FLOOR;' ASSEMBLE.

will result in the following four stack entries:

```
[1]  : 1 2 3;
[2]  : 10 RESHAPE DYADIC ADD;;
[3]  : ROLL;
[4]  MONADIC FLOOR;
```

[4] is in this case the top of stack, [1] the bottom.

works on: ASCII vector

<left> DISASSEMBLE => <ascii representation of left>

DISASSEMBLE takes any <left> argument and converts it to an ASCII vector which is the input/output (human readable) representation of the argument. The result is the same as if the following statement would be written: ***<left>*** **'temp.file' WRITE 'temp.file' TEXTREAD**.

The statement ***<left>*** **DISASSEMBLE EXECUTE** is redundant, the result is *<left>*.

works on: anything on stack

Memory dictionaries (contexts)

Virtue memory is defined with contexts. The contexts are kept as a stack, where the global context, available always at the main prompt, is on the bottom, and each individual function invocation pushes its own local context on top of all the others.

However, the contexts are individual to functions, as long as they are preserved, and are not individual to each recursive invocation. This means that all the recursive invocations of a function will keep the same context for names. A very deep recursive invocation could for example set some local name (variable) to some meaning (value), and when the recursion returns to the beginning, after the recursion that first invocation of the function would find this same meaning in that local name. And that name and its meaning would be available for any invocation of the same function, from whichever other context. If you need recursively local variables in your function, use the L<number> form. Those are real variables kept on the function execution stack, so they are completely local to each recursive or other invocation of a function and can never be seen from anywhere outside or deeper. Even functions written inside a function having VARS will not be able to access them.

The prefixes to the names (‘’, ‘_’, ‘$’ and ‘.’) define the way the context (symbol table) stack is searched. The empty form of the name-prefix (‘’) will search for the name only in the global context (the stack bottom); ‘.’ will search for the name only in the current local (topmost) context, and ‘_’ and ‘$’ will search for first occurence of the name in each context from the top (current) to the bottom (global), stopping in the context in which it is first found.

The prefixes of the names in addresses (‘@’, ‘@_’, ‘@$’, ‘@.’) define in which context the values, meanings, will be remembered. ‘@’ and ‘@_’ will define an address in the global

(bottom of stack) context; '@$' and '@.' will define an address in the current local (top of stack) context.

The verb CONTEXT and the phrases it forms allow explicit manipulation with the name spaces, i.e. contexts.

<left> DICTIONARY

CONTEXT

The CONTEXT verb will take the <left> argument to indicate which context shall be made the current, and therefore pushed on the top of the *context stack*.

Depending on the type of the argument we can have:

<function> CONTEXT: the context pushed will be that of the function. Then the function will be executed, to allow the memory to be updated. The context will be available as long as the function is anywhere in the memory. To make a temporary context for the rest of the function or sentence you could write something like **`:'Hello' @a SET 'World' @b SET; CONTEXT a b CATENATE PRINT.`**

<address> CONTEXT: the context pushed will be that in which the addressed name resides.

PREVIOUS DICTIONARY

PREVIOUS CONTEXT

If you switch to a previous dictionary, all names which have been resolved already inside your context will be still available from that context. However, variables which have been already assigned values in the present context will be preserved till the end of the sentence, although the old dictionary will not be any more available as a dictionary.

Beware, PREVIOUS DICTIONARY pops the top dictionary (context) form the top of the symbol table stack.

GLOBAL DICTIONARY

BASIC DICTIONARY, GLOBAL CONTEXT, BASIC CONTEXT

Will push the global dictionary to become local. Beware that now all '.' and '$' will actually end up in the global context, as it is now become also local to the sentence being executed.

NEW DICTIONARY

NEW SMALL DICTIONARY, NEW CONTEXT, NEW SMALL CONTEXT

Will push a completely fresh empty balanced binary tree dictionary on top of the symbol table stack

NEW LARGE DICTIONARY

NEW LARGE CONTEXT

Will push a completely empty new large hash table based dictionary on the top of the symbol table stack.

Manipulating names and their meanings

<left> <address> ASSIGN => <left>

assigns a value (any kind of Virtue data) to the variable whose address is the <right> argument. As opposed to SET, the stack is not changed, i.e. after assigning the <left> to the <address>, the left argument is left on stack.

This is very convenient if some interim results have to be saved, or any kind of more/ different processing done on the data. So, for example, when defining a FUNCTION as an OPERATOR, it may be convenient to keep the non-operator function in a separate memory (another address, with its other name), as Virtue verbs can not use operators as arguments (the operator immediately operates on the available data. A function has to be EXECUTEd, and therefore may be used for MASK, EACH etc.

For example, to define such a function and operator we could write: **`MONADIC PROBABLY NOT; @.maybe ASSIGN OPERATOR @Maybe SET.`**

<left> <address> SET =>

sets a value (any kind of Virtue data) into the variable whose address is the <right> argument. As opposed to ASSIGN, both arguments to SET are “consumed” by the verb, and nothing is left on stack. SET accepts only ADDRESS data as its right argument, i.e. only data produced by the @ operator.

This is a very convenient way to end a sentece when we are not interested in its immediate results, as for example when defining functions or operators, or when working with huge spaces, whose intermediary results may be too verbose for interactive work.

For example: **`(0)[100 100 100 100] @_empty_space SET.`**

<left> AT => <address>

<left> GET => <content of left>

<...> <left> DO => <result of execution of content of left>

<address> FORGET =>

forgets the data saved in memory under a name (address), and the name is unassigned till further assignment / setting of new data. As with DO, GET, AT, FORGET accepts as its argument both ADDRESS (i.e. **`@somewhere`**) and character vectors (i.e. **`'somewhere'`**).

Recalling data from an unassigned name will raise the exception "*23 - value has been passed an empty variable, i.e. can not use un-initialized variable.*", so after FORGET the meaning of the name is forgotten.

WIPE

Flow control

<left> <right> IF => <result>

<address> JUMP =>

<left> CHECK =>

<operator/function> IF_YES =>

?Y

<operator/function> IF_NO =>

?N

Stack manipulation verbs

<left> <right> LEFT => <left>

DISCARD, POP

Discard the right argument, i.e. the Top of Stack value.

The same result would be to apply the sentence **`SWAP RIGHT.`**

The name POP is synonymous with the same operator in Postscript.

<left> <right> RIGHT => <right>

Discard the left argument, i.e. the argument one below the top of stack.

The same result would be to apply the sentence **`:DISCARD; DIP.`**, or the sentence **`SWAP LEFT.`**

<left> DUPLICATE => <left> <left>

DUP

Duplicates whatever is on the top of stack.

For example, to get a square of any number we could use the following function:

```
MONADIC DUP MULTIPLY;
```

<left> <right> SWAP => <right> <left>

EXCHANGE, EXCH

Swap the two top elements on stack.

For example, if we would like to make an array of zeros the same shape and size as the input array, the following function could be used: **`MONADIC DUP SHAPE 0 SWAP RESHAPE;`**.

The name EXCH is synonymous with the same Postscript operator, EXCHANGE is the same word full spelt out.

COUNTSTACK => <integer>

Count the number of elements on the stack accessible by the present function/level. The result of the COUNTSTACK verb is a count of the stack elements accessible. For example:

`1 2 3 4 5 COUNTSTACK.` will give **`5`**. If we continue as following: **`:1 2 3 4 COUNTSTACK;!.`** will give **`10`** (the 5 original, the result of counting, and the 4 elements put on stack by the neadic function). As we used a Neadic function, a function which will take as

many arguments from the stack as it wishes, there is no new stack frame, and therefore the COUNTSTACK counts the whole stack. However, if we would now input the sentence: **`NILADIC 1 2 3 4 COUNTSTACK;!.`** the result would be, independent of the already existing 11 elements on stack, **4**, as a new stack frame was defined by the function (see ARGS ... FUNCTION).

MARK => <mark>

Puts the operator MARK on the stack. This is then used as a stack mark for the COLLECT, COUNTTOMARK and CLEARTOMARK instructions.

To get rid off the MARK just DISCARD it. However, the COLLECT verb will discard the MARK automatically.

<mark>... <...> COLLECT => <result>

COLLECT collects all stack elements from the <mark> (see MARK) to the top of stack into a vector. That is, the result of **`MARK 1 2 3 4 COLLECT.`** will be the vector **`(1 2 3 4)`**. Or, the result of **`MARK (1 2 3) 'Hello' 1i1j2 'Dolly' COLLECT`** will be the vector **`((1 2 3) 'Hello' 1i1j2 'Dolly')`**.

<mark>... <...> COUNTTOMARK => <integer>

Counts all elements on the stack above the latest MARK.

COUNTTOMARK will not erase the stack MARK. To do so, erase the MARK with DISCARD.

<mark>... <...> CLEARTOMARK

Clear all elements from the "highest", i.e. latest, MARK on the stack up to the present top of stack, and leaves the stack pointer pointing at the MARK.

CLEARTOMARK will not erase the stack MARK. To do so, erase the MARK with DISCARD.

<...stack> <right> <function> DIP => <...result> <right>

applies the function to the argument(s) on stack before the <left> argument, i.e. below the Top of Stack.

For example, let us put four elements on stack: **`1 2 3 4.`** The 4 is at the Top of Stack, the 1 at the bottom. Now applying the sentence **`:DISCARD DISCARD; DIP.`** we will get only two elements on stack: **1** at the bottom and **4** on the top of it. The result of the sentence **`1 2 3 4 DYADIC SWAP; DIP.`** would be **`1 3 2 4`** on the stack.

<...stack...> <function> <depth> DEEP => <...result...>

CLEAR

Clear the whole stack frame. When used by the user outside functions, it will clear all elements on stack. If used in a function, it will clear the function's stack frame (the arguments and/or any additional data produced by the function on stack).

For example, to clear the top 7 elements on stack you could write:

```
ARGS 7 FUNCTION CLEAR; EXECUTE.
```

Other

NOOP

No Operation. Do nothing.

Virtue keeps user comments in NOOPs, to be able to preserve them in internal form of the sentences. Therefore, as any respectful NOOP, the comments do nothing in the programme, except to explain to a human what the programme does in a computer.

OFF

QUIT, ENDPROCESS

END

.

End of sentence. This is commonly spelt by just using the punctuation mark, the dot '.'.

Syntax modification verbs

The syntax modification verbs PREFIX, INFIX and POSTFIX allow the Virtue verbs being defined to be differently contextually (syntactically) bound to their objects, i.e. arguments. Therefore all three of the common expression ways, the prefix, infix and postfix notation is possible, and can be freely intermixed in any Virtue text. However it is important, when writing sentences with different boundings (-fixes) to understand that Virtue will always start the execution of any verb for which enough arguments are provided on the stack.

<function> POSTFIX

The verb POSTFIX is actually a NOOP, no-operation, does nothing, as the default behaviour of Virtue and all of its <function>s to behave as postfix, i.e. Reverse Polish Notation, verb at the end, after the arguments.

The POSTFIX verb is provided for orthogonality. The difference between a NOOP and a POSTFIX verb is that a NOOP is niladic, has no arguments, whereas a POSTFIX verb will complain with a Warning if its argument is not a function.

<function> PREFIX

The PREFIX verb wil change the definiton of the <function> in such a way that when it is invoked (executed) it will be pushed on the *deffered execution stack*, and Virtue will start to collect as many elements on the stack as the function has arguments. After collecting the appropriate number of arguments on stack, Virtue will automatically pop the <function> from the deffered execution stack and process it as a normal postfix verb, i.e. using the stack for the arguments.

For example: **`DYADIC ADD; PREFIX EXECUTE 1 2 .`** will give the result **3**. **`DYADIC ADD; PREFIX OPERATOR @Add SET. Add 1 Add 2 3.`** will give the result 6. Beware when mixing verbs with different -fixes, i.e. PREFIX and POSTFIX notation, as, for example applying the aforedefined 'Add' in the following sentence: **`Add 2 3 4 *.`** will give the result 20 as the Add will be applied as soon as there are enough arguments on the stack for the given <function>. Therefore **`Add 2 :3 4 *;!.`** will give an Error, as a FUNCITION is applied to the verb ADD.

<dyadic function> INFIX

The INFIX verb will change the definition of the <function> in such a way that when it is invoked (executed) the topmost argument on the stack will become the left argument of the infix verb, the <function> will be pushed on the *deffered execution stack*, and processing will continue with the next verb. As soon as anything is pushed on the stack, thedeffered infix function will be executed as a normal postfix verb, i.e. using the stack for the two arguments.

For example: **`1 DYADIC ADD; INFIX EXECUTE 2 .`** will give the result **3**. **`DYADIC ADD; INFIX OPERATOR @+ SET. 1 + 2 + 3 .`** will give the result **6**. Beware when mixing verbs with different -fixes, i.e. INFIX and POSTFIX notation, as for example the sentence **`2 + 3 4 *.`** will give the result **20** as the '+' will be applied as soon as the number 3 is pushed onto the stack. Therefore **`2 + :3 4;!.`** will give an Error, as a FUNCTION is applied to the verb ADD.

It is important to understand that an INFIX verb can have only and only two arguments, the left and the right. Therefore any non-dyadic <function> applied to the INFIX verb will raise and Error.

Virtue programme debugging

DESCRIPTION:...

TRACE

After the TRACE command, Virtue will leave a written trace of everything it does, by printing out the word just to be executed and the level of recursion, program counter and stack pointer. TRACE will trace all what was written in the (functional) sentence it was invoked within, but not any other sentences, like the functions called. Virtue will stop tracing as soon as it finishes the sentence in which TRACE was asked for. If you want to trace all what is going on, use TRACEALL.

Trace is very usefull for Virtue sentences "debugging" and inspecting what exactly is being done. The trace will not include the data on the stack, as these may be extremely large and their printout would spoil the intention of TRACE. If you need to examine the data generated, use the STEP command.

TRACE can always be revoked by the RUN command. As TRACE is active only on the sentence it was invoked within, the RUN shall also be inside that sentence. The rest of this sentence after RUN will be executed without the trace. This is very practical when you want to trace just parts of a Virtue programme - put them into the right places.

TRACE inside a sentence will not

TRACEALL

STEP

Enable the stepping mode. After each word of the sentence being exeduted Virtue will stop to allow the user to inspect all data and stack, do additional manipulation and continue stepping. STEP will step only through the sentence in which it was invoked, and will not step through possible invocations of other functional sentences. If you want to step through everything that is happening, use STEPALL.

When in STEP mode, Virtue wil print out a trace and change its prompt to **`Inspect [x]:`**, where the x is the present top of stack pointer, i.e. the amount of data on the stack.

Virtue will not step through words inputed at the Inspect prompt.

When in stepping, just press the ENTER (RETURN, SEND) key on the keyboard to continue to the next instruction. If you write an punctuation mark (end of sentence '.'), Virtue will output top of stack content, just the same as in normal interaction. The stack frame is protecting from changing stack content of the calling sentences, but you can do anything you like on the stack of the sentence you are inspecting. If at the user interaction level, there is no difference between the stack frame and the whole stack.

Any sentence inputed while in the stepping mode can be finished just by pressing the "enter" key (NL ascii character, **`'`0#00001010`'`**, decimal 10). This is the same as with the INPUT and TEXTINPUT verbs. This means that in the "Inspect" mode it is not possible to enter multiple line sentences. This does not apply to comments and character strings or character literals

The trace (see TRACE) always show the word which is not yet processed. So you can always examine the stack and all names, variables etc. knowing which next operation will be executed. An example:

```
Virtue [0]> STEP 10 20 30 40 .

Trace ( 0:  2): . SCALAR: 10
Inspect [0]:

Trace ( 0:  3): . SCALAR: 20
Inspect [1]: .


 10
Inspect [1]:
```

To continue with normal execution use the command RUN.

STEPALL

To continue with normal execution use the command RUNALL. If you use the command RUN, Virtue will not step through the sentence in which you invoked RUN.

RUN

RUNALL

Special ortographical operators

@<name>

Pushes the address of the name on the stack. This name may be first used or already known. If a new name is used (one not ever yet mentioned to Virtue during the user session and not already in memory), a new symbol table (dictionary) entry will be made, with no content. If the name is already known, the content will not be changed. This is and _ADDRESS_.

For example **`1 @a SET.`** assigns the value **1** with the name **a**, technically putting the content **1** into the memory area provided for the name **a** by the address **@a**.

::<operator>

Uses the Virtue inbuilt meaning of the <operator>, independent of its redefinitions in dictionaries. This is actually a kind of “name escape”.

For example: **`DYADIC MULTIPLY; OPERATOR @+ SET. 5 6 +.`** will give **`30`**, as the verb '+' was defined to be a dyadic function which does multiplication. However, independent of the previous definition of the verb '+' to be multiplication **`5 6 ::+.`** will give **`11`**, as the '::' ortographical operator "converted" the name '+' to mean the original Virtue inbuilt addition operation.

(<x>)

Vector notation. An vector is a single-dimensional space, which however may have subspaces. This means that the (<x>) notation is recursive. An empty vector '()' is equal to NIL.

Examples: **`()`** - a NIL; **`(1)`** - a single element vector with the scalar 1 as its only element; **`(1 2 3)`** - a three-element vector of scalars; **`(1 (2 3) ((4 5 6)))`** - a three-element vector consisting of a scalar 1 as first element, a two-element vector of scalars 2 and 3 as the second element, and a one-element vector consisting of a three-element vector of scalars 4 5 and 6 as its third element.

(<x>)[<y>]

A shorthand notation for (<x>) (<y>) RESHAPE. The one-dimensional list of scalars inside the square brackets '[' and ']' defines the shape vector of the one-dimensional vector given by the parentheses '(' and ')' (see above the definition of the syntax '(<x>)').

For example: **`(1 2 3 4 5 6) [3 2].`** will produce the two-dimensional array:

```
1 2 3
4 5 6
```

'<characters>'

Start or end a character vector (string). A character string in Virtue may include all characters, even controls and newlines.

"<comment>"

A comment. Anything may be put inside the double quotes, including control characters and newlines.

Virtue nouns (constants)

All Virtue constants are NAMED, which means that, if no data changing operations are done on them, they will be shown to the user (outputed) in the form they were initially inputed. This means, for example, that the BOOLEAN value false may be outputed as F, #F or #FALSE, depending on the way it was written by the user.

All Virtue constants are indicated by the character # in front of them, except the #F, #FALSE and the #T, #TRUE, which may also be written only as **F** and **T**. These are provided as Virtue will output the BOOLEAN values as **F** and **T**, so they will be READ back properly.

#NIL

NIL is a special datum which indicates N*ih*IL, nothing. NILs can not be part of an array by themselves, though it is possible, if necessary, to enclose them, and use them as such.

#NIL #NIL CATENATE gives #NIL, ***<vector>* #NIL CATENATE** gives <vector>, **#NIL *<vector>* CATENATE** gives the <vector>, **#NIL *<scalar>* CATENATE** gives <scalar>, but ***<scalar>* #NIL CATENATE** gives a one element vector of the scalar.

The #NIL will always be written in structured writing (like WRITE) as **()**, as long as it is part of a structure. A selfstanding NIL will be WRITEd (written) as #NIL. On the user interaction terminal it will be printed just like a simple dot '**.**'.

#END

...

#FALSE

#F, F

A BOOLEAN constant meaning false. The value is 0. To see the value in numeric form, use the verb NUMEBER, i.e. **F NUMBER** gives **0**.

#TRUE

#T, T

A BOOLEAN constant meaning true. The value is 1. To see the value in numeric form, use the verb NUMEBER, i.e. **T NUMBER** gives **1**.

#PI

#pi

The mathematical constant π = 3.14159265358979323846.

#2PI

#2pi

Two times π = 6.28318530717958623. For multiples of #PI, specifically if a vector/ array/space of different multiples has to be generated, or the needed multiple of #PI is not

provided in the Virtue constants, use the verb PITIMES. That is **#2PI** is the same as **2 PITIMES**, and **#PI_4** is the same as **4 RECIPROCAL PITIMES** (that is **0.25 PITIMES**).

#PI_2

#pi_2

Half π = 1.57079632679489661923.

#PI_4

#pi_4

Quarter π = 0.78539816339744830962.

#1_PI

#1_pi

The reciprocal value of π = 0.31830988618379067154.

#2_PI

#2_pi

Two divided by π = 0.63661977236758134308.

#2_SQRTPI

#2_SQRTpi

Two divided by the square root of π = 1.12837916709551257390.

#SQRT2

Sqare root of 2 = 1.41421356237309504880.

#SQRT3

Square root of 3 = 1.73205080756887719.

#SQRT1_2

#1_SQRT2

Square root of one half, or, if you prefer, the reciprocal of square root of 2 = 0.70710678118654752440.

#E

#e, #EULER, #NAPIER

The mathematical constant e = 2.7182818284590452354.

#LOG2E

#LOG2e

The logarithm base 2 of e = 1.4426950408889634074.

#LOG10E

#LOG10e

The logarithm base 10 of e = 0.43429448190325182765.

#LN2

#LOGE2, #LOGe2

The natural logarithm of 2 = 0.69314718055994530942.

#LN10

#LOGE10, #LOGe10

The natular logarithm of 10 = 2.30258509299404568402.

#ATANH

The constant -7. Used for the CIRCULAR operation, denoting the Hyperbolic Arcus (inverse) Tangens trigonometric operation.

#ACOSH

The constant -6. Used for the CIRCULAR operation, denoting the Hyperbolic Arcus (inverse) Cosinus trigonometric operation.

#ASINH

The constant -5. Used for the CIRCULAR operation, denoting the Hyperbolic Arcus (inverse) Sinus trigonometric operation.

#SQRTX2m1

The constant -4. Used for the CIRCULAR operation, denoting the Square Root of (X Times X Minus 1).

#ATAN

The constant -3. Used for the CIRCULAR operation, denoting the Arcus (inverse) Tangens trigonometric operation.

#ACOS

The constant -2. Used for the CIRCULAR operation, denoting the Arcus (inverse) Cosinus trigonometric operation.

#ASIN

The constant -1. Used for the CIRCULAR operation, denoting the Arcus (inverse) Sinus trigonometric operation.

#SQRT1mX2

The constant 0. Used for the CIRCULAR operation, denoting the Square Root of (1 Minus X Times X).

#SIN

The constant 1. Used for the CIRCULAR operation, denoting the Sinus trigonometric operation.

#COS

The constant 2. Used for the CIRCULAR operation, denoting the Cosinus trigonometric operation.

#TAN

The constant 3. Used for the CIRCULAR operation, denoting the Tangens trigonometric operation.

#SQRTX2p1

The constant 4. Used for the CIRCULAR operation, denoting the Square Root of (X Times X Plus 1).

#SINH

The constant 5. Used for the CIRCULAR operation, denoting the Hyperbolic Sinus trigonometric operation.

#COSH

The constant 6. Used for the CIRCULAR operation, denoting the Hyperbolic Cosinus trigonometric operation.

#TANH

The constant 7. Used for the CIRCULAR operation, denoting the Hyperbolic Tangens trigonometric operation.

Variable Constants

The Constant Variables, or, if you prefer the Variable Constants, are internal Virtue constants which modify specific behaviours, as for example the comparison tolerance. These variable constants are all denoted, as all other constants, by the number sign (#), but, as opposed to the numerical and programming constants, whose value can not be changed, the value of these can. Their value is changed by simply ASSIGNing or SETting the value into their address (@ or AT).

Beware, and be careful when using the Virtue internal constants. Though, Virtue will take precausions not to use the #CT variable constant out of range, the behaviour may not be what you expected. Virtue will, as much as possible, make some sense of the value given. For example, the Comparison Tolerance will work correctly (although the operations will get a different meaning) with any _LOGICAL_ number, and negative numbers will be converted to positive. Bigger (positive or negative) values than 1.0 will be forced into _LOGICAL_ by the principles Virtue adheres to:

Just be careful!

#CT

Comparison Tolerance. The default value is depending on the size of the computer internal floating point representation (Virtue can be installed to use 32-bit C “float” floating point numbers and 64-bit C “double” floating point numbers). According to the IEEE-754 standard, 64-bit floating point numbers (the default for Virtue) can hold 15.59 decimal places, so the value of **`#CT`** is set to **`1.0e-15`**. For 32-bit floating point number Virtue installations the value of **`#CT`** is **`1.0e-7`**.

The value of #CT can be assigned just as assigning a value to any other variable (only the name begins with #). The user may want to change that value, as progressive mathematical calculations on floating point numbers may significantly degrade their precission. To still be able to cope with their unwanted inequalities, assigning a larger value to #CT will help resolve this inherent digital mathematics/computer flaw.

However, for special purposes, any value of #CT up to +/- 1.0 (REAL number!) will provide the output of #CT dependent operations to be inside the _LOGICAL_ range. For values larger than 1.0 the result of the #CT dependent operations may fall outside the range of _LOGICAL_ values, and certain rules of behaviour of _BOOLEAN_ and _LOGICAL_ operations may seem wierd. The #CT value outside the range [0, 1] will be forced to _BOOLEAN_ according to Virtue logical values rules, i.e. all values below zero are regarded as 0, all above one will be 1. A comparison tolerance of 1 can be interesting to use for special purposes, like getting all the compute errors as large as they get (up to 1.0). Beware, that then all comparison operations, floor and ceiling etc. will behave consistently, but quite differently.

So, for example, the result of **`1.0 @#CT SET 1.2 1.7 EQUIVALENT.`** is **`0.4`**.

A normal way of using #CT would be to set it to a higher value than the default, i.e.:

```
1.9e-13 @#CT SET.
```

PS: some of internal Virtue operations, like sinus and cosinus, have a much lower precision, so that specific implementation dependent constants allow Virtue to get the proper result for, e.g., **`-1 SQRT SQUARE`**, i.e. to get **`-1`** again.

TO BE ADDED

Multiple stacks shall be introduced!

The CONTEXTs (DICTIONARYs) have to be able to be put on execution stack, as to be manipulated and saved in dictionaries independent of the FUNCTION or the ADDRESS. In both cases the context is pushed on the context stack, whereas a unnamed DICTIONARY has to be able to figure, and be manipulated, on the *execution stack*! Such a DICTIONARY object is always SHARED!

Regular expression manipulation of string data (and as much as possible on all other types of data).

Additional stack manipulation, as in PostScript:

"COPY", "ROLL", MARK, "INDEX", "COUNT", COUNTOMARK, CLEARTOMARK

<left> SHARED

A flag shall be added to each object declared as SHARED, which disallows the reference counter, i.e. **`(1 2 3) SHARED DUP 2 + SWAP`** gives **`(3 4 5)`**, that is both TOS and TOS-1 are changed. This means that there is *only one* pointer is pointing to the SHARED object, and it is therefore instantly changed wherever it is mentioned. This is similar to the normal use of composite objects in the PostScript language[1].

The _SHARED_ object is actually analog to using pointers in conventional programming languages, as each duplication, saving etc. on the _SHARED_ type makes just copies of the pointer to the same *individual* structure.

If a atomic scalar (e.g. and REAL, INTEGER, ASCII) is proclaimed SHARED, it is converted to a single-dimensional one-element vector, which is than shared through the same "pointer".

FILE HANDLEs behave always as if they would have explicitly been SHARED.

<left> PROTECT

Protect the value of the object by disallowing its changes. It behaves like a kind of read-only flag. A SHARED PROTECTed object is strictly read-only in all its "incarnations", i.e. **`3 SHARED DUP 2 ADD`** is an Error!.

PROTECTed OPERATOR FUNCTIONs can not be used as data, i.e. they can only be executed.

1.http://partners.adobe.com/public/developer/en/ps/PLRM.pdf, p. 49.: "When a composite object is copied, the value is not copied; instead, the original and copy objects share the same value. Consequently, any changes made to the substructure of one object's value also appear as part of the other object's value."

<left> UNPROTECT

Unprotect the value of the object, so that it is again fully maniputable. A SHARED object can be again manipulated as described under “SHARED”.

<function> <depth> DEEP

The behaviour is the same as DIP, only DEEP is dyadic, indicating in the right argument the depth of stack from the top at which the <function> shall be performed. 0 DEEP is the same as EXECUTE, 1 DEEP the same as DIP. If the <depth> is larger than the stack frame in which it is executed, it is an Error!