ELI: a simple system for array programming

by Hanfeng Chen and Wai-Mee Ching
Dept. of Computer Science, Queens College, City University of New York, Flushing, NY 11367
Dept. of Computer Science, Zhejiang Normal University, Jinhua, Zhejiang, China, 321004 (retired)
wukefe@gmail.com, waimee_ching@yahoo.com

Abstract ELI is an interactive array-oriented programming language system based on APL. In addition to flat array operations of ISO APL, ELI features basic list for non-homogeneous data, temporal data, symbol type and control structures. By replacing each APL character with one or two ASCII characters, ELI retains APL's succinct and expressive way of doing array programming in a *dataflow style*. A scripting file facility helps a user to easily organize programs in a fashion similar to C with convenient input/output.

1. Introduction

Ken Iverson first developed the APL notation for teaching applied mathematics at Harvard; later APL became machine executable through the work of L. Breed and colleagues at IBM T.J. Watson Research Center. Iverson intends APL to be a *tool of thought* for communicating algorithmic ideas precisely (see his Turing Award Lecture [7]). This leads to two unique features of APL: i) arrays are first class citizens, ii) a comprehensive set of array operations as language primitives and a special character set enforcing an *one character one symbol* rule with uniform syntax. That results in succinct expressions and a *dataflow style* of programming, i.e. organizing tasks as chains of monadic or dyadic functions and operators acting on arrays. As a consequence, APL is remarkably productive and versatile; it has been used profitably in areas ranging from finance, actuarial, computer-aided design, logistics, manufacturing to that of research in physics, econometrics and biometrics ([8] has a recent example). In contrast, the array language MATLAB follows the FORTRAN style and its main application areas center on scientific computation and engineering.

We see three factors contributing to APL's decline despite its power to rapidly implement and deliver commercial applications: i) introduction of two nested array systems post-ISO APL fragmented APL community without an increase of markets; ii) its special character set and iii) no free version of APL is (J is both free and uses ASCII characters but it is more terse and difficult to learn than APL). The aim of ELI is to re-introduce APL-style programming to a wider audience by a free and simple array programming system which is easy to learn and convenient to use. The project started a decade ago [3], and reactivated recently (http://mpi.zjnu.edu.cn/eli for documents and executable). We hope ELI will attract new comers to APL family of languages who may later move on to commercial APL products.

ELI has all flat arrays operations specified in the ISO APL standard [1]; it does not have nested array feature of IBM APL2 and other APL vendors. However, ELI has lists and list operations to deal with irregular/non-homogeneous data. In addition, ELI has temporal data, symbol type and C-like control structures. In ELI, each APL character is replaced by one or two ASCII characters in a way that is intuitive and retains the succinctness of original symbolic representation of APL code. Finally, ELI has

scripting file facility to conveniently transfer data and code as well as to organize programs in a way similar to the use of `#include` in C, which is further helped by a short function definition form.

While MATLAB is popular in scientific/engineering applications, a simple array language of APL lineage offers unique advantage in programming complex systems such as database systems. ELI is easy to learn, has useful features beyond original APL and retains its original programming style. Being free, ELI can let more people appreciate the fact that simplicity of rules and notation in a programming language leads to greater programming productivity.

In section 2, we describe the basic features of ELI system and the symbol representation, in section 3, we explain the scripting file facility; details can be found in the Primer [4] on our web-site. We discuss future development plan for ELI before conclusion. ELI is written in C++ using Microsoft MFC, hence is first available on Windows, but it soon will be ported to Mac OS X and Linux. ELI is interpreted, but we already have a compiler written in ELI with some restrictions (exclusion of the execute function and the general use of lists). The compiler, translates ELI code into ANSI C code (with this in mind, ELI prohibits variable names to be reused for function names, nor a function name to be renamed as a variable). Once the compiler has compiled itself, it will be offered for general use. ELI can also be used as a tool for generating parallelized code for multi-core desktop as well as for other parallel machines similar to what have been done using APL [6]; the route to parallelism is not a parallel implementation of the interpreter but to automatically parallelize the translated C code by the compiler. We also intend to implement a simple column-based database system in ELI.

2. The Array Language ELI

ELI is based on APL1 as described in the ISO APL standard [1]; it replaces each APL character symbol by one or two ASCII characters. A sample is listed in the table below (for a complete list of ELI symbols see [4], which comes with the system) where the original APL symbol is in parentheses.

`#` (⍴)	*shape/reshape*	`#.` (⌹)	*matrix inverse/divide*
`?` (∈)	*where/member*	`?.` (?)	*roll/deal*
`!` (⍳)	*interval/index of*	`!.` (⍋⍒)	*execute/drop*
`^` (∧)	*count/and*	`^.` (↑)	*first/take*
`&` (∨)	*or*	`&.` (⍉)	*transpose*
`_.` (⌊)	*floor/min*	`<.` (⊤)	*enclose/encode*
`~.` (⌈)	*ceiling/max*	`>.` (⊥)	*grouping/decode*
`*` (×)	*signum/multiply*	`*.` (*)	*exponential/power*
`%` (÷)	*inverse/divide*	`%.` (⊛)	*natural log/log*
`|` (|)	*abs value/modulo*	`|.` (!)	*factorial/binomial*
`\.` (⍀)	*scan,expand along 1st axis*	`+.` (⍕)	*format*
`$` (⌽)	*reverse/rotate*	`$.` (⍉)	*reverse/rotate on 1st axis*
`>` (⍒>)	*grade down/greater*	`@` (○)	*circle function*
`<` (⍋<)	*grade up/less*	`->` (→)	*branch*
`<-` (←)	*assign*	`<=` (≤)	*less or equal*

An APL symbol usually represents either a ***monadic*** or a ***dyadic*** *primitive* function depending on whether it has one or two operands, and it is the same in ELI, the name(s) next to each symbol in the table above is the name of the monadic (and the dyadic) primitive function it represents. While ELI symbols lost the exquisite beauty of APL font, it retains the essential spirit of *one character one symbol* of APL notation. We see that the ELI symbols, all consist of special characters, are simple and suggestive, hence easy to remember and do not require a blank between symbols and names/data as in original APL. This is in contrast to the more elaborate multi-character symbol scheme of J and the introduction of word symbols in Q [2]. The current ELI symbol representation differs partly from that in [3], though it is conceived with the same pragmatic considerations: i) give frequently used primitives single character representation while less frequent ones with a second character '.', ii) for two-character symbols not ending in '.' the character pair should be self-suggestive.

The arithmetic, logical and relational primitives in APL/ELI are called *scalar* functions since they act on arrays by an extension of their action on each (pair of) scalar element(s) in that array(s). To take cube roots of a group of numbers, or take different roots of a single number (`%3` is `1%3`), we enter

```
    1 2 8 1000 81 125 *.%3
1 1.259921 2 10 4.3267487 5
    1024*.%1 2 3 5 10
1024 32 10.079368 4 2
     _1*.0.5
0j1
     *.0j1*@1
_1
```

For a dyadic scalar function *f*, the **`outer product`** of *f* is written as **.:***f* . For example, to see how 1000 dollars will grow under 3%, 5% and 8% of annual interests rates in 10 years, we do

```
    1000 *1.03 1.05 1.08 .:*. !10
1030  1060.9 1092.727 1125.5088 1159.2741 1194.0523 1229.8739 1266.7701 1304.7732 1343.9164
1050  1102.5 1157.625 1215.5063 1276.2816 1340.0956 1407.1004 1477.4554 1551.3282 1628.8946
1080  1166.4 1259.712 1360.489  1469.3281 1586.8743 1713.8243 1850.9302 1999.0046 2158.925
```

Please note that a line of code in ELI, as in APL, executes from right to left, one primitive or derived function at a time with equal precedence but respects parentheses.

For two dyadic scalar function *f* and *g*, *f:g* is the **`inner product`** of *f* and *g*. `+:*` (`+.×` in APL) is the well known matrix multiplication, but there are other equally useful inner products such as `^:=`. For a dyadic scalar function *f* the *reduce* operator produces a derived function *f/* and the *scan* operator produces *f*. To illustrate, `+/V` is the *sum* of `V` while `+\V` is the *partial sums* of `V`, and the *or scan* `&\B` is a vector derived from boolean vector `B` by turning all 1s once it encounters a 1. Now, suppose `emp_Div` is a 3000 by 2 character matrix indicating which division an employee belongs to where each row is one of the rows from the variable representing divisions

```
     Div<-4 2#'HQMKSLIT'
     Div
HQ
MK
SL
IT
```

To count employees in each division, we simply do

```
      +/.emp_Div ^:=&.Div
```

where `&.Div` is the transpose of `Div` and the result of inner product is a 3000 by 4 boolean matrix indicating the division an employee belongs; finally, +/. sums along the first axis gives a 4 elements vector showing the number of employees in each division.

A data of *symbol* type is entered with a back-tick ` followed by a character string (possibly empty); a character string which can form a symbol obeys the same rule as that governing a *name* for variables.

```
    #s3<-`abc `ddl `comp
3
    2 2#s3
`abc  `ddl
`comp `abc
```

There are 6 ***temporal*** *data types*. For *date* and *second*, we have the following examples:

```
     2012.10.15+!7
2012.10.16 2012.10.17 2012.10.18 2012.10.19 2012.10.20 2012.10.21 2012.10.22
     23:10:50+30
23:11:20
```

Eli provides ***lists*** to organize *non-homogeneous* data: a *list* is a group of *item*s, each of which can be a scalar or an array, separated by ';' and a list `L` can be assigned to a group of variables where the number of variables is equal to `#L`:

```
    #L<-(`abc `ddl `comp;1 2 3)
2
    L
<`abc `ddl `comp
<1 2 3
    L[1]
`abc `ddl `comp
    L[2]
1 2 3
```

```
     (s;n)<-L
     s
`abc `ddl `comp
     n
1 2 3
```

To enter a list of one item, we employ the monadic primitive function ***enclose*** `<.`:

```
     #L<-<.2 4#!12
1
     L
<1  2  3  4
 5  6  7  8
```

The ELI programming environment, follows that of APL, is called a *workspace*. After you installed ELI and click on the ELI icon, a window pops up with a line

```
      CLEAR WS
```

indicating a *clear workspace*, i.e. there is no user variable or defined function yet (on right most of the top bar there is a `Help` button, click on it to access the Primer [4] for a basic description of ELI); but it has default system variables such as `[]IO`, the *index origin*, which is set to 1 and can be changed to 0:

```
     !10
1 2 3 4 5 6 7 8 9 10
     []IO<-0
     !10
0 1 2 3 4 5 6 7 8 9
```

One is either in ***execution*** *mode*, i.e. executing an ELI expression, including assignments to variables as we have seen in previous examples; or in ***definition*** *mode* to define a user function. To switch to the definition mode, you type `@.` followed by the intended function head. A user *defined function* can have one or two arguments, or no argument, it can return a result or no result and all these are specified by the function head (sect. 3.1 in [4]). One can use a list to effectively input more than 2 arguments:

```
   `sales bk_load (sa;cu;it;am;py;dt;sp)
```

Once you finish writing your function, type a matching `@.` to get out of the editor and back to the execution mode of the interpreter. The function you just edited is available for testing, but it is not yet saved. To save, the workspace should already have a name, if not, then do

```
     )wsid ABC
     )save
```

Of course, variables will also be saved. A saved workspace can be loaded later by

```
)load ABC
```

ELI does not support the `)copy` command to copy in a workspace on top of an existing workspace. Instead, if a workspace has no suspension, it can be `)out` ..(see section below), and then bring in back by the command `)in ..` which differs from `)load` in that existing variables and functions previously existed would remain, but in case of conflict, those in conflict will be replaced by newer ones.

ELI has *control structures* quite similar to those in C (control structures are not prescribed in [1], and not implemented in current IBM APL2 but present in the products of other APL vendors). There are five *reserved words* in ELI for control structures

if else case for while

and simple statements can be grouped together by a pair of curly brackets. We illustrate their use with a recursive function `rprime` for finding primes up to `n`:

```
p<-rprime n;i
p<-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
if (n<=100) p<-(n>=p)/p
else {pl<-#p<-rprime _.n*.0.5
  b<-n#0
  for (i:1;pl) b<-b&n#(-p[i])^.1
  p<-p,1!.(~b)/!n
}
```

This shows how control structures improve program readability when there are alternate choices and irregular iteration while ELI boolean vector operations still entice a succinct dataflow style coding.

J (www.jsoftware.com) pushes APL to be more abstract in the direction of functional programming, but aims of ELI are fairly pragmatic: to make classical APL more accessible to general public and easier to mix with other application environments, and easy communication such as pasting code in e-mails. ELI also avoids the complexity of nested arrays in APL2 by providing lists for non-homogeneous data.

3. Scripting Files and Testing

While workspace is great for program development, as you can save not only clean code but partially debugged code as well, it takes quite lot of space (in fact, the old APL idea of workspace is what we call *IDE* for other conventional languages today). When your workspace contains no unresolved error, i.e. the *system indicator* `)SI` is empty, ELI (follows IBM APL2) let you output the content of your clean workspace to a ***transfer file*** by the system command `)out fnam`; and later the content of that file can be brought back by `)in fnam`. However, a transfer file must first come from some workspace. It is rather inconvenient to input a large amount of data or functions into an APL workspace. Hence, we

introduced the ***scripting file*** facility in ELI to directly bring data and functions, written in ordinary text files, into ELI system. We note that scripting file facility exists in another APL dialect, A+, developed at Morgan Stanley http://www.aplusdev.org, but there is no workspace facility in A+.

A scripting file can contain not only function definitions, as what you would write in the ELI editor in the definition mode of the ELI interpreter, but also values for variables. To do that you put

```
&ABC I 2 50 80
…
&
```

The first line is the variable name followed by its *type* (`B`:boolean; `I`:integer; `E`:floating point; `J`: complex number; `C`:character; `S`:symbol;`D`: datetime) and the *rank* and *shape* of that variable, then followed by the value of the variable in *ravel* order. To load in a scripting file `S`, type

```
)fload S
```

There is a companion command `)fcopy S` for copying in a script file which behaves similar to `)copy` of a workspace. You can even put in executable statements such as '`RPRIME 120`'(where `RPRIME` must be defined earlier) in a script file; that statement would then be executed at the time of loading/copying. Hence, we can say that scripting files provide a ***batch*** *mode execution* for ELI. A scripting file can also include other system commands such as `)fload …, )fcopy` … to bring in other scripting files.

For scripting files, ELI also has a *short-form* function definition facility for simple functions which do not access global variables as follows

```
{fnam: …}
```

where `fnam` is the name of a function; `z` is the result, or the last expression is the result, of the function, `x` is the right argument, and `y` is the left argument if present, and all other variables are local; comment lines must be outside of the function body `{..}`. There is a standard library `standard.esf` which includes the following functions:

```
{avg: (+/x)%_1^.#x}          //row-wise average of a numeric array
{gmean: (*/x)*.%#x}          //geometric mean of a numeric vector
{intersect: (y?x)/y}         //y intersects x, those in y which are from x
{less: (~y?x)/y}             //elements in vector y which are not in x
{xor:  2|y+x}                //exclusive or of boolean vectors y and x
{last: x[#x]}                //last element of a vector
{triml: (&\x~=' ')/x}        //trim leading blanks off a character vector
{trimr: $(&\r~=' ')/r<-$x}   //trim trailing blanks off a character vector
{stddev: ((+/(x-avg x)*.2)%#,x)*.05}   //standard deviation of vector x
{median: ((0.5*w[m]+w[m+1]),w[m<-_1+[]IO+~.0.5*#x])[[]IO+2|#w<-x[<x]]} //median
```

Following examples in this script file, one can easily extend the ELI language to have many built-in functions which may or may not present in other array languages such as MATLAB or R. Since a scripting file can contain `)fcopy` command lines in the beginning, the short function form can be utilized by users in an specific application area to group commonly used functions in a scripting file similar to the use of `#include` to bring in domain specific libraries in C. In addition to using scripting file for input/output, we also have provided a link between ELI and Excel.

For APL users who have legacy APL programs which do not use nested arrays, we offer an APL program to translate APL source codes into ELI scripting files. This, indeed, is our main tool for testing the ELI interpreter. The backbone of our testing procedure follows the two-pronged strategy of [5], i.e. unit test and a large suite of application programs translated from APL. In both cases, we utilized scripting files to automate the process of generating input suites as well as comparing output files for correctness. Yet testing an interpreter is more complicated than testing a compiler as it has to account for erratic ways a user types in codes, and that can only be tested by the usage of many users.

4. Conclusion

We have described an array-oriented programming system called ELI, which is derived from APL but uses ASCII characters. It remains to be simple, succinct with expressive power and encourages a dataflow style of programming as in APL. The addition of control structures aids to better present complex code; and the new scripting file facility provides convenient means of input/output and let a user organize programs similar to the use of `#include` in C.

We hope the easy availability of such an array programming system will let more people appreciate the fact that simplicity of rules and notation in a programming language leads to greater programming productivity. ELI can be used both as a tool for speedy implementation of highly complex applications as well as for rapid experimentation in building prototypes in search for an ideal design.

References

[1] International Organization for Standardization, ISO Draft Standard APL, ACM SIGAPL Quote Quad, vol.4, no.2, December, 1983.

[2] Jeffery Borror, q for Mortals, a tutorial in Q programming, Continuux LLC, New York, 2008.

[3] W.-M. Ching, The Design and Implementation of an APL dialect, ELI, APL Berlin 2000 Proc., Berlin, 2000, p69-76.

[4] W.-M. Ching, A Primer for ELI, a system for programming with arrays, preliminary version, 2011.

[5] W.-M. Ching and Alex Katz, The Testing of an APL Compiler, ACM SIGAPL Quote Quad, vol.20, no.1, 1993, p55-62.

[6] W.-M. Ching and Da Zheng, Automatic Parallelization of Array-oriented Programs for a Multi-core Machine, Int'l Jour. of Parallel Programming, vol. 40, no.5, 514-531, 2012.

[7] Ken. Iverson, Notation as a Tool of Thought, Comm. ACM, vol.23, no.8, 444-465, 1980.

[8] Lars Wentzel, CPAM, Array Structured Product Data at Volvo Cars, Conf. Proc., International Conf. on APL, Berlin, Germany, 2010, p199-208.