OpenVMS Technical Journal V15

5 – A Powerful Array Language

Dr. Bernd Ulmann
Hochschule fuer Oekonomie und Management, Frankfurt

Introduction

VAX APL and then?

Do you remember VAX APL? Wasn't it great to have something as powerful as APL at your fingertips on a VMS system (back then without the "Open")? Many years ago I used VAX APL extensively while studying mathematics. What I loved most about APL was its power and expressiveness -- which came at a price, though. APL programs, having been written in an interpreted language, were never as fast as programs written in FORTRAN or C. However, by using APL, my time-to-market was faster by at least one or two orders of magnitude compared with other, more traditional languages.

VAX APL is long since gone and with it much of the fun and productivity when it comes to exploring the behavior of some mathematical function or the quick implementation of a newly discovered algorithm.

Although there are quite a few APL interpreters available for various operating systems, most are commercial by nature, apart from A+[1]. Additionally, not one of them is available on OpenVMS systems, which is unfortunate, especially for me since I still prefer to do most of my daily work on an OpenVMS system (namely, FAFNER, a VAX-7000/820).

Expressiveness and complexity

What makes APL so powerful is its approach to handle deeply nested data in a transparent and elegant way. Adding two vectors in APL is as easy as writing `A+B`. The binary function `'+'` is extended on an item-by-item basis if it is applied to non- scalar values, so it will be applied to every two corresponding elements of the two vectors `A` and `B`. But APL goes further with complex functions and operators, which all apply naturally on nested data structures. In many cases, these can eliminate the need for explicit loops or conditionals in APL programs.

This expressiveness of APL comes at a price, too. APL makes use of a rather strange character set, which required terminals with loadable fonts or X-terminals in the times of VAX APL (as well as a keyboard with additional labels -- or the very good memory of the programmer).

What makes things even more arcane for the non-initiated programer is the fact that APL evaluates arithmetic expressions from right to left. This is mathematically elegant since it makes things like the evaluation of polynomials easy without cluttering the expression with lots of parentheses. But from a practical point of view, with the occasional APL programmer in mind, this was and still is a constant source of errors until one really gets adapted to APL.

Combining APL and Forth

Allowing for a short digression from the main theme of "array languages", let us explore a related topic. Many of you have worked with HP pocket calculators featuring RPL(Reverse Polish LISP) as their programming language, as I have. RPL is an interesting concept, as it combines the main features of two well-known languages, namely Forth and LISP. RPL extends the rather simple Forth stack with a stack that can hold nearly arbitrary objects, and allows the execution of LISP-like functions operating on these structures found on the stack. This works since nested data structures are represented as nested lists, which are readily accessible to a LISP-style approach of programming.

Some time ago I began thinking about how a language would look and feel if it combined the features of Forth and APL, two of my favorite programming languages. After some experimentation, with paper and pencil at first, it became clear that such an approach could very well work. Namely, by combining the powerful functions and operators of APL with the bottom-up programming style of

[1] Cf. `http://www.aplusdev.org`.

Forth, the result would encompass the expressiveness of both languages and the simplicity of the resulting parser in a very powerful blend.

Realizing this, the development of such a language, which has been named **5**, was started, using Perl as the basis for the 5 interpreter. A first implementation was done in a couple of weeks during spare evenings and weekends, and served as a test bed to explore the possibilities of such a stack-based array language. The findings of this first iteration were then used as the foundation for a second implementation, which was done from scratch[2]. It is this second implementation that will be described in the following sections.

It was decided at the very beginning of the development of 5 to abandon the complex character set of APL in favor of easily remembered mnemonics for all functions and operators to be implemented.

Since the 5 interpreter is written in pure Perl and does not require any non-standard packages, it can be run on any system which has a Perl interpreter installed. The 5 interpreter has been written explicitly with the goal of easy portability – and specifically with OpenVMS in mind as a target system. It runs out-of-the-box on any OpenVMS system that has a Perl interpreter installed.

The complete 5 project is hosted by sourceforge. The homepage for this project can be found at http://lang5.sourceforge.net/. To get started with 5 on any reasonable platform, including OpenVMS, you will need the distribution package, which can be downloaded from https://sourceforge.net/projects/lang5/files/ as a ZIP-file (about 270 KB). This file contains the 5-interpreter, some examples, and some PDF documents containing an introduction to programming in 5. To install the interpreter on an OpenVMS system, extract the ZIP-file into a suitable subdirectory. To use 5, all you need to do is to define a foreign command, such as[3]:

```
$ FIVE :== "PERL DISK$SOFTWARE:[5]5"
```

To start the interpreter in its interactive mode, just type `five` on the DCL prompt:

```
ULMANN:FAFNER$ five
----> loading mathlib.5
----> loading stdlib.5
5> "Hello world!\n" .
Hello world!
5>
```

Programming 5

Basics

At the beginning of the VAX APL Reference Manual, it says:

> "This manual is not a tutorial and is inappropriate for novice users. Programmers experienced with other languages such as FORTRAN or BASIC can learn VAX APL from this manual, but are advised to study it in conjunction with an APL language primer."

The same holds true for the following sections. 5 is far too complex to be described in a few pages, but it is possible to give some examples illustrating the expressiveness of 5 and the ease by which even complex algorithms may be implemented. A much more complete description of the language, together with introductory examples, can be found in the PDF files of the distribution kit, which has already been mentioned.

[2] The author would like to thank Mr. Thomas Kratz who did most of this second implementation.

[3] This, of course, assumes that the interpreter resides in a directory called `[5]` located on the disk `DISK$SOFTWARE` – adapt this definition to suit your local situation.

The look and feel of 5 is quite like Forth at first sight, since there is a central stack holding all data to which functions and operators will be applied. The parser is thus rather simple since all operators and functions only act on the stack so that no precedence rules – parentheses, etc., must be taken care of. In fact, the parser just splits the input read from `STDIN` or from a file on whitespace characters and either executes the tokens found if these are valid operators, functions, or words, or pushes raw data onto the stack. Thus, to compute the result of (1+2)*3 using 5, one just has to enter this expression in postfix notation at the prompt of the interpreter:

```
5> 3 2 1 + * .
9
5>
```

The single dot at the end of the command line prints the element found on the top of stack, which will be referred to as *TOS* in the following sections. Now, how does this example work? First, the interpreter has pushed the three scalar values `3`, `2`, and `1` onto the stack. Following this, the binary operator '`+`' has been applied, which in effect has removed the two top-most elements from the stack (or `1` and `2` respectively), and placed the resulting sum on the TOS, which now contains the scalar value `3`. The next step then calculates the product of `3` and this sum and places the result, `9`, onto the stack, which is then printed by using the dot function.

As in APL, the unary and binary operators as they are referred to in 5, normally act on simple scalar values. If such operators are applied to nested structures, they will, in effect, be applied in an element-wise fashion to all elements of the affected structure(s), as the following example shows:

```
5> [1 2 3] 2 * .
[    2     4     6  ]
5> [1 2 3] [4 5 6] + .
[    5     7     9  ]
5>
```

This is where 5 departs from a traditional Forth interpreter, as its stack can hold arbitrarily structured data instead of only simple scalars. In the first example shown above, every element of a three-element vector is multiplied by 2, while in the second example, two three-element vectors are added element by element.

User-defined words and more complex array operations

Another feature that 5 borrows from Forth is the possibility to extend the language itself by introducing user-defined words, which can be used in exactly the same way as the built-in functions and operators. The following example shows the definition of a word that returns the square of a value:

```
5> : square dup * ;
5> 5 square .
25
5>
```

As one can easily see, any word definition that starts with a colon followed by the name of the word can be defined. The end of such a definition, which in effect is just a list of 5 operators, functions, other words or operands, is denoted by a semicolon. The word `square` defined above will duplicate the element found on TOS and multiply the resulting two elements. Thus, applying this newly defined word `square` to the value 5 on the TOS yields the result 25 on the TOS.

Let us have a look at this slightly more complex example[4] that simulates throwing a six-sided dice 100 times and returns the arithmetic mean of the results:

[4] Note that word definitions which are made in interactive mode have to fit in a single line! This example is taken from a file which was executed by the 5 interpreter.

```
: throw_dice
  6 over reshape
  ? int 1 +
  '+ reduce swap /
;

100 throw_dice .
```

What does this example illustrate? First of all a new word, throw_dice, is defined. This word places the value 6 onto the TOS and copies the number of runs, which is now on the element just below the TOS, using the over function to the TOS again. Assuming that 100 runs are to be performed, the stack now contains the values 100, 6, and 100. The two last values are then used as arguments for the reshape function, which yields a vector containing 100 elements, each having the value 6 on the TOS: [6 6 … 6].

This vector is then used as argument for the ? operator, which generates a pseudo-random number between 0 (inclusively) and the element to which it is applied (exclusively). Since the ? operator is a unary operator, it will be applied to all the elements of a nested data structure automatically by the 5 interpreter. This will result in a vector containing pseudo-random numbers between 0 and 6 being placed on the TOS. Applying the int-operator will remove the fractional part of the elements of this vector; adding 1 will yield a vector with elements ranging randomly between 1 and 6.

In the following step, something tricky happens. A multiplication operator is pushed onto the stack, which is done by preceding it with a single quote to prevent the interpreter from executing it immediately. This operator and the vector described above are then used as arguments to the reduce-function, which acts exactly like the reduce operator in APL. It applies the operator found on TOS between every two elements of the vector found on the element just below TOS. In effect, this calculates the sum of all elements of the vector.

Since we duplicated the value found on the TOS in the very first step of the user-defined word, which corresponded to the number of times we should throw the dice, there is still a copy of this value found in the element just below TOS. Using the swap-function, this value and the value of the sum calculated above are interchanged, so that applying the /-operator will yield the desired arithmetic mean.

A more complex example – calculating primes

The next example is a bit more complex than the dice simulation shown above. The goal is to generate a list of prime numbers between 2 and a number given on the TOS. Instead of applying test divisions by odd integers or the like, a more APL-like approach has been chosen.

Imagine that a list of primes between 2 and 100 is to be calculated. This information will be used to form a vector [2 3 4 … 100]. Two of these vectors (copies are made using the dup-function) are then used to create a matrix by computing an outer vector product. This matrix contains 99 times 99 elements, none of which is a prime since all the elements in this matrix are the result of at least one multiplication. In the next step, the original vector and this matrix are used as arguments to the set operation in, which yields a vector containing 99 elements being 0 or 1. A 0 denotes an element of the original vector that was not found in the matrix, while 1 represents the opposite case. Clearly, this vector contains a 0 at every location, where a prime number was found in the original vector. Inverting this vector will yield a selection vector containing the value 1 at every location, corresponding to a prime element in the original vector. This selection vector is then applied to the original vector using the select-function, which yields all prime numbers between 2 and 100.

The corresponding user-defined word in 5 looks like this:

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

Entering `100 prime_list .` will yield:

```
[    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  ]
```

as a result.

At first glance, this program does not look very intuitive to someone who is used to programming in traditional imperative or OO languages like C, Fortran, etc. However, one can quickly become accustomed to the idea of array languages in general, and 5 in particular, since the interactive nature of the 5-interpreter makes experiments easy and ensures that it can be learned quickly.

It is noteworthy that things like `if-else` constructions, recursion or loops, which are all supported by 5, are used only rarely when an array-oriented approach to programming is employed!

Even more complex – multiplying a matrix by a vector

To illustrate the basic idea of this more or less loop-free programming style, let us have a look at another example program, which implements a matrix-vector multiplication word in 5:

```
5> : inner+{u} '+ reduce ;
5> : mv* 1 compress '* apply 'inner+ apply ;
5> [[1 2 3][4 5 6][7 8 9]] [10 11 12]
5> mv* .
[   68   167   266  ]
```

The main word is mv* (short for matrix-vector-multiplication), which expects a vector on TOS and a matrix in the element below TOS. As an initial step, the vector found in TOS is enclosed in another vector yielding a vector of the form `[[ … ]]`. Following this, the operator * is pushed onto the stack and then applied in an element-wise fashion along the first axis of the matrix and vector found in the two top-most stack elements by means of the `apply`-function. This yields a vector with all partial products of the desired matrix-vector product. Applying a special user-defined word `inner+` will then reduce these partial products by adding them together. Note that this word definition differs from a simple user-defined word as described in the preceding sections by specifying `{u}` after the name of the word to be defined. This tells the interpreter that the word to be defined will be a unary word, which will be handled in exactly the same way as a built-in unary operator. Otherwise, it would not have been possible to apply this word to all elements of the resulting structure along its first axis using the `apply` function.

Conclusion

5 is a rather powerful array language and supports a highly interactive style of programming due to the features inherited from Forth. Since the basic data structures that 5 operates on are nested arrays, many problems can be solved without the need for explicit loops or complex conditionals, as the preceding examples have shown. Since I have 5 running on my OpenVMS system, I do not miss VAX APL any longer since 5 brings nearly all of the power of APL to my system without the need for costly third-party software, etc.

Nevertheless, 5 is still a work in progress. Although the set of operators and functions which are implemented at the time of writing this journal is rather large, there may be problems that may require extensions to the language. In many cases, these extensions can be incorporated in the standard library `stdlib.5` or the mathematical library `mathlib.5,` which are both part of the 5 distribution kit. If this is not feasible for a particular problem, it is rather easy to extend the interpreter itself.

If you like what you have been introduced to in this paper and are interested in learning more, please feel cordially invited to participate in the development efforts of 5. The next steps of the development team will involve the addition of new words in the libraries, enhancing the documentation (a quick reference guide is currently being written, in addition to the rather complete introductory documents), and the development of a stable regression test suite to facilitate further developments, etc.

For more information

Contact the author at ulmann@vaxman.de.