Zorislav Šojat

Virtue

The Virtue or the Virtual Interactive Resource-Tasking Universal Environment

Introduction

Origins and Early History

The 1970s brought a new wave of excitement to the field of computer science. The first time-sharing machines and operating systems emerged, enabling interactive, conversational work with computers. This meant that a person could assign a task to the machine, receive a result, input something again, and so on. This was vastly different from working with most previous machines (except for single-user "workstations" or small computers; Unix was just beginning to develop and spread), where one would prepare a program and data on punched cards (or similar), submit them for processing, and then, for example, collect the printed results the next day. The possibility of interactive work was revolutionary at that time.

At the Multimedia Center in Zagreb, dedicated to public education in programming and computer use, under the leadership of Ing. Branimir Makanec, there was a machine with an interactive operating system: the Hewlett Packard HP 2000F Time-sharing Basic[1]. This concept of a "conversation" with the machine naturally inspired the Center's team to think about how wonderful it would be to communicate with the machine in everyday human language. Inspired by this idea, I began working on a program in June 1975 that could self-organize and "learn to converse like a child." Given the incredibly slow processing speed of our single-stage machine (and considering there were about a dozen others using it simultaneously), I achieved a fair degree of success with this project. I named it PAM ("Smart One"). If nothing else, although its working capacity was limited by processing speed to around 15 words, it could differentiate between Croatian and English (after training, unfortunately only on a few sentences in both languages).

1 HP2000 System Specifications: HP2100A main and input/output processor with 32 KiW (approximately 64 KiB) of memory, running at circa 1 MHz, and featuring a 16-bit architecture. The system included two hard disk drives providing a total of 10 MiB storage (2.5 MiB removable and 2.5 MiB fixed). Input/output devices consisted of an 8-bit punch card reader, a magnetic tape reader/writer, and a teleprinter serving as the console, with support for up to 32 terminals - in the MMC configuration, this was utilized with 16 terminals, several of which were connected via telephone modems.

What does this have to do with Virtue?

Over the years, I've worked on a vast number of programming languages and even implemented several on various machines (e.g., LISP, APL, Pascal p-code optimizer, as well as application-oriented languages I devised, such as HIPE, FlowTru, and GraFil.

However, PAM was always on my mind, leading me to develop several algorithms. By 1999, using a learned approach, I achieved a breakthrough: responding vocally (roughly) "I'm fine, thank you, and how are you?" to a vocal input (something like) "Good day, how are you?"[2] on a Sun SPARC 32-bit, 110 MHz, with 32 GiB

And somehow, around that time, I realized that creating a machine that learns and converses isn't the main problem in using computers. Namely, despite its ability to converse, think, and become intelligent through self-organization, I would still have to teach it, for example, mathematics, patience in executing incredibly complex command sequences... And, of course, I couldn't expect it to be precise and good at it. How could I teach it to quickly calculate, for instance, the distances of all points from the center of a four-dimensional field?[3]

with nearly minimal human instructions on what to do, calculates the most complex astrophysical tasks and computes results within a short timeframe. A machine intelligence can't do this because it lacks the "intelligence" of the computer itself, i.e., everything that can be flawlessly executed at the programming level. We just need to find a good way to explain simply to the computer, through a program, what needs to be done. Simply, I say, because the act of programming in most languages requires an unbelievable amount of attention and vast quantities of individual, simple yet algorithmically interconnected instructions (think of any program in any language, whether Pascal, C, Python, Fortran, C++...).

Thus, the solution is to approach this from two sides: from the bottom up, developing a way to easily explain extremely complex problems using a simple, consistent, and incredibly powerful language; and from the top down, developing stimulation for machine intelligence, which can then be easily merged with the power of computational accuracy and strict algorithm execution in the middle.

Concluding that developing such a language can also significantly simplify programming for humans, I embarked on this path.

From APL to Virtue

One of the extremely rare programming languages that can truly express highly complex things in a very simple way is undoubtedly APL, and its successors APL2, J, K, and a few others. These languages process scalars, vectors, matrices, i.e., fields, in a

2 I say "something like" because I don't recall the exact question and response, but these two sentences ("Good day, how are you?" and "I'm fine, thank you, and how are you?") were repeatedly iterated throughout much of my work on PAM.

3 This [four-dimensional field calculation] wasn't the example I was thinking of back then; rather, even simple operations like addition, multiplication, and especially division are extremely challenging to learn through conversation – just recall how easily and well we humans supposedly learned these in school, yet most people still struggle to perform them quickly and accurately from memory (if at all).

mutually compatible manner. In other words, for example, `5 + ⍳ 10` yields a one-dimensional field: `6 7 8 9 10 11 12 13 14 15`. Field members can be both numbers and characters, and APL2 also allows the existence of substructures, e.g., `('prvi broj') 1 (('drugi broj')) (2)`, which creates a vector of four members, where the first member is a vector of nine character members, the second member is the scalar `1`, the third member is a vector with one member being a vector of nine members, and the final, fourth member, is a vector with one scalar member `2`. Admittedly, this might seem complicated at first glance, but such an approach enables exceptionally complex data structures. A significant advantage is the ease of writing programs.

However, despite all its fascinating advantages, the APL family has two major drawbacks: although programs are easy to write, they are very difficult to read, even one's own, let alone others', and APL uses special symbols for all operations, requiring a special font installation. At that time, I couldn't find a sufficiently stable open-source implementation.

My first attempts at bringing programming languages closer to conversational languages were made using APL, defining individual actions as functions with Croatian names. This actually works very well. Writing becomes easier, and readability increases significantly (indeed, very significantly). However, as I mentioned earlier, the instability of existing implementations, despite my efforts to refine and fix them, forced me to abandon those implementations.

As an additional experiment, I created a program for real-time (re-)organization of people's movement through a multi-space urban event (the Shadow Casters' art performance) entirely in Croatian, translated into machine code using a C compiler. How? By defining Croatian expressions as shorter sequences of C symbols, as well as functions with Croatian names. This resulted in a remarkably concise and readable text, truly a fully valid C program (written in Croatian).

During the early 2000s, the idea emerged to put multiple processors into one computer and connect multiple computers into unified processing nodes. Thus, APL, which runs on a single processor, could not satisfy even then-contemporary multi-processor requirements. I searched the Internet for a parallel implementation of APL, as working with large fields inherently calls for multi-processing, but I couldn't find anything like it.

And so I decided to create a kind of assembler, better said pseudo-code, for implementing multi-processor APL (at that time, graphics processors were not even close on the horizon). Thus, the first name for Virtue was ViAPL (Vitler[4] APL), and the lexer translator was called TrAPL (Translate APL, actually APL p-code).

After some time spent developing based on APL, a friend told me that this was actually a good standalone language, and I had started to feel the same way.

And so Virtue was born, named Vrlina (which translates to "Virtue" in English). Through a process of "de-acronymization" one day, I also created what could be called the

4 "Vitler" is a name I coined for my endeavors (as well as the ventures I had) when I was just five or six years old, which I've since retained.

"full name" – Virtual Interactive Resource-Tasking Universal Environment, which truly represents the fundamental idea behind Virtue: to be a universal interactive environment that autonomously assigns tasks to various hardware subsystems. This is an idea I still strive for in its development.

In this emerging Virtue system and language, stimulated machine intelligence – which we can now call inorganic intelligence in today's era – can be integrated at two levels. Firstly, as a direct possibility invoked from within a Virtue program [something like "'Tell me how many kW are in one KS.' ASK.", which would yield the number 0.745699872 at the top of the execution stack, or something like "'List the main components of a human cell.' ASK.", which would return a vector of character sub-vectors with those names, e.g., " ('mitochondria', 'membrane', 'nucleus' ...) "]. Secondly, as a direct possibility for inorganic intelligence to write and execute programs.

Virtue developed, in a way, organically, as my long-standing experience with a multitude of programming languages – ranging from elegant, simple, and practical ones to completely cryptic ones (e.g., W. M. Waite's Stage2, which I used for optimizing the MC6809 assembler translation of Wirth's Pascal compiler P-code) – cultivated a sense for the specific practicalities of individual approaches.

Execution

The Virtue language has been largely implemented, with significant additions that were previously conceived but not implemented, as an interactive system that can also be used for 'batch' processing. However, it has been implemented with, unfortunately, a considerable number of errors, commonly referred to as "bugs". These issues are mostly found in memory allocation and deallocation during multi-processor execution of fields with subfields (substructures), as well as in some other areas.

In reality, most errors arose from the development process, which was simultaneously a series of attempts at shaping the language interpreter, memory allocation, and deallocation [since these operations at the operating system level require significant time, a special internal memory allocator/deallocator was developed – Virtue[5] maintains a separate usage counter for each utilized portion of memory, so for example, a large field stored under several names and used as an object for processing occupies only one memory space. In practice, it has been shown that with this allocation approach, even for complicated calculations, the average total memory required is only about 3 (to 4) times greater than the memory demand of the largest data field being processed].

The Virtue interpreter itself is written entirely in Kernighan & Richie C (K&R C), i.e., the first standardized version of the C language, completely avoiding all newer C (let alone C++) "features". As the language in which UNIX was still being written in the 1970s, K&R C is fully capable and sufficient without any new additions. A significant advantage of writing in K&R C is that all C compilers, from the earliest to the latest, must be able to compile it, since it is the foundation of the language. This enables the Virtue interpreter to

5 In the subsequent text, the name "Virtue" will be used to refer to both the language itself and its machine interpreter. The context will clarify which one is being referred to, except when explicitly distinguished for complete clarity.

be completely independent of the operating system, machine type, version, and C compiler manufacturer. Wherever there is a C compiler, Virtue can be compiled into machine code without errors. The lexical analyzer is written in lex, and for compilation, flex is required (which can also be easily installed everywhere, and moreover, it's possible to compile flex on any other machine since its output is K&R C code).

Virtue exclusively uses libraries that are part of the fundamental POSIX standard, making it not dependent on any modern libraries, ensuring full compatibility with any POSIX environment without any additional installations of external programs.

For instance, the latest (32-bit or 64-bit, depending on the machine) version of Virtue is currently installed on MC68020 SunOS 4.1.1, Sparc Solaris 5.1, ia64 Solaris 10, MIPS SGI Irix, 32-bit FreeBSD, 64-bit FreeBSD, DragonflyBSD, ARM Linux, ia64 Linux, Windows Vista cygwin, and Windows 10 Unix Subsystem. Earlier versions were also tested on MacOS (Darwin), Solaris 7, MIPS NetBSD, and even the Windows Borland compiler. On multi-processor machines (Symmetrical MultiProcessing – SMP), Virtue distributes fields and calculations across all or a user-defined number of processors.

Developing and testing in parallel on numerous different machines, C compilers, and operating systems has often shown that a program working flawlessly on one machine reveals errors on another. Therefore, to fix genuine errors in a program intended to work everywhere, it is indeed necessary to test it under such diverse conditions.[6]

As of April 19, 2026, the implemented Virtue version is v0.8 n0.3 smp0.7 build 70. As mentioned, not all elements of the initially conceived Virtue language specification have been realized yet.

The Foundations of the Virtue Language

Virtue can be described as a semantically very complex yet syntactically extremely simple language rich in synonyms, which also enables the creation of new synonyms and changes to the meaning of (almost) all built-in words. In the not-yet-implemented part of the language, syntactic redefinition of word order (prefix, infix, or postfix notation) is also allowed. Virtue text consists of sentences, with results printed after executing each individual sentence. Sentences end (naturally) with a period " . ". All Virtue verbs are written in uppercase (capital letters), or have special simple symbolic designations (e.g., " + ", " , ", etc.). Nouns include numbers, characters, fields, names... Accusative forms of names are marked with the prefix "@". Accusatives of numbers, fields, functional sentences, etc. are denoted by the word " AT " following them. Quantifiers (quantity designations) are written within "[" "]" after a vector of content, creating an n-dimensional field (n being the number of numerical elements in the quantifier).

6 Here I must note that a multitude of modern software developments boast an elegant form of "self-configuration", which should theoretically account for differences across various systems, but it often turns out that this configuration was never developed or tested on other operating systems for which it's presumed to work. Even common Unix/GNU programs are frequently specifically tailored to a particular Linux version and distribution, and many must be specially adapted for Unix systems, despite both Unix and Linux (all versions of each), as well as, for example, cygwin and mingw, strictly adhering to the POSIX standard in their foundation.

The ability to create accusatives from any language element or result enables exceptional flexibility. Imagine a difficult calculation for which we know the input data. We execute the calculation and store it under the name of the data. In other words, we convert the input data into an accusative and then store the calculation in it (the accusative). A simple example would be to convert the array " (1 2 3 4 5) " using " AT " into an accusative name and then store the array " (5 4 3 2 1) " in it. Subsequently, we can always obtain the calculated result with the verb " GET ". Virtue actually has a special word " RESULT ", which automatically creates a name based on the argument(s) and function itself, checks if that function has previously calculated the result for those arguments, and, if so, immediately returns the result; if not, it executes the function and stores the result under the aforementioned name. In computer science, this process is called "memoization", which Virtue fully automates.

Fundamentally, Virtue is a postfix or postscript language, i.e., RPN ("Reversed Polish Notation"), which is actually the only possible way to execute an operation. Let's take the example "Sum". What? "Five". Okay... and? "Six.". Meaning I remember, I have 5 and I have 6 and then I sum them. or "5 * (2 + 3).". Meaning I have 5, then I have *, then a parenthesis, meaning I need to remember five and * and start a new calculation. I have 2. Okay. Now I have +. And now I have 3. That can be summed, that's 5. Now I can finish the multiplication on the previous level, meaning I have 5 and 5 and I multiply them. The result is 25.

In Virtue, preceding tasks are written as they are executed: object object predicate[7], or subject subject action (with a varying number of subjects depending on the action). The result of each predicate is a new object (Virtue also has adverbs, but that's not part of this discussion). So, the first example: " 5 6 +. ", or " 5 6 ADD[8]. ". Second example: " 5 2 3 + *. ", or " 2 3 + 5 *. ", or " 2 3 ADD 5 MULTIPLY. ", depending on the practicality at that moment of inquiry.

When displaying stored sentences, for instance, Virtue always prints the synonym of the word used by the user.

The use of synonyms in a programming language can significantly contribute to clarity, because, for example, if we want to create a four-dimensional field, the word SPACE much better matches the perceived meaning than the word INTERVAL. We will naturally use the word LEFT if our "right" argument (i.e., the one closer to the command in order) is not of interest, and the word DISCARD when we want to discard the result (Many calculations are unnecessary!). Although the computer action is the same.

The word RIGHT actually corresponds to the syntax " : DISCARD; DIP. ", but more on that later.

New words are created by storing functional phrases with a special OPERATOR designation under the name that will be the new word. These newly formed verbs, as

7 For example, in German, the verb typically comes at the end (of a sentence).

8 Example of synonyms in Virtue: + and ADD are synonyms, * and MULTIPLY are also synonymous, or for instance .., INTERVAL, and SPACE are similarly interchangeable. Many Virtue words have shorter, longer, and symbolic synonyms. For example, the sum of numbers from 1 to 100 can be written as " 100 INTERVAL ADD REDUCE. " or more concisely as " 100.. SUM.".

predicates, can have either any number or a specific number of objects (elements in the stack) to which they refer. In the "language of functions", these functional phrases can have either no arguments, a fixed number, or any number of arguments. In Virtue, it is possible to change the meaning of any already built-in word by creating a new "operator function" and storing it under the name of that built-in word. For example, " DIADIC 1.1 * ::ADD; OPERATOR @ADD SET "[9] will enable that from now on, every (right) number added will be increased by 10%. This, for instance, can allow for fine-tuning of data without modifying the existing program. The semantic prefix " :: " indicates that in this case, the original, built-in meaning of the word must be used, rather than its newly defined meaning.

Programming Styles

With its vast lexical wealth, Virtue enables programming in multiple styles, ranging from something akin to the lowest, assembler-like level, with explicit jumps and checks, to a style similar to APL (with significant differences in grammar and script), LISP (which is precisely the opposite of Virtue, being a prefix language with numerous parentheses), linear or structured style, conditional execution of commands in the style of Pilot (a computer-aided instruction programming language), to high-level (functional) programming using appropriately chosen word-names.

Considering its original postfix, i.e., Reverse Polish Notation syntax, and the consequent use of the stack, Virtue is similar to Forth. The language specification also envisions an approach from my language Flowtru [filtering data that leaves only the data satisfying certain (functional) conditions].

Virtue is a fully dynamic language, meaning all functions (sentences that produce a result based on some number of data) can be dynamically changed, and it's possible to write programs that create (write) functions, which can then be executed. In other words, Virtue enables the creation of self-modifying/self-describing programs.

In terms of meeting the requirements for being a Turing machine (i.e., a programmable device capable of executing everything that can be computed through an algorithmic process), Virtue is comprehensive within multiple distinct programming approaches, without relying on others (e.g., using "assembler-level" expressiveness versus "APL-level" expressiveness in Virtue).

As previously mentioned, Virtue sentences end with a period " . ". During interactive work, a sentence can be ended at any point (even an empty one, i.e., just " . "), except within functional and data phrases (which must remain syntactic wholes within any sentence). After each completed sentence, Virtue prints the processing result. This enables using Virtue as a powerful calculator, and in this sense, its operation is very similar to working with HP RPN calculators. Additionally, this allows for step-by-step testing of an intended future program.

---

9 Create a function with two arguments that multiplies the right argument (top of the stack) by 1.1, then adds the left argument to the newly formed right argument, and convert this functional phrase into an active verb and store (SET) it under the name ADD. From now on, the word ADD will have a new meaning: x + y * 1.1.

The Stack

Virtue is a postfix (postscript) language, thus RPN (object ... predicate), and all operations are performed on elements that are on the stack. In computing terms, the stack is analogous to a stack of hay (hence the name): what's first put on the stack remains at the bottom, and the last thing added is at the top. Therefore, the stack is Last In First Out (LIFO). The depth of the stack in Virtue is 1024 elements, which is more than sufficient for extremely complex operations.

We can visually imagine the Virtue stack being filled from left to right. The first thing goes on the left, at the "bottom" of the stack, the second is to the right of the first, and so on.

When entering a Virtue sentence, all elements of that sentence are first converted into internal forms, and links to these forms are placed in the execution program (i.e., each entered sentence is a separate program at the execution level). After the user has entered a sentence, Virtue goes through the translated execution program step by step, distributing the input onto objects and predicates. Objects, as they appear in order, are stacked one on top of the other (so the last object in the sentence is at the top of the stack). Predicates, or verbs, are executable commands that can have any number of objects (from none onwards) based on their syntax.

For example: `(1 2 3 4)` `{one object}` `2` `{second object}` `DIVIDE` `{predicate}` `,` `{end of sentence}` puts the first object, a one-dimensional space (vector) with four members, at the top of the stack, then adds the second object, scalar 2, on top of the first in the stack, executes the predicate, verb DIVIDE, between the "left" `[(1 2 3 4)]` and "right" `[2]` arguments, i.e., divides "left" by "right", and prints the output `0.5 1 1.5 2` – a one-dimensional field with four members now at the top of the stack. The verb DIVIDE removed its two objects from the stack and left the result in their place. Meaning the stack was empty. We added one object, added another – now the stack has two members. We executed the division, and the result remains on the stack – now the stack has one member. If we want to work further with the result, it awaits us at the top of the stack, expecting the next guiding sentence.

Any Virtue data can be on the stack (except, of course, for verbs that are not part of a functional phrase or have had their meaning altered).

LEFT, DISCARD, RIGHT & SWAP

The fundamental commands for manipulating the stack itself have already been mentioned: LEFT and RIGHT (I've never actually used the latter, but it could certainly be practical in some cases) and DISCARD, which is a synonym for LEFT. The entire stack can be cleared with the command CLEAR. Additionally, saying SWAP will cause the top two objects on the stack to exchange places, switching their positions.

DUP = DUPLICATE

The DUPLICATE command duplicates the item at the top of the stack. This is often extremely important to perform an operation on a piece of data (e.g., to determine its

dimensions) while keeping the original data intact. Incidentally, a simple " DUP MULTIPLY " effectively squares the data.

DIP

Sometimes it's crucial in a program to manipulate objects that are located below the topmost one on the stack. This is made possible by the phrase " : <desired action> ; DIP ", which leaves the topmost element of the stack in its original position and performs the <desired action> on the stack beneath it. The phrase " : <...> ; " can be written out in full as " NEADIC FUNCTION <...> ENDFUNCTION ", providing a syntactic framework for a sequence of Virtue words, which we'll call a function, that can take any number of objects as its arguments. The DEEP command, which would perform the same action as DIP but to any arbitrary depth in the stack, has not yet been implemented.

COLLECT, MARK & DISPERSE

The commands that enable collecting objects from the stack and creating a one-dimensional field from them are: COLLECT: gathers all objects from the entire stack (from left to right!) into a single field with a number of elements equal to the number of objects that were on the stack. Alternatively, if a position on the stack was marked with the word MARK during its creation, COLLECT will gather all data from that MARK to the top of the stack. DISPERSE (unfortunately, not yet implemented): "disperses" the elements of a one-dimensional data field across the stack, with the first element going to the top of the stack. The dispersal order is such that each subsequent element occupies the next available position on the stack.

COUNTSTACK, COUNTTOMARK, CLEAR & CLEARTOMARK

In addition to previous commands, it's possible to determine the number of data elements on the entire stack using COUNTSTACK (whole stack) and from the topmost (rightmost) MARK label with COUNTTOMARK. Apart from CLEAR, which clears the entire stack, there's also CLEARTOMARK, which clears the stack up to the topmost (rightmost) MARK label.

When a new function is called, it receives a protected stack on top of the existing one, only able to manipulate arguments present on the calling stack as specified in its definition, categorizing functions into niladic (no arguments), monadic (one argument), diadic (two arguments), triadic (three arguments), or those with multiple arguments starting with "ARGS <number of arguments> ...". The NEADIC label allows a function to operate on any number of data elements on the stack, as exemplified by "NEADIC COUNTTOMARK CLEARTOMARK; OPERATOR @cleanup SET.", which creates a new word "cleanup" that clears all elements from the topmost MARK label and leaves the count of cleared elements on the stack.

Data

The primary feature of Virtue is that every "entity" (i.e., everything that exists in Virtue) "knows" what it is, and Virtue knows which operations can be performed on which entities and how they are executed for each. For example, when multiplying, a character (e.g., 'a') cannot be multiplied by a number, but it can be multiplied by true (T) or the number 1, leaving ('a'), or false (F) or the number 0, causing it to be deleted or replaced with an empty space, a blank (' ').

At the highest level of differentiation, things in Virtue are divided into ADDRESS (the location where the meaning/content of a name is found), VARIABLE (a name that invokes its content), OPERATOR (verbs), FUNCTION (a sequence of Virtue words in a functional phrase), LABEL (a program marker used as a point to "jump" to, i.e., transfer execution flow), SCALAR (denoting a scalar), and ARRAY (a data field).

Addresses, variable contents, functions, scalars, and arrays can be placed on the stack. Any of these objects can be included in a field.

Fields

Addresses and functions behave on the stack as singular elements (effectively non-dimensional). Incorporating addresses and functions into fields has not yet been implemented, but it will enable the execution of multiple instructions on a single piece of data (Multiple Instruction Single Data - MISD) and multiple instructions on multiple data (Multiple Instruction Multiple Data - MIMD[10]), provided that both fields have the same dimensions. In Virtue, these operations will be performed using the diadic `EACH` command (which, unfortunately, is also not yet implemented).

Scalars are individual data elements with no dimensional markings. Depending on the semantics of a particular verb (i.e., operation), scalars can be directly included in operations with any field, including structured fields (i.e., fields with subfields). For example, " 2 (1 2 3 4) * " yields 2 4 6 8.[11]

Fields are the primary "workspace" for working in Virtue. Each field has labels for its dimensional rank (RANK), i.e., the number of dimensions, and its shape (SHAPE). Additionally, it has a label indicating whether it's simple or structured, which also includes a label for the depth of the deepest substructure (DEPTH), as well as whether all data within it are of the same type or not (this significantly accelerates execution, as it doesn't require checking the data types[12] every time).

An unstructured field can be entered in the form " (1 2 3 4 5 6) [3 2] . ", which will yield a field with three rows and two columns. Virtue's dimensional arrangement, unlike C but similar to Fortran, follows the "column major order", where the first dimension corresponds to columns and the second to rows. Visually (when printing or creating an

---

10 Concurrent execution of multiple, distinct programs on the same data (i.e., MISD - Multiple Instruction Single Data) and multiple, distinct programs on a corresponding amount of data (i.e., MIMD - Multiple Instruction Multiple Data) are extremely rare but also extremely powerful approaches.

11 When printing unstructured fields, for clarity, Virtue omits the surrounding brackets that are required when the user inputs them.

12 Data types include, for example, character, logical, polar, complex, stochastic (random), and others.

image), the "x" coordinate goes from left to right, the "y" coordinate from top to bottom, and higher-dimensional coordinates are represented as x-y planes, x-y-z planes, etc., depending on the dimensionality. This arrangement of "columns" and "rows" is also directly compatible with the use of multidimensional numbers, where the real part represents the "x" coordinate (columns), and the imaginary parts represent the "y" coordinate (rows), "z" (planes), etc. It's worth noting that in Virtue, the "y" coordinate, as mentioned, goes downwards, whereas in the geometric representation of complex numbers, the "y" coordinate, i.e., the axis of imaginary numbers, is oriented upwards.

RESHAPE

The aforementioned field can be created from the data (1 2 3 4 5 6) using the RESHAPE verb, whose right argument is a one-dimensional field (vector) specifying the dimensions: " (1 2 3 4 5 6) (3 2) RESHAPE . ". If the left argument has too many members, it will be truncated; if it has too few, it will be replicated from the end back to the beginning until all elements of the field with the specified dimensions are filled.

ENLIST

Viewed linearly, i.e., by converting a multidimensional field into a one-dimensional vector (e.g., using the `ENLIST` verb), the element from the second column of the first row will immediately follow, in sequence, the element from the last column of the first row, which is also evident from the aforementioned field creation process. Viewed linearly, i.e., by converting a multidimensional field into a one-dimensional vector (e.g., using the `ENLIST` verb), the element from the second column of the first row will immediately follow, in sequence, the element from the last column of the first row, which is also evident from the aforementioned field creation process.

INTERVAL = SPACE = ..

A very important verb in Virtue is INTERVAL (also known as SPACE, or symbolically " .. "), which takes a single one-dimensional or multidimensional integer object (argument) and creates an index field of the corresponding size. An index field means that each position in this field holds a value corresponding to the element's position within the entire field. In other words, the sentence " 3 INTERVAL . " will yield the vector 1 2 3, while " 3i2 SPACE[13] . " will produce the field (1i1 2i1 3i1 1i2 2i2 3i2 1i3 2i3 3i3) with dimensions [3 2]. Creating an index field at the beginning of many algorithms enables the calculation of local values within the field based on their spatial position. For example, the sentence " (-10i-10 10i10) SPACE MAGNITUDE 7 NOTGREATER . " will create a circle of true values with a diameter of 14 within a square of false values sized 21x21.

To obtain a structured, single-element, one-dimensional field, the `ENCLOSE` verb is used. When entering a constant from the user, inner brackets are employed, for example: " `( (1) (2 (3 4)) (5 (6 (7))) .` " will yield a three-member vector (one-dimensional field) consisting of a single-element vector in the first position, a two-member vector comprising

13 U ovome slučaju sinonim SPACE bolje odgovara namjeri nego li INTERVAL, iako su, naravno, kao sinonimi uvijek zamijenjivi, dajući isti rezultat i ako bismo napisali " `3i2 ..` ".

one number and a two-member vector in the second position, and a two-member vector consisting of one number and a two-member vector that itself contains one number and a single-element vector.[14]. ...

... Unlike a straightforward input, the ENCLOSE verb operates on only one field, so " (1 2 3 4 5) ENCLOSE . " creates a single-element vector consisting of the vector (1 2 3 4 5). This newly created vector, now the sole member of its "supervector", can be "extracted" using the DISCLOSE verb.

The aforementioned field can be included in scalar operations (i.e., operations applied individually to each element) in several compatible ways, as long as one argument is a scalar or both fields have the same dimensions. For example: " `( (1) (2 (3 4)) (5 (6 (7)))) 5 * .` "[15]," `( (1) (2 (3 4)) (5 (6 (7)))) (2 3 4) * .` "[16]," `( (1) (2 (3 4)) (5 (6 (7)))) ((2) (3 4) (2 (3 (4)))) * .` "[17], etc.

Two vectors can be combined into one using the CATENATE verb or its symbolic equivalent, a comma (" , "). For example: " (1 2 3) (4 5 6) , . " yields a six-element vector 1 2 3 4 5 6.

Individual elements of a field can be separated using "LISP-like" verbs: FIRST (CAR), returning the first element REST (CDR), returning all elements except the first TAKE and DROP, along with their derivatives. Fields can also be shortened or lengthened with the COMPRESS and EXPAND verbs, respectively, using an argument to specify where the operation should be applied.

Characters

Characters and character strings in Virtue are entered within apostrophes: " 'Dobar dan.' ". This will place a vector of ten character elements at the top of the stack.

In computer science, the "first program" that prints the words "Hello world!" is well-known. In Virtue, this task appears simple as follows: " 'Hello world!' . ", to which Virtue will respond by printing Hello world!.

Characters and character fields are processed scalarly in the same manner as all other data in Virtue, as described above, and the verbs applicable to them can be applied, such as, for example, the aforementioned multiplication with T and F (or 1 and 0), comparison, and similar operations.

Numbers

Virtue works with multidimensional numbers up to a maximum of five dimensions. Numbers can thus be real (1), complex (2), quaternions (4), or octonions (8). When developing index fields, quaternions can be reduced to 3 dimensions, and octonions to 5, 6, or 7. In mathematical operations, it's essential to consider quaternion and octonion

14 The Virtue expression `((1) (2 (3)) (5 (6 (7))))` nonetheless conveys more than 51 words of explanation can, serving as a picture that speaks volumes.

15 Gives `( 5 ) ( 10 ( 15 20 ) ) ( 25 ( 30 ( 35 ) ) ) .`

16 Gives `( 2 ) ( 6 ( 9 12 ) ) ( 20 ( 24 ( 28 ) ) ) .`

17 Gives `( 2 ) ( 6 ( 12 16 ) ) ( 10 ( 18 ( 28 ) ) ) .`

geometry, especially during rotations and similar manipulations. Numbers are written as: 1, 1i1, 1i1j1, 1i1j1k1, 1i1j1k1l1, 1i1j1k1l1m1, 1i1j1k1l1m1n1, 1i1j1k1l1m1n1o1.

All multidimensional numbers (complex, quaternions, and octonions) can be displayed in both Cartesian and polar systems, with polar numbers also available in degrees or radians (fractions of a full circle from 0 to 2π). In polar numbers, the "real" part (i.e., the first basic number) represents the distance from the origin, while subsequent coordinates represent angles in each dimension. For example, d1p45 is equivalent to 1p0.7853981634 in polar and 0.7071067812i0.7071067812 in Cartesian coordinates. Virtue stores the subtype for each number. Polar numbers in radians are written as 1p1, 1p1q1, 1p1q1r1, ..., with a 'd' prefix (e.g., d1p1, d1p1q1 ...) indicating that all "imaginary" parts are expressed in degrees.

CARTESIAN, POLAR, DEG, RAD & NUMBER

Cartesian coordinates are converted to polar coordinates using the POLAR command, while polar coordinates are converted to Cartesian coordinates Cartesian coordinates are converted to polar coordinates using the POLAR command, while polar coordinates are converted to Cartesian coordinates with CARTESIAN. Polar coordinates can be converted to degrees with DEG or to radians with RAD. The NUMBER command removes the polarity or Cartesian marker from a data point. For example, a number like 1i1.308996939, normally treated as Cartesian, can be directly transformed as if it were polar d1p75[18] using the sentence " 1i1.308996939 NUMBER POLAR DEG . ". In contrast to this direct "casting" of data into a new form without value transformation, " 1i1.308996939 POLAR DEG . " yields a polar number equivalent to the Cartesian input, which is d1.64726227p52.62220839[19].

Virtue has integrated mathematical operations that automatically adapt to combinations of number dimensionalities and characteristics. This enables, among other things, significant acceleration of mathematical operations on 3-dimensional quaternions or 5-, 6-, or 7-dimensional octonions, as the complexity of many operations is substantially reduced. For instance, multiplying two octonions requires 64 simple multiplications and 56 additions, while multiplying two 7-dimensional numbers demands 49 multiplications and 42 additions, and multiplying a full octonion with a complex number only requires 16 multiplications and 8 additions. By tracking the characteristics of individual numbers, Virtue allows for seamless use of any combination of Cartesian and polar numbers. For example, 0.7071067812i0.7071067812 multiplied by d1p45 yields 0i1, or d1p90, effectively performing a 45-degree rotation in the direction opposite to the clock hand. When dealing with more than two-dimensional rotations, it's always wise to think in four- or eight-dimensional rotational space.

Multidimensional Fuzzy Logical Values

As logical values, all numeric values greater than 1 are considered true (T), and all less than 0 are false (F). All values between 0 and 1 are non-crisp (fuzzy) logical values; for

18 d1p75 is in radijans 1p1.308996939.
19 In radijans 1.64726227p0.918430796.

instance, 0.7 is fairly true, and 0.1 is almost certainly incorrect. Then, for 0.5, it's known that its truth cannot be definitively determined as either true or false.

AND, OR, STRONGAND, STRONGOR & NOT

All logical operations in Virtue follow the rules of Łukasiewicz-Tarski logic, with infinitely (albeit computationally precision-limited) many intermediate values. Virtue features both normal logical operations (e.g. AND[20] or OR[21]) and strong logical operations (e.g. STRONGAND[22] or STRONGOR[23]). The definition of the monadic NOT (negation) is 1 - x, etc.

The use of multidimensional numbers (complex, quaternions, and octonions) also enables the application of multidimensional logic, i.e., multidimensional logical values and operations. Each numerical dimension represents a separate logical dimension. For instance, the logical data point 0.1i0.8j1 could represent that, across three different parameters in some statistical analysis, the influence of the first parameter was 10%, the second 80%, and the third 100%. If we obtained results like 0.2i0.7j0.9 from another corresponding analysis, we can use logical operations to derive various comparison aspects.

Binary Numbers

Binary numbers are denoted by the prefix `0#`, e.g., `0#1001` represents the binary number 9.

Constants

Virtue has a large number of built-in constants, such as #PI, #2PI, #SQRT2, #E, #LN2, etc.

Names and Contents - Variables

Every programming language must have the ability to store intermediate results for later use. The common term for these "storage units" is variables, i.e., names that hold something. In Virtue, all data types that can be on the stack (addresses, functions, scalars, and fields) can be stored under a name. Names can be derived from any of these "stackable" data types. In other words, a multidimensional field can serve as a name for a scalar, a function can be a name for an address, or a scalar (number) can be a name for a function. Although Virtue has a large set of built-in words and symbols (e.g., +, -, ==, etc.), all of these can be used as names. If converted to accusative form by prefixing with the symbol @ during user input, they are placed on the stack as storage addresses for those names. For example, the accusative @ADD in a program is stored on the stack as the address of the name ADD. The verb ADD in a program executes as a command. Note:

---

20 min (L, R)
21 max (L, R)
22 max (0.0, L + R - 1.0)
23 min (1.0, L+ R)

Verbs (i.e., commands) cannot stand alone on the stack but only as part of a functional phrase or function, especially those marked as OPERATOR.

AT

A non-operator function is a data item placed on the stack; an operator function is executed immediately. Data items that can be on the stack are converted to addresses using the AT verb.

ASSIGN & SET

To store some content in an address variable expressed in this way, the verbs ASSIGN and SET are used. The left argument (below the top of the stack) is the content, i.e., the data item to be stored, while the right argument (at the top of the stack) is the address, i.e., the name under which the data item is to be stored. ASSIGN performs the storage and removes the address from the top of the stack but leaves the data item, which can then be further manipulated. SET performs the storage and removes both the address and the data item from the stack.

Example: " 3 @a ASSIGN @b SET. " will store the scalar number 3 under the name "a" and also under the name "b".

When names under which something is stored (i.e., variables with content) appear in a sentence (program) as standalone names, not as addresses, they place their content on the stack. Thus, after the previous example, if we enter " a. " (or " b. "), we will get the scalar number 3 at the top of the stack.

GET

The GET verb takes an address as its argument and replaces it at the top of the stack with its content. For example, we could retrieve the number 3 from the previous example using the address like this: " @a GET. ".

Here's an example of storing and retrieving data using two-element vectors:

```
"Fixed Data Entry:"
'Ljerka' (1 1) AT SET
'Antun' (1 2) AT SET
'Marija' (2 1) AT SET
'Zvonko' (2 2) AT SET.
```

(Note: Single quotes in Virtue denote comments.) Then, a "user-facing" program could look like this:

```
'Upis rednoga broja doma i rednoga broja stanovnika u obliku (x
y): ' TELL
INPUT AT GET PRINT.
```

If we enter, for example, " (1 1). ", we would get the name 'Ljerka'.

INPUT & TEXTINPUT

Important note: In Virtue, INPUT allows the user to perform full linguistic control over the Virtue interpreter before sending what they want to input, meaning any Virtue sentence can be entered. Therefore, it's essential that the user's input when using the INPUT verb ends like any other Virtue sentence, i.e., with a period ("."). To prevent the user from using Virtue during data entry with INPUT, we can use TEXTINPUT, which will accept the input as a character string.

EXECUTE

If we now "execute" this string using the EXECUTE verb or its symbol " ! ", it will be converted into a program and executed. In other words, we'll get from a character string with some data to an actual piece of data on the stack:

```
TEXTINPUT ! AT GET PRINT.
```

In this case, the user will enter " (1 2) " (without a period), press "Enter", and receive 'Antun' as the result.

Memoisation

RESULT

As previously mentioned, Virtue enables automatic memoization, i.e., using stored results of already executed functions with specific arguments. Manual memoization can be achieved using the previously explained AT and GET, but Virtue has the RESULT verb for automated memoization.

RESULT converts the function's arguments and the function itself into a unique name, checks if a result is already stored under that name, and if so, directly places the pre-computed result from memory onto the stack. If not, it calls the function with the given arguments, stores the result under the generated unique name, and puts it on the stack.

By using RESULT, we can significantly accelerate certain calculations by retaining intermediate results of complex functional manipulations. Although manual memoization is possible in any programming language, it's often too complicated to be worthwhile in terms of human programming effort, making it rarely used. In Virtue, as mentioned, this can be done manually with AT and GET, but the automation of memoization via RESULT greatly contributes to the simplicity and usability of this optimization method.

As I've explained earlier, Virtue stores "knowledge" about each individual data point, including what it represents. During automated memoization, it's crucial not to reuse existing results for combinations of data and functions if the input data was generated randomly (e.g., in programs using stochastic methods). Therefore, Virtue internally records and tracks a marker with each result indicating whether it originated from a random operation (e.g., RANDOM, DEAL, PROBABLY... see later in the text).

Consequently, RESULT will not be activated if any data point involved in the current function argument processing was the result of a random action.

Flow control

Jumping

JUMP

To enable jumping within a Virtue sentence, i.e., explicit change of execution flow, Virtue uses labels within the sequence of a sentence, marked with a prefix %. The verb that invokes a jump to a specified labeled location is JUMP.

For example, an infinite loop in Virtue would be written as:

```
'Endless poem' PRINT %ponovi 'I will never stop writing!' PRINT
@%ponovi JUMP. [24]
```

Meaning: Place a vector of twelve character elements ('Endless poem') on the stack. Print this vector and remove it from the stack (Virtue has several print commands; PRINT prints and removes from the stack, OUTPUT prints and leaves on the stack, and there are others like TELL, REVEAL, etc.). Label a point in the sentence with the name %ponovi. Place the vector 'I will never stop writing!' on the stack. Print it and remove from the stack. Place the address (sentence location) of the label named %ponovi on the stack. Remove this address from the stack and resume execution at the location marked by the name %ponovi.

Funkcije

Although we've already mentioned functions, they are one of the most crucial building blocks in Virtue. Unlike most other programming languages, Virtue functions (program parts enclosed by functional phrases) can have any number of arguments (none, many, or an arbitrary amount) and can create multiple data as solutions, leaving as many outputs on the stack top as needed.

NEADIC = :, NILADIC, MONADIC, DIADIC, TRIADIC & ARGS x FUNCTION

Functions can be classified based on their argument count as NEADIC (having no fixed number of arguments, with the entire stack from top to bottom being accessible), and functions that create a new stack on top of the existing one, which, apart from the number of arguments they receive in their part of the stack, cannot affect other stack elements. All functions, except for neadic ones, remove arguments from the stack and leave only the result(s) of the functional execution on the stack. Since non-adiadic functions are frequently used, the abbreviation " : " can be used instead.

---

24 In the current implementation of Virtue, unfortunately, Unicode is not supported; it only accepts 8-bit characters. Therefore, the text here is in English, as using Croatian characters (e.g., Ć, Č, Š, Đ) is not possible.

NILADIC has no arguments, MONADIC has one argument, DIADIC has two arguments, TRIADIC has three arguments. For more than three arguments, a function can start with ARGS <number of arguments> FUNCTION.

ENDFUNCTION = ;

Functions end with the word ENDFUNCTION, but using the " ; " symbol is more common and stylistically elegant.

Example of a function converting nautical miles to kilometers:

```
MONADIC 1.852 *;.
```

Now it's on the stack top. If we store this function: " @nm_km SET. ", we can recall it by name (nm_km) anytime. To execute it from the stack top, we must say EXECUTE (or "!"). The ability to place functions on the stack is crucial in Virtue, as it enables handling, writing, and modifying parts of a Virtue program within the program itself; thus, Virtue can handle both values and executable functions.

If we want this function to execute immediately when called by name (like all built-in words), i.e., as a command, before storing it, we must assign it an operator label with the word OPERATOR:

```
MONADIC 1.852 *; OPERATOR @nm_km SET.
```

What is two and a half nautical miles in kilometers? `" 2.5 nm_km. "`

Looping or not?

Since Virtue primarily operates on data fields, classical loops are rarely necessary. We've already mentioned the example of a circle of true values within a square of false values. The following sentence will generate a hyper-sphere from one[25] to eight dimensions, displaying it as an array of asterisks ("*") on a blank field. It expects three arguments on the stack – the sphere's radius (a one-dimensional number), the sphere's center coordinate (a multidimensional number), and the field size (as a dimensional number):

```
INTERVAL SWAP SUBTRACT MAGNITUDE NOTGREATER '*' MULTIPLY.
```

If we enter this sentence after placing, for example, the arguments 8, 11i11j11k11, and 21i21j21k21 on the stack, we will obtain a four-dimensional hyper-sphere of asterisks. First, INTERVAL will create a four-dimensional index field from 1i1j1k1 to 21i21j21k21. Then, we'll subtract the center, i.e., 11i11j11k11, from this field. Now, our index field will range from -10i-10j-10k-10 to 10i10j10k10, with 0i0j0k0 (i.e., 0) at the center. MAGNITUDE will calculate the distances of individual points from the center using Pythagoras' theorem in four-dimensional geometry.[26]. NOTGREATER will compare the desired radius (8, left

25 A one-dimensional sphere is a line segment, a two-dimensional sphere is commonly called a circle, a three-dimensional sphere is simply a sphere, and those with more than three dimensions are usually referred to as hyperspheres.

26 sqrt (r*r + i*i + j*j + k*k).

argument) with the distances from the center at the top of the stack (right argument). The result will be a four-dimensional field of true values where points were within the desired radius and false values where we intended to be outside the sphere. Then, by multiplying the asterisk ("*") with this field, we'll get a space (" ") where it's false, i.e., outside the sphere, and an asterisk ("*") where it's true that the distance is less than 8, meaning inside the sphere. The automatic output after entering the sentence will show us how the four-dimensional sphere looks like in two-dimensional "slices", i.e., cross-sections.

LOOP

Of course, sometimes it's necessary to execute a part of a program a certain number of times (say we want to create a series of spheres with different radii from the previous example). Here's a simpler example:

```
10
MONADIC
     'Ovo je broj ' SWAP CATENATE PRINT;
LOOP.
```

And we'll get a list of numbers from 1 to 10. CATENATE combined the text 'This is number ' with the loop iteration number (which we placed on top of the stack, i.e., to the right of the text using SWAP, resulting in a complex vector of 13 elements, where the first 12 are characters and the 13th is the number).

argumenta

EACH

EACH repeats the execution of a function for each individual element of its argument[27]. In other words, the sentence " (1 6 2 7 9 3) NEGATIVE. " yields the same result as " (1 6 2 7 9 3) MONADIC NEGATIVE; EACH. ", but the latter is slower in execution.

So, what's the purpose of EACH? Imagine you want to create a multiplication table for numbers from 1 to 10:

```
10 INTERVAL MONADIC 10 INTERVAL MULTIPLY; EACH.
```

Here's what we've done: Created an interval (a vector with a list of numbers from one to ten); Applied a function to each individual number in that list; Within the function, we created another interval from 1 to 10; Then multiplied it by the respective number from the first list. What did we get? We obtained a one-dimensional field (vector) with 10 elements, where each element is itself a vector of 10 elements. The first element contains multiplication results from 1 to 10 with 1, the second element contains multiplication results with 2, and so on - until the tenth element, which is a vector of results multiplied by 10. In essence, we created a multiplication table, where each base vector element corresponds to a row of results.

27 Diadic EACH is, unfortunately, not yet implemented, only the monadic one is.

To get a two-dimensional table (a field of size 10x10), as multiplication tables are commonly represented:

```
ENLIST (10 10) RESHAPE.
```

Or, if you wanted to create a general-purpose function for any array of numbers, you could write it like this (we use the name .numbers for storing the input field within the function; the leading dot in .numbers indicates that this variable is local to this function execution, as each function call in Virtue creates a new small symbol table[28]):

```
MONADIC
     @.numbers ASSIGN "Save the argument field"
     MONADIC
          .numbers MULTIPLY;  "Multiply each member of the field"
     EACH                     "with the whole field"
     DUP ENLIST       "The result field has twice more dimensions"
     SWAP SHAPE DUP CATENATE RESHAPE;
OPERATOR @tablica_mnozxenja SET.
```

This function, or operator, which we can genuinely call a verb, calculates the so-called outer product of any input field with any dimension. Therefore, it's perfectly fine to also name it OUTERMULT (outer multiplication) and use it in any case where we need an outer product:

```
@tablica_mnozxenja GET @OUTERMULT SET.
```

However, our OUTERMULT isn't the actual vector product, which should be diadic. So, let's adjust the above function to take two arguments and properly adjust the output field dimensions.

28 Virtue recognizes four types of storage/access through hierarchical lookup for an individual name. Unprefixed name: Stored in the base symbol table and only searched within this table upon invocation. "$" prefix: Searches all symbol tables from top to bottom; if not found, it's stored in the topmost (currently active function's) symbol table. "_" prefix: Indicates searching for a name from top to bottom along the stack of symbol tables; if not found, it's stored in the base symbol table. "." prefix: Specifies looking up a name only within the topmost (active function's) symbol table, where it's also stored. These prefixes are not part of the name itself but rather modifiers for the lookup and storage method. They enable fine-grained handling of functional depths related to individual names. For instance, "neko_ime" has two different contents in the following example: @neko_ime SET MONADIC 2 @.neko_ime SET ;.

```
DIADIC
     @.numbers SET
     MONADIC
          .numbers MULTIPLY;
     EACH
     DUP ENLIST
     SWAP SHAPE .numbers SHAPE
          SWAP "make column first"
     CATENATE RESHAPE;
OPERATOR @OUTERPROD SET.
```

To obtain a general outer product of any function, we can convert the above function to TRIADIC, calling it with two vectors and one function on the stack. First, store that function (" @.function SET "), and replace the word MULTIPLY with " .function EXECUTE " or " .function! ". Now, this function creating an outer product of two vectors can be as complex as needed.

MASK

Some problems require that the result at a certain point in a field be the output of operations on its neighboring fields. Such requirements are common in, for example, meteorology, where the influence of surrounding points is crucial at each point, not the entire field. In the history of computer science, John Conway's Game of Life is well-known, where "life and multiplication" evolve on a board with certain "cells," depending on their environment.

The basic algorithm involves examining the eight neighboring cells; if a cell is alone or only two are present (itself and one other), it "dies." If there are three or four in total, it "survives," but if there are more than five, it again "dies" (due to overpopulation). A new cell is "born" in an empty space if its environment contains exactly three "live" cells.

The `MASK` command takes three arguments: Input field (deepest on the stack): The field on which you want to operate. Masking field (second deepest): A field with odd dimensions (e.g., 3x3, 5x5) containing numerical values that will be multiplied with corresponding parts of the input field and then passed for processing by the third argument, a function. Function (topmost): To process the masked data.

`MASK` starts from the first element of the input field, applies the "mask" through multiplication on the neighborhood, executes the function, places the result in the corresponding position in the output, and repeats this process with the next element of the input field. It's as if you cut a square hole in a piece of paper, then examined a table through it, processed the data seen, wrote down the result, and moved your "window" one step further.

Here's an example using `MASK` in a program that executes the next generation in the aforementioned Game of Life:

```
MONADIC
   (1 1 1 1 0.5 1 1 1 1) [3 3]
      MONADIC
         RAVEL SUM
         (2.5 3.0 3.5) IDENTICAL
         ANY
      ;
   MASK
; OPERATOR
@lifegen SET.
```

This might be the shortest program that generates the next generation in this Game of Life.

Conditionals

IF

<true/false> <objekt> IF

The IF command leaves <object> at the top of the stack if the first (left) argument is true (i.e., T, or >= 1), and leaves #NIL (an empty scalar, explicitly nothing) if the left argument is not true (i.e., F, or <= 0). If we use an address to a program location (e.g., @ %ponovi) as <object>, followed by the JUMP command, the program will continue executing at the %ponovi location if the condition is met (i.e., the left argument of IF was true). Otherwise, IF yields #NIL, meaning JUMP doesn't need to do anything, and execution continues sequentially.

Example: "Guessing the number 2:

```
'Enter a number: ' TELL
INPUT
2 NOTIDENTICAL @%wrong IF
'Congratulations, you guessed the number!' PRINT
%wrong 'Thank you for participating.' PRINT.
```

If any number other than 2 is entered, only "Thank you for participating" will be printed. If the number is 2, both "Congratulations, you guessed the number!" and "Thank you for participating" will be printed. TELL differs from PRINT in that it doesn't move to a new line after printing, but like PRINT, it removes the top of the stack after printing. For example, 'Good' TELL 'day' TELL will print "Goodday".

A more Complex Example: Calculating the n-th Fibonacci Number. We use double recursive calls (recursion). The n-th Fibonacci number is recursively defined as: fib(1) = 1, otherwise, fib(n) = fib(n-2) + fib(n-1).

```
MONADIC
   MONADIC
      DUP 2 < @%a IF JUMP
      DUP 1 - fib
      SWAP 2 - fib
      + RETURN
%a    DISCARD
      1;
   RESULT; OPERATOR
@fib SET.
```

Here, we utilized memoization with the RESULT command, significantly shortening each calculation of a Fibonacci number after the first one.

CHECK & IF_YES/IF_NO

<true/false> CHECK .. <object/predicate> {IF_YES / IF_NO}

Let's revisit the "Guessing the number 2" example using CHECK and IF_YES/IF_NO:

```
'Enter a number: ' TELL
INPUT
2 IDENTICAL CHECK
'Congratulations, you guessed the number!'  PRINT IF_YES
DISCARD IF_NO
'Thank you for participating.' PRINT.
```

In this sentence, we checked if the input number was 2 and placed 'Congratulations, you guessed the number!' on the stack, which would then be printed and removed from the stack if the condition was met (i.e., the number was 2). However, if the condition wasn't met (the user entered a different number, making the check result false), PRINT would be skipped, leaving 'Congratulations, you guessed the number!' on the stack. To avoid this, we add DISCARD IF_NO, which removes the string from the stack if PRINT isn't called.

IF_YES abbreviated as ?Y, and IF_NO as ?N.

CHECK/IF_{YES,NO} in Virtue (and Pilot, a teaching language) enable checking a condition once (CHECK) and then using IF_YES and IF_NO to execute or skip specific parts of the program based on that check. This works within a single sentence (since CHECK only applies within a sentence).

Important Note: Virtue's logic is based on **fuzziness**, meaning logical data represent "probabilities" of truth. Therefore, CHECK also considers this fuzziness. If the argument for CHECK is 1 or T (true), or 0 or F (false), or anything below 0 or above 1 there's no ambiguity; if true, it will be YES, if false, NO. However, if the argument is **between 0 and 1**, a probabilistic decision is made. This value is treated as the probability of truth, and a random decision is made to return either YES or NO. For example: 0.7 CHECK would give YES about 70% of the time when called and NO about 30%. This allows for elegant programming of stochastic methods and other cases requiring probabilistic decision-making.

Here is an example of calculating the n-th Fibonacci Number (without RESULT) using CHECK and IF_YES/IF_NO:

```
MONADIC
   DUP 2 < CHECK
      :DUP 1 - fibon
       SWAP 2 - fibon
       +;
   IF_NO
   DISCARD IF_YES
   1 IF_YES;
OPERATOR @fibon SET.
```

Fields Manipulation

A series of commands in Virtue manipulate fields. We've already mentioned RESHAPE, which uses a dimension vector to reorganize a field, and its counterpart SHAPE, which returns a dimension vector for a given field. Additionally, there's ENLIST, which converts all members of a deeply nested field into a one-dimensional vector.

Field sizes and dimensions are modified by other important operations, primarily reductions. These involve decreasing the field size by one (last or first, depending on the command) dimension while performing an operation, such as + (known as sum) or * (known as product), across the remaining dimensions.

REDUCE & REDUCELASTAXIS

For example " `(1 2 3 4 5 6)[3 2] ADD REDUCE.` " will yield 6 15 (1+2+3 4+5+6), a "`(1 2 3 4 5 6)[3 2] ADD REDUCELASTAXIS` " will yield 5 7 9 (1+4 2+5 3+6).

REDUCE applies the operation (in this case, addition) along the first dimension of the field (from left to right, as if reading a matrix row-wise), resulting in a new field with one less dimension. REDUCELASTAXIS, on the other hand, performs the operation along the last dimension of the field (from top to bottom, as if reading a matrix column-wise), again reducing the dimensionality by one. This highlights how REDUCE and REDUCELASTAXIS can produce different results based on the orientation of operations across the field's dimensions.

FIRST & REST

Similar to LISP, Virtue has FIRST (CAR[29]) and REST (CDR). FIRST returns the first dimension, while REST returns all the remaining dimensions. For example: " (1 2 3 4 5 6)[3 2] FIRST REST FIRST. " yields 4.

---

29 The LISP names for these operations originate from the register parts names in IBM's 704, where data was stored in the first implementation of LISP. In LISP, the left part of a "cell" `[ (a . b) ]` is usually an atom (i.e., not a linked list), while the right member is typically a memory pointer to the next such pair. Thus, the left element is referred to as "Contents of Address (part of) Register = CAR", and the right one as "Contents of Decrement (part of) Register = CDR".

REVERSE & TRANSPOSE

REVERSE reverses the field, i.e., the last element of the field becomes the first, and vice versa. TRANSPOSE performs a transposition (as expected), i.e., the first dimension becomes the last, and vice versa.

Stochastics

Several words in Virtue are specifically dedicated to stochastic computing.

RANDOM & ROLL

RANDOM takes one argument (scalar or field) and generates a corresponding field of random real (floating-point) numbers within the range from zero to the number specified in the input field.

ROLL functions similarly to RANDOM, but the result is an integer within the same range.

DEAL

The DEAL command is like drawing cards, as it allows selecting m unique integers from a range of 1 to n without repetition. In other words, if you want to get a vector of 256 elements with no repeated numbers, randomly shuffled, you can write 256 256 DEAL and receive 256 unique integers in random order. For example, 100 10 DEAL will give you 10 non-repeating numbers between 1 and 100.

PROBABLY

As previously explained, Virtue has multidimensional (1-8) fuzzy logical variables, and each logical operation (e.g., the aforementioned AND or OR) works on all dimensions separately. For instance: * 1i0j1k1 0i1j1k0 AND yields 0i0j1k0, meaning false in the real and i-th parts of the quaternion, true in the j-th part, and false in the k-th part. Earlier, it was mentioned that in all logical operations, values greater than or equal to 1 are considered true (T), while those less than or equal to 0 are considered false (F). Values between 0 and 1 represent probabilities of truth when a decision is required (see CHECK).

PROBABLY: This word enables converting any numerical field into a true/false field based on the probabilities provided in the input field. For example, (0.3 0.5 1) PROBABLY will yield a three-element vector with probabilistic outcomes: The first element has a 30% chance of being true. The second element will statistically give an equal probability of being true or false. The third element will always result in true.

Debugging - Meta-Commands

As an interactive language, Virtue enables direct testing of individual ideas for solving algorithmic problems. However, it's not possible to write everything in a way that can be executed directly sentence by sentence. When we experiment with elements of an

algorithm, we'll start writing functions, naming various things, and creating new verbs (OPERATORs).

To find and fix errors in a program, it's crucial to understand what the program does, be able to pause it, inspect intermediate results, and possibly intervene with some data (when we realize what's wrong and want to test the rest of the program).

TRACE & TRACEALL

Using TRACE or TRACEALL, either interactively or within any Virtue text, will cause the Virtue interpreter to display the functional level (i.e., "stack" height of nested function calls), the word/data index in the current execution context, the number of data elements on the stack, the object type in the program and the object itself [e.g., SHAPE, (1 2 3), etc.]. Example output:

```
Trace ( 0:  2) [ 3]:      . OPERATOR:             DUP
```

STEP & STEPALL

Unlike TRACE and TRACEALL, STEP and STEPALL behave similarly but pause before executing each displayed item (displayed in the same way as "Trace" but with a "Step" indicator). They wait for the user's input to continue. Entering an empty line will proceed to the next program element. If a period (".") is entered, it will display what's at the top of the stack. However, the user has the opportunity to utilize all Virtue provides and thus can adjust some data mid-execution to test the rest of the program with that new data.

RUN & RUNALL

In brief: TRACE or STEP are enabled until the RUN command. TRACEALL and STEPALL are enabled until the RUNALL command.

The End

Since Virtue is primarily an interactive language, there must be some command to exit that interactive environment.

OFF = QUIT = ENDPROCESS

The three listed words are synonyms that enable the exit from the interactive environment of Virtue, and end the performance of the interpreter. However, this only applies at the direct level of the user's writing. While maintaining consistency of meaning, OFF statements terminate the execution of any sequence of instructions. In other words, the output of " 2 MONADIC DUP ENDPROCESS +; EXECUTE " will not be the sum of 2 and 2, but will leave two 2's on the stack. In this context, i.e. within function phrases, they behave the same as RETURN, and that is the reason that in this example we used the context more appropriate synonym ENDPROCESS.

List of Virtue Words

Please note that this list includes all recognized words in Virtue, some of which may not be fully implemented or meet all specification criteria. Additionally, as Virtue is under continuous development with new ideas (and words), this list might not be entirely aligned with the specified or implemented language version. This list serves primarily as a brief overview. Important Notes: Only full words are listed here, although many frequently used words have symbolic variants. The list includes all words, some of which are synonyms (e.g., INTERVAL and SPACE, or SUM being synonymous with the phrase ADD REDUCE, also expressible symbolically as +//). Certain words can be prefixes (like REFLECT or WARP for MASK) or even serve as prefixes for reductions (e.g., ADD or MULTIPLY).

```
ADD, AND, APPEND, ASSEMBLE, ASSIGN, AT, BINOMIAL, BOOLEAN,
CARTESIAN, CATENATE, CATENATEAXIS, CATENATEFIRSTAXIS, CEILING,
CHECK, CIRCULAR, CLEAR, CONJUGATE, CONTRADICTION, DEAL,
DECAPSULATE, DECODE, DEPTH, DIP, DIRECTION, DISASSEMBLE, DISCARD,
DISCLOSE, DISCLOSEAXIS, DIVIDE, DO, DROP, DROPAXIS, DUMPSTACK,
DUPLICATE, EACH, ENASCII, ENCAPSULATE, ENCLOSE, ENCLOSEAXIS,
ENCODE, ENDPROCESS, ENLIST, EQUIVALENT, EXECUTE, EXPAND,
EXPANDAXIS, EXPANDFIRSTAXIS, EXPONENTIAL, EXTRODUCE, FACTORIAL,
FIND, FIRST, FLOOR,, FORMAT, FORMATSPECIFIED, FUNCTIONEND,
FUNCTIONS, GET, GOTO, GRADEDOWN, GRADEDOWNCOLLATING, GRADEUP,
GRADEUPCOLLATING, GREATER, IDENTICAL, IF, IF_NO, IF_YES, INDEX,
INDEXAXIS, INDEXING, INDEXOF, INNERPRODUCT, INPUT, INTERVAL,
INTRODUCE, ISBOOLEAN, ISLOGICAL, JUMP, LAMINATE, LESS, LOAD,
LOGARITHM, LOGICAL, LOOP, MAGNITUDE, MASK, MATCH, MATRIXDIVIDE,
MATRIXINVERSE, MAXIMUM, MEMBER, MINIMUM, MULTIPLY, NAND,
NATURALLOG, NEGATIVE, NOOP, NOR, NOT, NOTEQUIVALENT, NOTGREATER,
NOTIDENTICAL, NOTLESS, OR, OUTERPRODUCT, OUTPUT, PARALLEL,
PARTITION, PARTITIONAXIS, PICK, PITIMES, POLAR, POWER, PRINT,
PROBABLY, QUOTEDIN, RANK, RAVEL, RAVELAXIS, READ, RECIPROCAL,
REDUCE, REDUCEAXIS, REDUCEFIRSTAXIS, REDUCENWISE, REDUCENWISEAXIS,
REDUCENWISEFIRSTAXIS, REFLECT, REMEMBER, REPLICATE, REPLICATEAXIS,
REPLICATEFIRSTAXIS, RESHAPE, RESIDUE, REST, RETURN, REVERSE,
REVERSEAXIS, REVERSEFIRSTAXIS, RIGHT, ROLL, ROTATE, ROTATEAXIS,
ROTATEFIRSTAXIS, SAVE, SCAN, SCANAXIS, SCANFIRSTAXIS, SET, SHAPE,
SIGNUM, STRONGAND, STRONGNAND, STRONGNOR, STRONGOR, SUBTRACT,
SWAP, TABLE, TAKE, TAKEAXIS, TEXTAPPEND, TEXTREAD, TEXTWRITE,
TRANSPOSE, TRANSPOSEGENERAL, VARIABLES, WIPE, WITHOUT, WRAP, WRITE
```

And a Few More Words

And a Few More Words, not Virtue words, but words about this brief language overview.

It's extremely challenging to describe a language as complex and uniquely powerful as Virtue in such a concise text. Nevertheless, I believe that this brief introduction has

managed to convey the essence of Virtue, providing readers with a fundamental understanding of its capabilities.

The approach to programming in Virtue diverges significantly from traditional views on programming. Conventional computer programming originated from simple bit-level operations, evolving into more complex instructions like take, store, add, compare, and jump (e.g., the early 1960s Eurocomp LGP21 had only 16 instructions, with its instruction codes mirroring the Flexowriter typewriter keys – i: input, m: multiply, j: jump, etc.). Even our most advanced languages today, such as C++, Java, Python, and the vast majority of others, still rely on this fundamental, linear execution of basic commands, despite the abstractions they offer. Multidimensionality must be explicitly programmed. Complex numbers don't know they're complex, and octonions don't know they're octonions. If you want to use them, you have to program their entire mathematics from scratch. In contrast, Virtue inherently understands how arithmetic operations are performed on all data types (including combinations thereof), allowing seamless multiplication of quaternions and octonions without requiring explicit programming of their mathematics. Try creating a program in one of those popular languages that accepts real, complex, quaternion, or combined fields as arguments without modification.

The approach to solving problems in Virtue often differs significantly from the way problems are tackled in most programming languages (with exceptions like APL, LISP, Prolog, Haskell, etc.).

As some possible tips for Thinking in Virtue, envision your problem as a whole, with all relevant data processed through a unified operation to yield the result. Think in terms of index fields, where each element's address within the field is its value. This leads to the possibility of processing every piece of data at a specific location with a single command.

Visualize your data in geometric shapes and imagine how you can transform this entire shape into your desired outcome. This approach can be particularly helpful.

Of course, it's impossible to provide definitive guidelines on how to program, as aptly put by one of my professors: "You cannot learn to play the violin from a blackboard!"

Documentation

(Very) Preliminary "Virtue Language Reference Manual": Detailed explanations and syntactic definitions can be found at:
http://grgur.irb.hr/Virtue/The.Virtue.of.Programming.in.Virtue.pdf

Although for an older version of Virtue, for most purposes, as a kind of "quick reference" the Virtue-0.3 description can be used, please beware that some things evolved and changed in the meantime, as, for example, the usage of prefixes for global/local names, so use this only as a more full overview than this introductory text: http://grgur.irb.hr/Virtue/Virtue.language.0.3.html

Introduction to the Core Idea: Available on the main Virtue website page: http://grgur.irb.hr/Virtue/

Virtue Interpreters for Various Machines and Operating Systems: Can be found at: http://grgur.irb.hr/Virtue/Downloads.html. Please note that not all combinations of machines and operating systems have the latest versions available. However, as mentioned earlier, due to adhering to standard C programming, it's only necessary to recompile the latest version on those machines.

Scientific Papers related to Virtue or programming methods that inspired its developme: The following links point to research papers exploring aspects of Virtue or programming approaches that were either a consequence of or an inspiration for the language's development:

https://www.researchgate.net/publication/269106250_An_Approach_Towards_High_Productivity_Computing (2014)

https://www.researchgate.net/publication/265796309_Virtue_-_A_different_approach_to_humancomputer_interaction (2014)

https://www.researchgate.net/publication/256089611_Scientific_Visualization_by_Cluster_Computing (2005)

https://www.researchgate.net/publication/220856017_Towards_a_Grid_Applicable_Parallel_Architecture_Machine (2004)

https://www.researchgate.net/publication/256089664_Multiple_Programme_Single_Data_Stream_Approach_to_Grid_Programming (2004)

https://www.researchgate.net/publication/256089691_ISOCOM_20_Filter_System_User_Documentation_Flow_Through_Filter_Language_and_Graphics_Organisation_Language (1992)