Volume 25, No.3

# 5 – a stack-based array language

*Bernd Ulmann (ulmann@vaxman.de)*

5 is a portable and extensible stack-oriented array language that combines the features of APL and Forth and some ideas from Perl to yield a highly interactive environment for the professional developer as well as students of computer science. The interpreter is very lightweight (about 650 Kb all in all, including documentation) and runs readily on a variety of operating systems including Windows, LINUX, Mac OS X and even OpenVMS (VAX, Alpha, Itanium). 5 is Open Source and resides on SourceForge[1].

## What is 5 and why was it developed?

The development of 5 started in August 2009 during a boring train ride when I programmed my beloved HP48GX pocket calculator to kill some time. As much as I love the stack-oriented approach of HP's calculator-language RPL, I always thought that one could have done better by not combining Forth and LISP but Forth and APL instead. I wondered what such a language inheriting the main ideas of APL and Forth (and some bits from Perl) would look like and started writing a simple interpreter as a proof of concept. Since this new language looked like 'Forth on steroids' the name 5 seemed quite natural (quickly abandoning the idea of 'Fifth').

It turned out that a stack-based array language yielded very concise code and during the next six months the language was steadily extended until it turned out that this first interpreter was indeed too limited in its design to incorporate all of the new ideas that popped up. Thus Mr Thomas Kratz and I decided to restart from scratch and implement a more flexible interpreter. (Most of the interpreter has been written since by Mr Kratz, whom I would like to thank for this truly great work).

## First steps

The 5 interpreter is quite mighty and thus the following sections can and will only give a brief overview by showing and explaining a couple of typical 5-programs – much more information can be found in the introductory manual[2].

### Getting and installing 5

The 5 interpreter and its accompanying documentation can be downloaded from SourceForge[1]. The only prerequisite for installing 5 is a Perl interpreter, which is already found on most UNIX systems and easily installed on Windows machines (ActiveState Perl works fine). Since the 5 interpreter does not need any special Perl-packages, its installation does not require any changes on an existing Perl installation. To install 5 all you need to do is to unzip the distribution kit, which is only about 330 Kb in size to any suitable location and make the file `5` executable. On UNIX systems you might want to extend your environment variable PATH by the directory into which you unzipped the distribution kit. On Windows and OpenVMS systems the interpreter can be started most easily from the command line by typing `perl 5` (although one would normally define a foreign command on OpenVMS to make 5 directly callable).

The distribution kit not only contains the interpreter itself but also detailed documentation, as well as many examples which will help one get used to the language and the interpreter.

### Using 5 interactively

Let us use the 5 interpreter interactively to simulate dice being thrown 100 times and computing the arithmetic mean of the outcomes:

```
6 100 reshape ? int 1 + '+ reduce 100 / .
```

There is not much to say about this simple program which would not be obvious to people inclined to work in APL and other array languages. The most noteworthy thing is the stack-oriented operation of the interpreter, so all programs are read strictly from left to right. (As a result there are neither parentheses nor operator precedence rules in 5.)

1. First two scalars, `6` and `100`, are pushed onto the stack.
2. These values are used for the operator `reshape` which removes both from the stack and pushes an array containing 100 elements back onto the stack: `[6 6 6 … 6]`
3. To this array, the unary `?` operator is applied, which generates pseudorandom numbers in a range between 0 and a given maximum value (exclusively). Since this operator is unary it is applied to all elements of the vector automatically and yields a new vector on the stack containing 100 elements between 0 (inclusively) and 6 (exclusively). Applying the unary `int` operator to this vector yields a vector of integer values which are then incremented all by one due to `1 +`.
4. The next step pushes the name of the binary addition operator onto the stack, `'+`, and applies the `reduce` function, which in turn sums all vector elements into a scalar.
5. Dividing this result by 100 and printing it to the console is accomplished by `100 / .`

Although one has to get used to the reverse-Polish notation style of 5 it turns out to be very efficient and intuitive after a short time of playing with the interpreter. The main advantage of the stack-based nature is that one can build complex expressions iteratively and 'watch' the results of a computation through the different stages step by step. (Using the word `.s` one can generate a pretty printed view of the stack without destroying its contents, which is quite handy for understanding the actions of a program.)

## More complex examples

The following examples introduce some of the more sophisticated features of 5 – due to the complexity of the language many concepts will be mentioned only briefly – a comprehensive description of the language and its many operators and functions can be found in the documentation (see above).

### Generating a list of primes

One of the archetypical examples found in nearly every introductory text for APL is the generation of a list of primes without any explicit loops or the like. The following program shows how this is done in 5:

```
: prime_list
1 - iota 2 + dup dup dup
'* outer swap in not select
;

100 prime_list .
```

This example is much more complex than the one before and could be run from a file using 5 in batch mode. To accomplish this just call the 5 interpreter with the name of the source code file as a command line parameter. (5 also supports quite a lot of qualifiers to get statistical information about program runs and the like, which are described in the documentation.) Assuming that there is a file named `prime.5` containing the code shown above, it can be run by typing `5 prime.5` or `perl 5 prime.5`.

This example introduces the concept of so called 'user defined words' ('UDW' or just 'word' for short) that effectively extend the language itself and can be used in exactly the same way as built-in functions and unary or binary operators (in that respect they are much more powerful than the traditional words of Forth which have no provisions for acting as unary or binary operators extending their usability to nested data structures).

First a word named `prime_list` is created – the colon starts a word definition that ends with a semicolon. This user-defined word is neither unary nor binary, so it just sees the stack as it is and operates on it. (In contrast to that, unary and binary words get a local stack with only one or two elements on it to operate on when they are being called. When such a unary or binary word terminates only the top most stack element of its local stack is copied back to the main stack of 5 thus unary or binary operators are side-effect free with regard to the main stack.)

The basic idea of generating a list of primes up to some value `n` is to generate two vectors `[2 3 4 … n]` and generate a matrix by applying an outer product operator to these two vectors. Since this matrix obviously contains only non-primes, it can be used to select all primes from a copy of such a vector. The vector itself is generated by `1 - iota 2 +` which expects a number like `100` on the stack: Subtracting one yields `99`, applying `iota` yields a vector `[0 1 2 … 98]`, adding two to this vector yields the desired vector `[2 3 4 … 100]`. The command sequence `dup dup dup` creates three copies of this vector which will be needed soon.

In the next step the name of the multiplication operator, `'*`, is pushed onto the stack and `outer` is called, which expects an operator's name ( `*` ) and two vectors on the stack and creates a matrix as the result of an outer product in this case.

`swap` swaps the two topmost stack elements, so now one of the remaining copies of the vector is on top and the matrix is the second element from top. Applying the `in` function generates a vector containing a `1` in every place corresponding to an element of the vector that exists in the matrix and a `0` otherwise. Inverting this vector with `not` yields a selection vector which is then applied to the last copy of the original vector by `select`. This yields a vector containing prime numbers between `2` and `n` only.

The main program only consists of `100 prime_list .` This places the value `100` onto the stack, calls the word `prime_list` and prints the resulting vector using the dot.

### Sum of cubes

The following two-liner computes all natural numbers less than 1000 that equal the sum of the cubes of their digits:

```
: cube_sum(*) "" split 3 ** '+ reduce ;
999 iota 1 + dup dup cube_sum == select .
```

The first line again defines a word but this time it is a unary word – denoted by `(*)` following the name of the word. This has the effect that this word will not only work on scalar values but will be automatically applied to all elements of nested data structures. (So applying this word to a vector like `[1 2 3]` will implicitly and automatically apply it to the three vector elements `1`, `2` and `3` and return another three element vector containing the particular results.) The star denotes that the type of the argument is not relevant (a later example will show the ability of 5 to 'dress' data structures – the 5 way of overloading operators etc.).

What does the word `cube_sum` do? First of all it pushes an empty string onto the stack and calls the `split`-function. This function expects a regular expression on the stack and splits the scalar found below on every place where this expression matches. Since the expression is an empty string in this case, it will perform a split after each character of a value. The nice thing is that this naturally extends to numerical values, too – if there was the value `123` on the top of the stack prior to performing `"" split` the result of this operation would be a vector `[1 2 3]`.

This vector is then cubed element wise by `3 **` and summed (element wise) yielding a scalar value by `'+ reduce`, so the word `cube_sum` expects a value on the stack and returns the sum of the cubes of its digits.

The main program is equally simple: First a vector running from `1` to `999` is generated by `999 iota 1 +`. This vector is then copied two times with `dup dup` before `cube_sum` is applied. Since `cube_sum` is a unary word, it will be applied in an element wise fashion to the elements of this vector and yields another vector with 999 elements which are the sums of the cubes of the digits of the numbers of the original value. This cube-sum-vector is then compared element wise with one of the copies made before, which yields another vector with 999 elements being `1` or `0` reflecting the result of the comparison operator. This vector is in turn used to `select` only those elements from the last copy of the original vector that equal their digit-cube-sum, which is then printed with the dot.

## Dressed data structures

If that were about all that 5 can do, it would not be too worthwhile but there is more: 5 allows one to "dress" data structures – i.e. mark some data as being of a certain type like a complex number, a quaternion, a matrix, whatever. The following example shows how to use this feature in the generation of a Mandelbrot set:

```
: d2c(*,*) 2 compress 'c dress ;

: iterate(c) [0 0](c) "dup * over +" steps reshape execute ;

: print_line(*) "#*+-. " "" split swap subscript "" join . "\n" . ;

75 iota 45 - 20 /
29 iota 14 - 10 /
'd2c outer

10 'steps set

iterate abs int 5 min 'print_line apply
```

What is a Mandelbrot set anyhow? It is the result of applying an iterative calculation to points of the complex plane, so first of all we will need a matrix of complex numbers. The 5 interpreter has no idea what a complex number might be but it is easy to extend the language by overloading operators to handle data dressed in a special way. So the basic arithmetic operators are already overloaded in `mathlib.5` to handle complex numbers, which are dressed by the letter `c`. This 'dress code' is just a convention – one could have chosen anything but in order to keep 5 code short and concise, a single letter was chosen to denote complex numbers (`m` denotes a matrix, `v` a vector and `p` a polar coordinate).

Generating a matrix from two vectors by creating an outer product was already shown in the prime number example above. We will use this technique to generate a matrix consisting of complex numbers. Therefore we need a binary word which takes two scalar values and returns a complex number made from these two values. This word is called `d2c` in the code shown above, short for "dupel to complex". Since the name of the word is followed by `(*,*)` it is a binary word which does not care about the type of its arguments. All that it does is to compress the two values found on its local stack by `2 compress` into a simple two-element vector. This vector is then dressed by `'c dress` to form a complex number. Let us assume that `d2c` is called with `1 2 d2c`, so it finds the values `1` and `2` on its local stack. Executing `2 compress` yields the vector `[1 2]` which is then dressed to return `[1 2](c)` – a complex number.

To see how this word is used, let us look at the three lines in the middle of the program: `75 iota 45 - 20 /` generates a vector `[-2.25 -2.2 -2.15 … 1.3 1.35 1.4 1.45]` while `29 iota 14 - 10 /` yields `[-1.4 -1.3 … 1.3 1.4]` respectively. These two vectors are then combined into a matrix by using `d2c` as the binary operator for the `outer`-function. The result of this is a two dimensional matrix of complex numbers – the basis of our Mandelbrot set.

Now that we have a complex matrix, a unary word is needed that operates on complex numbers and performs the necessary iteration for a Mandelbrot set. This iteration has the form $z_{i+1}=z_i^2+c$, where c is a point of the complex plane with $z_0=0$. If this series is non-divergent, the point c belongs to the Mandelbrot set. In the program shown above this iterative formula is applied 10 times and the resulting value is used to choose a display character for the point c in question.

To compute this iterative formula, the word `iterate` is defined, which is a unary operator expecting a complex number which is denoted by the start of the word definition: `: iterate(c)`. In a first step this word pushes the complex number `[0 0](c)` onto the stack which serves as $z_0$. A single iteration step can now be performed by the instruction sequence `dup * over +`. The function `dup` makes a copy of the value on the top of the stack, `*` multiplies the two topmost stack elements, effectively computing $z_i^2$ while `over` fetches the element below the top of stack, which is c, so $z_i^2+c$ is computed with `+`.

To perform a given number of iterations a sequence of these steps must be generated. A traditional language like C or Java would need an explicit loop for this, but array languages like APL or 5 have the means to express this much more elegantly. ("Look Ma – no loops!") The main program sets a variable named `steps` to `10` which is used in this word to control the number of iteration steps being performed. The result of `"dup * over +" steps reshape` in this case yields a one-dimensional vector with 10 elements, looking like this (effectively unrolling the loop): `["dup * over +" … "dup * over +"]` This vector is then used as an instruction stream by means of the `execute` function, which effectively computes the iterative sequence desired.

The main program now calls `iterate`, which is implicitly applied to all elements of the complex matrix built before. The result of this is another complex matrix, which is transformed into a matrix of simple scalars by applying the `abs`-operator to it (`abs` has been already overloaded in `mathlib.5` to work on complex numbers and returns simple floats). The resulting elements are then capped by `5 min` and then another unary user defined word, `print_line`, is applied in a row-like fashion to the matrix by using it as an argument to the `apply`-function.

`print_line` is now called for every row of the matrix. It first generates a vector `["#" "*" "+" "-" "." " "]` by `split`ting the string `"#*+-. "` on an empty regular expression and then uses the elements of the line vector, which are integers between `0` and `5`, as an index into this character vector. The result is a vector containing as many characters as the line contained integer values. This vector is then concatenated into a simple string by `join`ing it with an empty string. This resulting string is then printed with a newline character appended. The resulting picture looks like this:

```
                                  #
                                   * **
                                   ##*
                                  *#####
                                 +*####*-     *
                               *################*#*#*
                              #*####################
                  *          *######################-*
                 -##*###*..*########################
                 *##################################
               *####################################
     +*#*#*#***##########################################*
               *####################################
                 *##################################
                 -##*###*..*########################
                 *          *######################-*
                              #*####################
                               *################*#*#*
                                     +*####*-     *
                                      *#####
                                        ##*
                                        * **
                                       #
```

## Conclusion

Although there is much more to say about 5, I hope that the few examples given above made you curious about this language and I would like to refer you to the extensive documentation which comes with the installation kit. Why should one use 5 when there are APL, J, K etc. implementations available? Some things that may speak in favour of 5 are listed below:

1. 5 is Open Source and easily portable to any architecture for which a Perl interpreter exists.
2. Since 5 does not need any special characters there is no need to install additional fonts, which makes its installation even simpler.
3. Installing 5 does not require any special rights – even end-users can install it locally in any directory, even `C:\Temp` on Windows systems, which makes the interpreter an ideal tool for ad-hoc analyses and experiments at customer locations.
4. The interpreter is very well structured, which makes extensions to the 5 code quite straight forward, thus facilitating experiments in language design etc. (The complete interpreter consists of only 2895 lines of Perl code and 457 lines of 5 code contained in the standard libraries `stdlib.5` and `mathlib.5`.)
5. 5 is an emerging language where the individual can have a real impact on the directions of future developments. The development of the interpreter is still ongoing and we would love to

hear about your suggestions and needs.

Some of the areas which will see future developments are those listed in the following:

The mathematical library `mathlib.5` needs to be extended to overload more operators for complex numbers, polar coordinates, quaternions and the like. Also the library is still lacking most of the common linear algebraic operators and functions.

The interpreter currently lacks powerful operators for transposing and rotating matrices etc.

We need more test cases for the interpreter – although there are a lot of test cases defined, which are run every time changes to the interpreter have been made, these are no longer sufficient and need to be extended heavily.

We need example programs from the field to learn more about the power of 5 and to enhance the interpreter.

We would love to hear from you and I hope that I could interest you in this new array language. Have fun with 5 and happy array programming.

## References

1. 5 at SourceForge: lang5.sourceforge.net
2. 5 documentation at SourceForge: lang5.sourceforge.net/viewvc/lang5/trunk/doc/introduction/introduction.pdf