# Solving a simple data science task in ELI

Wai-Mee Ching
waimeeching@gmail.com

June 20, 2019

In [2], Jiahao Chen gives a detailed description on how a simple data science task (involving ratings of Uber drivers by users) is solved in programming languages R, MATLAB, APL and Julia. We would like to provide a solution to that task in ELI [1], a modern extension of the classical APL language as defined in [4] which we refer to as APL1. We'll call the APL language used in [2] as APL2. The main difference between ELI and APL2 is the way each extends a homogeneous rectangular array in APL1 to handle non-homogeneous or irregular data. In APL2, it is extended inward to a general array where cells in such an array no longer need to be scalars or homogeneous but can possibly themselves be arrays (this we've already seen in the code for APL in [2]). In ELI, it is extended upward to a list, a linear collection of items; each one can be a scalar, an array or another list. Both have the each operator **"f** which applies a function f to each item in a list or a general array.

We note that ELI uses one or two ASCII characters to represent a primitive function/operator (those provided by the language system) symbol which would be one APL character in APL1; and a function symbol can represent a monadic or a dyadic function, i.e. having a right argument only or having a left and a right argument. For example, a=b is the equality function but =v is the unique function while `a^b` is the and function but `^v` is the count function. For a complete list of such symbols and the primitive functions/operators them represent, together with associated APL characters, see the table in page 3 of [3]. Let us start coding in ELI. In the following, `<-` is the assignment symbol, `[]<-is` for display and `//` is for comment. Also as in APL1, lines entered for execution are indented and we only number executed lines (1-7) here:

```
      us<-381 1291 3992 193942 9493 381 3992 381 3992 193942
                                                  //[1] getting input1
      rs<-5 4 4 4 5 5 5 3 5 4                     //[2] getting input2
      []<-I<- >.us                                //[3]
<1 6 8
<2
<3 7 9
<4 10
<5
      rs[I]                                       //[4]
<5 5 3
<4
<4 5 5
<4 4
<5
      {avg:(+/x)%^x}                              //[5] define function avg
avg
      []<-ra<-avg"rs[I]                           //[6]
<4.333333333
```

```
<4
<4.666666667
<4
<5
      &.‘uid ‘rating:((=us);,.ra)                    //[7]
uid    rating
──────────────────
381      4.333333333
1291     4
3992     4.666666667
193942   4
9493     5
      &.‘uid ‘rating:((=us);,.avg"rs[>.us])   //[8] get the same output as above
```

The main splitting function is the primitive monadic function grouping (`>.`): for a vector v, `>.v` is a list of indices of v such that each item in the resulting list consists of indices of elements in v corresponding to unique values in v. Note that in our case, the result of applying the monadic primitive function unique = to variable us is shown in the first column of the output for line 7 above and the result of applying grouping to us is I which is displayed after line 3. A list is enclosed by a pair of parentheses and items are separated by a ';' when entered in code. When displayed each item of a list is preceded by a '<'; I is a length 5 list (the index origin []IO is 1 here which can be set to 0 if so desired). The first item of I are the indices in us where 381 occurred, and so on so forth for the other items. More example and detail specification of the grouping function is given in Sect. 2.2 of [4]. Since all elements of list I consist of indices of vector rs, then we can index rs by I, resulting in a corresponding list rs[I] of the same structure as that of I but with values in each item replaced by the corresponding indexed values of rs, the ratings given by each user-id. Thus we grouped ratings by unique user-ids as shown in the display after line 4. To each of these groups (items) we would like to get its average.

Line 5 above defines a short function (see sect. 1.9, [4]) named avg in ELI. In defining a short function x is assumed to be the right argument and y is assumed to be the left argument if present while the final expression is the return value of the short function. In our case, it is the sum (+/x) of x, divided (%) by of the count ^x of x, i.e. the number of elements in x. In line 6 this function avg is applied to each item in the list rs[I] by the each operator " yielding a result ra shown in the display following line 6. By applying the raze function ,. (see Sect.2.1, [4]) to ra , we get a vector ,.ra of the same values as that of ra. Lastly, line 7 is mainly for the purpose of displaying our final result in a table form. ‘a ‘b:(v1;v2) is a dictionary D whose domain is a pair of symbols (symbols such as ‘John ‘ibm ‘ap12 is a new data type in ELI which is not in APL1 or APL2 but quite handy in labeling columns) and whose range is a list of length 2 (see Sect.5.1, [3]); and in case v1 and v2 are of equal length we can turn this dictionary into a table

&.‘a ‘b:(v1;v2)

In our case in line 7, v1 is =us a vector of unique elements in us, and v2 is ,.ra. Put all together, we have line 8 which gives the same output as that from line 7.

To clean up, we see that only the four high-lighted lines above (line 1, 2, 5, 8) need to be executed in ELI to get the final result to our simple data science task. Compare with the solution using APL2 written in [2], only one extra function avg needs to be defined and its definition is quite understandable. Nothing is missing. Even though ELI code deployed more primitive functions such as unique, count and raze but they are quite intuitive, other than the grouping function which does the major work in this split-combine task. In our opinion, a list is easier to understand, to manipulate and to display than a general array since it is linear. By using one or two ASCII characters to replace an APL character in APL, ELI code loses certain intrinsic beauty while still maintaining the succinctness of code as in APL1. But this adoption provides other benefits in communication and code file handling.

We remark that the solution to this simple data science task in ELI is at the heart of building a native database management subsystem in ELI (the original idea comes from the Q language of www.kx.com which is widely used at Wall Street right now). Please see Chapter 5 Dictionaries and Tables and Chapter 6 Queries:esql in [3]. In contrast to APL1 and Q, ELI has complex numbers as in APL2 and MATLAB which is quite useful for scientific programming. Unlike APL1, ELI also has modern control structures. In addition, ELI provides a ELI-to-C compiler ecc which translates a ELI program to a corresponding C program for performance. But at present, the compiler can only compile ELI programs restricted to using features corresponding to those in APL1.

# References


[1] Hanfeng Chen and Wai-Mee Ching, ELI: a simple system for array programming, Vector, vol.26 No.1, 94-102, 2013.

[2] Jiahao Chen, Linguistic Relativity and Programming Languages, `https://arxiv.org/pdf/1808.03916.pdf`

[3] W.-M. Ching, A Primer for ELI, a system for programming with arrays, `http://fastarray.appspot.com`, 2013.

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