5

Dr. Bernd Ulmann

10-NOV-2009

Raven Information Technologies GmbH

Introduction

The following talk is about a new (rather eclectic) interpretive programming language called **5** which is, in essence, the result of combining the basic ideas and aspects of Forth and of APL.

The idea of extending Forth is not new to say the least – the most prominent example being HP's well known programming language RPL which was for many years the work horse of their top-of-the-line pocket calculators like the HP-28, the HP-48 etc.

While RPL, short for *Reverse Polish LISP*, combines Forth's stack approach to programming with typical list processing operations normally to be found in the language LISP, the language described briefly in the following is also based on Forth but extends it with the programming paradigm of APL, Ken Iverson's brain child which is still unmatched when it comes to elegance and compactness of code.

Why 5

Why would one create yet another programming language? Aren't there more than enough already? The main reasons for the development described in the following 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.

What are the main characteristics of **5**?

- **5** is completely stack based, its builtin operations are called *built in words* while user defined operations are just called *words*.
- The stack can hold entities of arbitrary structure as long as these may be represented as nested arrays.
- All built in words as well as user defined words operate on those arbitrary structures which can be placed on the stack. Thus `2 3 + .` yields `5` while `[1 2] [3 4] + .` yields `[4 6]`.
- During startup the **5**-interpreter looks for a file named `stdlib.5` – if one is present it will be loaded prior to loading the program to be executed. This standard library contains **5**-extensions written in **5** (for example the words `grot` and `ggrot`).

The implementation of 5

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

This implementation is neither complete nor stable enough to be called production grade – its main purpose is to serve as a proof of concept of the language itself and its basic implementation concepts.

The current **5**-implementation is based on Perl which led to a very rapid development of the interpreter which took, until now, only about two man weeks.

A typical **5**-program could look like this:

```
4 iota dup '* outer
dup 2 compress .
```

Executing 5

Such a **5**-program is rather simple to scan, parse and execute:

- 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 to be pushed onto the stack 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.

Example structure

```
0 do
    100 do
        dup .
        1 +
        dup 105 > if
            break
        then
    loop
    drop
#
    dup .
    1 +
    dup 5 > if
        break
    then
loop
drop
```

Example structure

The preceding program is then represented internally like this:

```
[ '0', 'do',
  [ '100', 'do',
    [ 'dup', '.', '1', '+', 'dup', '105', '>', 'if',
      [ 'break' ]
    ],
    'drop', 'dup', '.', '1', '+', 'dup', '5', '>', 'if',
    [ 'break' ]
  ],
  'drop'
]
```

Currently the following words are implemented:

Binary built in words: `+ - * / & | ^ > < == >= <= != <=>`
`% ** eq ne gt lt ge le`

Unary built in words: `not neg ! sin cos ? int`

Stack operations: `dup drop swap over rot depth`

Array operations: `iota reduce remove outer in select`
`expand compress reverse`

IO-operations: `. .s .v read`

Variable operations: `set del`

Control operations: `if else then do loop break exit`

Examples

Although the **5**-interpreter is far from being complete, some more or less simple and actually working examples can already be shown:

These examples include

- Some introductory programs,
- cosine approximation using MacLaurin series and
- the sieve of Eratosthenes.

Recursion is such a powerful tool that not only the **5**-interpreter itself is highly recursive internally but the language **5** itself allows recursion as well.

The following program computes the well known Fibonacci number sequence implementing the recursive definition

$$\begin{aligned} f(0) &= 1 \\ f(1) &= 1 \\ f(n) &= f(n-1) + f(n-2). \end{aligned}$$

Fibonacci numbers

```
: fib
    dup 2 <
    if
        drop 1
    else
        dup
        1 - fib
        swap 2 - fib
        +
    then
;

0 do
    dup fib . 1 +
    dup 10 > if break then
loop
```

Throwing dice

Throw a dice 100 times and calculate the arithmetic mean of the results:

```
: throw_dice
    100 dup iota undef ne 6 *
    ? int 1 +
    '+ reduce swap / .
;
```

Cosine approximation

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`:

Cosine approximation

```
: mc_cos
    # Save x and the number of terms for future use
    'x set 9 'terms set

    # Generate a vector containing x ** (2 * i)
    terms iota dup undef ne x * swap 2 * dup v2i set **

    # Generate the (2 * i)! vector and
    # divide the previous vector by this
    v2i ! /

    # Generate a vector of the form [1 -1 1 -1 1 ...]
    terms iota 1 + 2 % 2 * 1 -

    # Multiply both vectors and reduce the result by '+'
    * '+ reduce
;
```

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:

```
: prime_list
  iota 1 +
  0 remove
  dup dup dup
  '* outer
  swap in not
  select
;

100 prime_list .
```

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 ]
```

Combining the power of Forth and APL, **5** requires a consistent programming style and rational factoring of words to ensure code maintainability.

The cosine example from above could have been written also like this:

```
: mc_cos
    'x set 9 'terms set
    terms iota dup undef ne x * swap 2 * dup v2i set
    ** v2i ! / terms iota 1 + 2 % 2 * 1 - * '+ reduce
;
```

This code is not really what one would call maintainable compared with the far better formatting and commenting shown in the original example.

All in all the following topics should be taken into account when programming in **5**:

- Use short word definitions.
- Words should do only one thing.
- Words should have no side effects.
- Indentation of control and data structures is vital for readability.
- Resist the temptation of using really clever programming trickery! :-) (It is hard, but...)

- Next steps: Add more (complex) words like rho etc.
- The source code of the **5**-interpreter is available upon request and it is planned to setup a Source Forge project for **5**.
- The power of Perl for implementing interpreters and the like is remarkable – the complete **5**-interpreter currently consists of only about 700 lines of code.
- I would like to thank Mr. Thomas Kratz for the many hours of peer programming during the implementation of the current **5**-interpreter.
- The author can be reached at

ulmann@vaxman.de
ulmann@raven-infotech.de