
5 – a stack based array language implemented in Perl

Bernd Ulmann

YAPC::EU 2010

Hochschule fuer Oekonomie und Management, Frankfurt

First of all, **5** is a blend between APL and Forth¹.

Having said this a couple of questions may raise:

- Why would one implement yet another programming language – especially an array language?
- Of all the programming languages on earth – why take APL and Forth as the basis for a new language?

The first question is easy to answer: Because it is fun and it is interesting and I really like dynamic languages and...

The second question is not as easily answered but here are some rather personal reasons²:

¹If this sounds strange then remember HP's programming language RPL for their late pocket calculator series which was based on LISP and Forth.

²Do you remember the "Lithp"-talk by Marty Pauley at the YAPC::EU 2006 and my follow on lightning talk about APL? :-)

■ About APL:

- +++ I love APL's mathematical purity.
- +++ APL is incredibly powerful!
- ++ APL is highly interactive.
 - It requires a really arcane character set.
 - It is completely unreadable for the uninitiated and hard to read even for those knowing APL³.
- There is only one truly free APL interpreter available⁴.

■ About Forth:

- ++ Very simple to implement and use.
- ++ Highly interactive, too.
 - Its central data structure, the stack, only supports basic data types (sometimes only integers).

³APL is sometimes called a write-only language.

⁴A+ – cf. <http://www.aplusdev.org>

- $$\nabla Z \leftarrow \text{FFT } X; C; D; E; J; K; LL; M; N; O$$
- [1] $LL \leftarrow \lfloor 2 * -O - 1 M \leftarrow \lfloor 2 \otimes N, 0 \rho E \leftarrow 1 - 2 * \sim O \leftarrow 1 J \leftarrow 1 L$
 $\leftarrow 0, 0 \rho K \leftarrow 1 N \leftarrow \bar{1} \uparrow \rho X$
- [2] $\rightarrow (M > L \leftarrow L + 1) / 1 + \rho \rho J \leftarrow J, N \rho 0 1 \circ . = ($
 $2 * L) \rho 1$
- [3] $Z \leftarrow X [; (L \leftarrow 0) + (\phi LL) + . * J \leftarrow (M, N) \rho J]$
- [4] $X \leftarrow 2 1 \circ . \circ \circ (-O - K) \div \bar{1} \uparrow LL$
- [5] $Z \leftarrow Z [; K -, LL [L] * J [L ;]] + (\rho Z) \rho (- / X [; D] * X$
 $Z [; C]), + / X [; D \leftarrow O + N \rho LL [E + M - L] * -O - 1$
 $2 * LL [L]] * \ominus Z [; C \leftarrow K +, LL [L] * 0 = J [L ;]]$
- [6] $\rightarrow ((M + O) > L \leftarrow L + 1) / 5$
- $$\nabla$$

...see what I mean? Believe it or not, but this implements a fast Fourier transform⁵.

⁵Cf. [Giloi 77][p. 212]

. . . a stack based array language, called **5**:

- The stack can hold scalars as well as nested data structures.
- It is possible to write so called *user defined words* which correspond to functions in traditional languages.
- The interpreter automatically applies unary or binary operators or unary or binary user defined words on all elements of nested data structures (implied multidimensional `map`).
- **5** is really powerful and fun to program in – and it does not need a strange character set! :-)
- **5** is completely implemented in Perl and is thus easily portable (currently **5** is used on Max OS X, OpenVMS and Windows).
- **5** is available at no cost at all and everybody is welcome to contribute to **5**'s development!
- The **5**-repository can be found at <http://lang5.sourceforge.net>.

Let's start with a simple example – calculate $\sum_{i=1}^{100} i$ and $100!$ in **5**:

```

1 # Define some simple words
2 : gauss iota 1 + '+' reduce ;
3 : factorial iota 1 + '* reduce ;
4
5 # Use these new user defined words:
6 100 dup gauss . factorial .

```

```

alberich$ 5 gauss_factorial.5
----> loading mathlib.5
----> loading stdlib.5
loading gauss_factorial.5
5050
9.33262154439441e+157

```

How does this work? Let us have a look at this simplified version of the sum:

```

1  ┌────────────────────────── simple_sum.5 ───────────────────────────┐
   │ 100 iota 1 + '+' reduce .                                          │
   └────────────────────────── simple_sum.5 ───────────────────────────┘

```

- `100 iota` generates a vector `[0 1 2 ... 99]`.
- Adding 1 to this vector yields `[1 2 3 ... 100]`.
- `'+'` pushes the operator "+" onto the stack.
- The reduce-function expects an operator on the top of the stack (*TOS* for short) and a vector below. It then applies this operator between all successive vector elements yielding `1 + 2 + 3 + ... + 100` in this case.
- The `.-`-function prints the TOS.
- That's all – no explicit loops, nothing...

Suppose you have to simulate throwing a six sided dice 100 times and calculate the arithmetic mean of the results you get:

```
1 : throw_dice
2   6 over reshape
3   ? int 1 +
4   '+ reduce swap /
5 ;
6
7 100 throw_dice .
```

```
alberich$ 5 throw_dice.5
----> loading mathlib.5
----> loading stdlib.5
loading throw_dice.5
3.35
```


How does this work?

- `100 throw_dice` pushes 100 onto the stack and calls the word `throw_dice`.
- `6 over` yields 100 6 100 on the stack.
- The `reshape`-function expects a dimension vector (or a scalar in the one-dimensional case) on the TOS and rearranges the object found below accordingly. In this case the result is a vector of the form `[6 6 6 ... 6]`.
- The unary `?`-operator generates a pseudo random number between 0 and the number found on the TOS. Since it is unary it is automatically applied to all elements of the vector we just created.
- `int 1 +` gets rid of the fractional part of the resulting vector elements and makes sure they are between 1 and 6.
- `'+ reduce` then computes the sum of the vector elements.
- `swap /` swaps this sum and the 100 from the beginning and divides, yielding the arithmetic mean.

No example collection would be complete without Fibonacci numbers...

```

1  : fib{u}
2  dup 2 < if drop 1 break then
3  dup 1 - fib swap 2 - fib +
4  ;
5
6  10 iota fib .

```

fibr_apply.5

```

alberich$ 5 fibr_apply.5
----> loading mathlib.5
----> loading stdlib.5
loading fibr_apply.5
[ 1 1 2 3 5 8 13 21
 34 55 ]

```

How does this work?

- The `{u}` following the name `fib` of the user defined word denotes that this will be an unary user defined word, so everything that holds true for built-in unary operators will also work for this word. Please note that this includes the automatic application of this word to all scalar elements of nested data structures.
- `10 iota fib .` creates a vector `[0 1 2 ... 9]` and applies the word `fib` to every single element.
- `fib` is a simple recursive implementation of the Fibonacci rule.

Recently I found the following Fortran-example program⁶ which prints all numbers between 1 and 999 which are equal to the sum of the cubes of their digits:

```
sum_of_cubes.for
1 program sum_of_cubes
2 implicit none
3 integer :: H, T, U
4 do H = 1, 9
5     do T = 0, 9
6         do U = 0, 9
7             if (100*H + 10*T + U == H**3 + T**3 + U**3) &
8                 print "(3I1)", H, T, U
9         end do
10    end do
11 end do
12 end program sum_of_cubes
sum_of_cubes.for
```

Horrible, isn't it? Let's do it in **5**:

⁶Cf. [Adams et al. 09][p. 41]

The **5**-solution is a bit shorter ("Look Mom, no Loops!"):

```

1  : cube_sum{u} "" split 3 ** '+ reduce ;
2  999 iota 1 + dup dup cube_sum == select .

```

```

alberich$ 5 sum_of_cubes.5
----> loading mathlib.5
----> loading stdlib.5
loading sum_of_cubes.5
[ 1 153 370 371 407 ]

```

How does this work?

- `cube_sum{u}` defines an unary word.
- This word pushes an empty string onto the stack and splits the element found below yielding a vector of the individual digits of the number which was found on the stack before.
- It then calculates the cubes of the vector elements by `3 **`.
- This vector of cubed digits is then summed using `'+ reduce`. The word thus transforms a number found on the TOS into the sum of its digit cubes.
- `999 iota 1 +` yields `[1 2 3 ... 999]`.
- Since we need three of these vectors, it is duplicated twice.
- Then `cube_sum` is applied element wise to this vector.
- `==` compares the result of this operation with the first copy of the original vector yielding something like `[1 0 0 ...]`.
- `select` selects elements from a vector controlled by a corresponding boolean vector.

Generate a list of primes (between 2 and 100 eg.):

```

1  : prime_list
2  1 - iota 2 + dup dup dup
3  '* outer
4  swap in not
5  select
6  ;
7
8  100 prime_list .

```

```
alberich$ 5 prime.5
```

```
----> loading mathlib.5
```

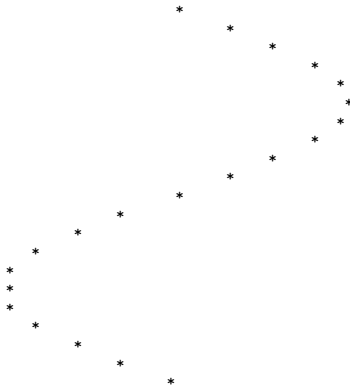
```
----> loading stdlib.5
```

```
loading prime.5
```

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

```
1 : print_dot{u} " " 1 compress swap reshape "*\n" append "" join . ;
2 21 iota 10 / 3.14159265 * sin 20 * 25 + int
   sine_curve.5
```

```
alberich$ 5 sine_curve.5
----> loading mathlib.5
----> loading stdlib.5
loading sine_curve.5
```



Multiply $\begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix}$ by $\begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$:

```

matrix_vector.5
1 : inner+{u} '+ reduce ;
2 : mv* 1 compress '* apply 'inner+ apply ;
3
4 9 iota 1 + [3 3] reshape
5 3 iota 10 +
6
7 mv* .
matrix_vector.5

```

```

alberich$ 5 matrix.5
----> loading mathlib.5
----> loading stdlib.5
loading matrix.5
[ 68 167 266 ]

```

Generate an alternating sequence [0 1 0 1 ...]:

1

```
20 iota 2 % .
```

sequence.5

sequence.5

The first 20 powers of 2:

1

```
2 20 reshape 20 iota ** .
```

powers.5

powers.5

Perfect numbers between 1 and 500:

1

```
: p{u}
```

2

```
dup dup 1 - iota 1 + dup rot swap
```

3

```
% not select '+ reduce ==
```

4

```
;
```

5

```
500 iota 1 + dup p select .
```

perfect.5

perfect.5



5 was developed during the last year and went through two incarnations. Concluding this short talk I would like to mention the following points:

- **5** is quite as powerful as APL although there are still some operators and functions missing which will be implemented soon.
- **5** is a joy to work with – especially since loops and other control structures are rarely necessary due to the APL-like approach to programming.
- The routines for handling deeply nested structures like `shape`, `reshape`, `copy` and many, many more will be moved into a stand alone Perl module which will be made available at CPAN in the near future.
- We are still looking for persons interested in the development of **5** to participate in the development effort, so feel cordially invited to join the current development team (mainly Thomas Kratz and the author).

The author would like to express his thanks to the following persons:

- Mr. Thomas Kratz who did most of the current **5**-interpreter implementation,
- Frederic Fournis and Dr. Reinhard Steffens for many fruitful discussions about **5**,
- My wife, Rikka, for her support and her feedback
- and last but not least you, the audience, for your interest and patience.

The author can be reached at `ulmann@vaxman.de`.

-  [Adams et al. 09] Jeanne C. Adams, Walter S. Brainerd, Richard A. Hendrickson, Richard E. Maine, Jeanne T. Martin, Brian T. Smith, *The Fortran 2003 Handbook*, Springer, 2009
-  [Giloi 77] Wolfgang K. Giloi, *Programmieren in APL*, deGruyter, Berlin, 1977