5 – a stack based array language

Prof. Dr. Bernd Ulmann

18-OCT-2010

Hochschule fuer Oekonomie und Management, Frankfurt

Introduction

Introduction

The following talk is about a new programming language called **5** which is a blend of APL and Forth and is available for a variety of platforms (essentially all systems for which a Perl interpreter is available, including OpenVMS) for free.

5 has been presented for the first time a year ago at the Connect-Symposium 2009 – a lot has happened since then; the language has been extended, the interpreter has been improved for speed etc.

Thus the following talk will not only give an introduction to **5** but also contains a lot of more complex examples as well as some background information about the interpreter itself.

What is **5**?

- **5** is a dynamic programming language (dynamic typing, dynamic memory management).
- **5** incorporates the main features of APL[1] and Forth[2] (*n*-dimensional arrays are the basic data structure, all operators and functions work on a stack, the language is extended by user defined words).
- The **5**-interpreter is written in Perl and easily portable.
- **5** runs out of the box on OpenVMS. (It may thus be used to replace the much missed VAX APL interpreter.)
- **5** is free software.
- **5**'s home is located at `lang5.sourceforge.net`.

[1] Cf. [Giloi 77] and [Katzan 70].
[2] Cf. [Brodie 04] and [Conklin et al. 07].

Why would one create yet another programming language? Aren't there more than enough programming languages already? The main reasons that led to the development of **5** are these:

- Writing compilers and interpreters is really interesting and gives an insight into the design of programming languages which is hard to achieve otherwise.
- Both Forth and APL have features making them more or less unique in the programming languages zoo, so both languages are definitely worth to be at least taken into account as the basis for new developments.
- On the other hand, both languages have their deficiencies like the need for special characters in APL, the non-overloading of basic arithmetic operators in Forth etc.
- Languages based on array operations like APL might be an ideal tool to phrase algorithms for vector processors and GPUs. It should be worthwhile to think about a compiler generating CUDA-code from **5**-programs.

So, in more detail, what is **5**?

- It is a stack oriented language just like Forth.
- In contrast to Forth, the stack can hold scalars as well as nested data structures of arbitrary shape and depth.
- Typing is done dynamically, so there is no need to to declare the type of variables (variables are used rather seldom in **5**).
- The language can be extended by itself making use of so called *User Defined Words* (the distribution kit contains two libraries which contain **5**-extensions written entirely in **5**).
- User defined words can act as unary or binary operators and are thus equivalent to builtin operators.
- Unary and binary operators are automatically applied in an element wise fashion to the elements of nested data structures.

Installing 5

Installing **5** on your system is rather simple as shown in the following:

- Make sure you have a Perl interpreter running on your system (**5** does not need any additional Perl modules to be installed so even a precompiled Perl distribution will be fine).
- Download the **5** distribution kit from `https://sourceforge.net/projects/lang5/files/`. This kit contains the interpreter, a lot of examples and the complete documentation in PDF format.
- Unpack the distribution kit at a location suitable for your environment.

- Extend your PATH environment variable (UNIX) or define a so called foreign command (OpenVMS) to call the interpreter:

  UNIX: `export PATH=$PATH:<path_to_the_interpreter>`

  OpenVMS: `$ FIVE :== PERL <path_to_the_interpreter>5`

- Start the interpreter by calling 5 (UNIX) or FIVE respectively (OpenVMS):

```
alberich$ 5
----> loading mathlib.5
----> loading stdlib.5
5> 100 iota 1 + '+ reduce .
5050
5> exit

alberich$
```

First steps

Programming **5** is rather different from programming in most other languages so the following introduction will rely heavily on examples.

Example 1 – calculating $\sum_{i=1}^{100} i$ and 100!:

$\sum_{i=1}^{100} i$ can be interpreted as the sum of the elements of a vector with unit stride, running from 1 to 100. The same holds true for 100! which is the product of all these elements which leads to the following solution:

Sum and factorial
```
1  100 iota 1 + '+ reduce .
2  100 iota 1 + '* reduce .
```
Sum and factorial

Sum and factorial

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

```
100 iota 1 + '+ reduce .
```

- `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...

This can be written better by introducing two user defined words (UDW) to calculate the Gauss sum and the factorial for any number found on the top of the stack:

Example 2 – simple user defined words

gauss_factorial.5

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

gauss_factorial.5

These two user defined words `gauss` and `factorial` work directly on the value found on the top of the stack (TOS for short) which will work only for scalar values!

Running this example program is simple and yields the following results:

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

One particular strength of **5** is its builtin mechanism of applying unary and binary operators automatically to all elements of nested data structures without any need for explicit loops as the following example shows:

Example 3 – some binary operators

Operators

```
alberich$ 5
----> loading mathlib.5
----> loading stdlib.5
5> [1 2 3] [4 5 6] + .
[    5     7     9  ]
5> [1 2 3] 2 ** .
[    1     4     9  ]
5>
```

Operators

This mechanism can also be used in the case of user defined words which must be declared as unary or binary words respectively:

Example 4 – unary user defined words

gauss_factorial.5

```
# Define two unary words:
: gauss{u}     iota 1 + '+ reduce ;
: factorial{u} iota 1 + '* reduce ;

10 iota dup gauss . factorial .
```

gauss_factorial.5

```
alberich$ 5 gauss_factorial_unary.5
----> loading mathlib.5
----> loading stdlib.5
loading gauss_factorial_unary.5
[ 0   1   3   6   10   15   21   28   36   45  ]
[ 1   1   2   6   24   120   720  5040  40320  362880  ]
alberich$
```

5 allows recursive calls of words, too:

Example 5 – the ubiquitous Fibonacci series:

fibr_apply.5

```
: fib{u}
  dup 2 < if drop 1 break then
  dup 1 - fib swap 2 - fib +
;

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?

- First of all, `10 iota fib .` places a vector `[0 1 2 ... 9]` onto the TOS, calls the unary UDW `fib` and prints the resulting vector.
- `fib` is executed once for each element of the nested data structure it is applied since it is a unary UDW.
- The first step is to check if the value found on the TOS is less than 2 – in this case `fib` will just drop the value and return `1`.
- Otherwise `fib` calls itself twice with new arguments smaller by one and two respectively and returns the sum of the results of these calls.

5 offers a rich complement of functions and operators which allow to generate and restructure nested data structures. The two main functions for this are `shape` and `reshape`:

Example 6 – `shape` and `reshape`

shape and reshape

```
1  alberich$ 5
2  ----> loading mathlib.5
3  ----> loading stdlib.5
4  5> [1 2 3] shape .
5  [    3  ]
6  5> [[1 2 3] [4 5 6] [7 8 9]] shape .
7  [    3     3  ]
8  5> 9 iota 1 + [3 3] reshape .
9  [
10   [    1     2     3  ]
11   [    4     5     6  ]
12   [    7     8     9  ]
13 ]
14 5> 1 [2 2] reshape .
15 [
16   [    1     1  ]
17   [    1     1  ]
18 ]
```

shape and reshape

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

Example 7 – throwing dice

throw_dice.5

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

throw_dice.5

```
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.

Recently I found the following Fortran-example program[3] which prints all numbers between 1 and 999 which are equal to the sum of the cubes of their digits:

sum_of_cubes.for

```
program sum_of_cubes
implicit none
integer :: H, T, U
do H = 1, 9
  do T = 0, 9
    do U = 0, 9
      if (100*H + 10*T + U == H**3 + T**3 + U**3) &
        print "(3I1)", H, T, U
    end do
  end do
end do
end program sum_of_cubes
```

sum_of_cubes.for

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

[3] Cf. [Adams et al. 09][p. 41]

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

Example 8 – sum of cubes

sum_of_cubes.5

```
1 : cube_sum{u}
2   "" split 3 ** '+ reduce
3 ;
4
5 999 iota 1 + dup dup cube_sum == select .
```

sum_of_cubes.5

```
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.

These introductory examples only showed a small fraction of the principles of **5**, its functions and operators.

Nevertheless the power of the stack based approach to an array language may have become apparent – **5** incorporates the best of the worlds of Forth[4] and APL[5].

[4] Cf. [Brodie 04] and [Conklin et al. 07].

[5] Cf. [Giloi 77] and [Katzan 70].

More complex examples

The following word computes the cosine of a value given in radians using the MacLaurin expansion

$$\cos x \approx \sum_{i=0}^{n} (-1)^i \frac{x^{2i}}{(2i)!}$$

with 9 terms.

To accomplish this without an explicit loop three basic vectors representing $(-1)^i$, x^{2i} and $(2i)!$ are generated. Multiplying the first two and dividing the result by the third one yields a vector which is then processed by summing all of its elements using the `reduce` operation.

The following slide shows the complete word definition of `mc_cos`:

Example 9 – cosine approximation

mc_cos.5

```
1  : mc_cos
2      # Save x and the number of terms for future use
3      'x set 9 'terms set
4
5      # Generate a vector containing x ** (2 * i)
6      terms iota dup undef ne x * swap 2 * dup v2i set **
7
8      # Generate the (2 * i)! vector and
9      # divide the previous vector by this
10     v2i ! /
11
12     # Generate a vector of the form [1 -1 1 -1 1 ...]
13     terms iota 1 + 2 % 2 * 1 -
14
15     # Multiply both vectors and reduce the result by '+'
16     * '+ reduce
17 ;
```

mc_cos.5

The following program implements a form of the sieve of Eratosthenes which is quite popular in the APL community. The basic ideas for generating a list of primes between 2 and a given value `n` are these:

- Generate a vector `[1, 2, 3, ..., n]`.
- Drop the first vector element yielding `[2, 3, 4, ..., n]`.
- Compute the outer product of two such vectors yielding a matrix like this:

$$\begin{pmatrix} 4 & 6 & 8 & 10 & \dots \\ 6 & 9 & 12 & 15 & \dots \\ 8 & 12 & 16 & 20 & \dots \\ 10 & 15 & 20 & 25 & \dots \\ \vdots & \vdots & \vdots & \vdots & \ddots \end{pmatrix}$$

- Obviously this matrix contains everything but prime numbers, so the next step is to determine which number contained in the original vector `[2, 3, ..., n]` is *not* contained in this matrix which can be done using the set operation `in`.
- The result of `in` is a vector with `n-1` elements each being `0` (its corresponding vector element was not found in matrix and is thus not prime) or `1`.
- After inverting this binary vector it can be used to select all prime numbers from the initial vector `[2, 3, ..., n]`.

This is accomplished by the following **5**-program:

Example 10 – list of primes

primes.5

```
1  : prime_list
2    iota 1 +
3    0 remove
4    dup dup dup
5    '* outer
6    swap in not
7    select
8  ;
9
10 100 prime_list .
```

primes.5

This program yields the following output:

```
[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 ]
```

The following example shows how functions can be plotted on a dumb ASCII terminal by employing `reshape` to generate vectors consisting of blank characters which are then used for indentation.

To plot a simple sine curve the following code could be used:

Example 11 – ASCII plotting

sine_curve.5

```
1 : print_dot{u}
2   " " 1 compress swap reshape "*\n" append "" join .
3 ;
4
5 21 iota 10 / 3.14159265 * sin 20 * 25 + int
```

sine_curve.5

This yields the following output:

ASCII plotting

```
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 $\begin{pmatrix} 10 \\ 11 \\ 12 \end{pmatrix}$:

Example 12 – matrix vector multiplication

matrix_vector.5

```
: inner+{u} '+ reduce ;

: mv* 1 compress '* apply 'inner+ apply ;

9 iota 1 + [3 3] reshape
3 iota 10 +

mv* .
```

matrix_vector.5

The code example above yields the following:

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

Quoting Wikipedia, a perfect number is "*is a positive integer that is the sum of its proper positive divisors, that is, the sum of the positive divisors excluding the number itself.*" Given the task to determine all perfect numbers between 1 and some arbitrary number, say 500, this can be accomplished using **5** quite easily:

Example 13 – perfect numbers between 1 and 500

perfect.5
```
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

Implementation

The following slides give a short overview of the implementation of **5**.

The interpreter is completely based on Perl which not only made it possible to achieve a remarkable short time to market but also simplifies porting the interpreter to different target systems (normally all that must be taken care of are file system specifics).

Executing a **5**-program is done like this – the main interpreter routine, called `execute`, implements a finite state machine:

Executing 5

- A **5**-program is parsed by basically splitting its source on whitespace with some special provision for arrays. The basic entities of a **5**-program are called *tokens*.
- Word definitions start with `:` and end with `;`. The start of a word definition has highest priority for the interpreter.
- If no word is to be defined, **5** tries to execute a built in word of the name found in the current token.
- If no matching word can be found **5** tries to execute a user defined word named like the token read.
- If no word is found, **5** checks if there is a variable with a matching name – if it succeeds the contents of that variable will just be pushed onto the stack.
- If even this did not work the element just read will be pushed onto the stack.

If there were no nested structures and no `if-else`-constructions or loops, scanning and parsing **5** would be really trivial.

To handle nested data structures like `[1 [2 3] 4]` special treatment of the tokens generated by splitting the source code at whitespaces is needed since these tokens would look like `[1`, `[2`, `3]` and `4]` which does not represent what the programmer intended.

Therefore the raw program representation is subjected to a special step which gathers data of nested data structures and transforms the example given above back to `[1 [2 3] 4]` from the token stream.

The same holds true for nested program structures like `if–else–then`- and `do–loop`-structures which are processed similarly and yield a nested program structure for every controlled block.

Thus a **5**-program is represented within the interpreter as a nested array containing

- an entry for every word to be executed,
- an entry for each scalar used in the program,
- an entry containing a reference to a nested array structure for every such structure found in the **5** source code,
- an entry containing a reference to a nested structure for every `if-else-then` or `do-loop` controlled block.

The following example shows a simple **5**-program consisting of two nested loops and its internal representation in the interpreter.

Consider the fibonacci example using a unary UDW. The internal represenation of this program looks like this:

```
[
  ':', 'fib{u}',
    'dup', '2', '<', 'if',
    [
      'drop', '1', 'break'
    ],
    'dup', '1', '-', 'fib', 'swap', '2', '-', 'fib', '+',
  ';',

  '10', 'iota', 'fib', '.'
]
```

Language elements

Currently the following functions and operators are implemented:

Unary operators: ?, cos, defined, int, neg, not, sin, sqrt

Binary operators: +, -, *, /, %, **, &, |, $\wedge$, ==, !=, >, <, >=, <=, eq, ne, gt, lt, ge, le, <=>, cmp, ||, &&, and, or

Stack operations: clear, depth, drop, dup, over, _roll, swap

Array operations: apply, collapse, compress, copy, expand, grade, in, iota, join, length, outer, reduce, remove, reshape, reverse, select, shape, split, subscript

IO operations: ., .., close, eof, fin, fout, load, open, read, unlink

Variable and word handling operations: .ofw, del, eval, set, vlist, wlist

Control structures: break, do, else, execute, exit, if, loop, panic, system, then

Miscellaneous: help, type, ver

5 can be readily extended by defining user defined words. At startup, the interpreter looks for a directory named `lib` and loads every file with a file extension `.5`. The distribution kit already contains the libraries `stdlib.5` and `mathlib.5` which contain the following UDWs:

- `stdlib.5`: `.s`, `.v`, `2dup`, `append`, `clear`, `dreduce`, `explain`, `extract`, `ndrop`, `pick`, `roll`, `rot`, `save`, `slurp`
- `mathlib.5`: `!`, `abs`, `amean`, `cmean`, `distinct`, `gmean`, `gplot`, `hmean`, `hoelder`, `inner+`, `median`, `prime`, `qmean`, `subset`

Conclusion

Conclusion

- **5** is a powerful tool and can be used for a wide range of (mostly mathematical) applications, running from rapid prototyping to ad hoc data analysis and the like.
- **5** is currently running on various UNIX systems as well as on OpenVMS and Windows.
- If you are interested in using or even extending the **5**-interpreter do not hesitate to contact the author at `ulmann@vaxman.de`
- The author would like to thank Mr. Thomas Kratz who wrote most of the current incarnation of the **5**-interpreter.
- The author would also like to thank Dr. Reinhard Steffens for his support and proof reading.

Bibliography

- [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
- [Brodie 04] Leo Brodie, *Thinking Forth – A Language and Philosophy for Solving Problems*, 2004nframe
- [Conklin 07] Edward K. Conklin, Elizabeth D. Rather, *Forth Programmer's Handbook*, FORTH, Inc., 2007
- [Giloi 77] Wolfgang K. Giloi, *Programmieren in APL*, deGruyter, Berlin, 1977
- [Katzan 70] Harry Katzan Jr., *APL Programming and Computer Techniques*, Van Nostrand Reinhold Company, 1970