A SUCCINCT PROGRAMMING LANGUAGE WITH A NATIVE DATABASE COMPONENT

Hanfeng Chen[1] and Wai-Mee Ching[2]

[1]School of Computer Science, McGill University, Montreal, Canada
[2]201 Kensington Way, Mount Kisco, NY 10549, USA

ABSTRACT

ELI is a succinct interactive programming language system organized around a few simple principles. Its main data structures include arrays, lists, dictionaries and tables. In addition, it has an integrated database management component which is capable of processing a basic set of SQL statements. ELI, with a compiler, covering the array portion of the language, is also an excellent tool for coding scientific and engineering solutions. Moreover, it is productive in writing complex applications as well, such as compilers and trading systems.

1. INTRODUCTION

ELI is a succinct interactive programming language system with its heritage derived from the classical array-based programming language APL invented by Ken Iverson [1]. Its organizing principles, as those of APL, are: i) arrays are treated as first class citizens, ii) it provides a large number of symbolically represented intrinsic (primitive) functions with equal precedence which operate on arrays as a whole, iii) the symbolic representation of intrinsic function results in succinctness of code that encourages a *dataflow style* of programming, i.e. the result of one operation can feed as the input to the next operation in the same line. This greatly contributes to the clarity of code and overall productivity. ELI has most of the functionality of ISO APL [2] (we shall refer this from now on as APL1). The only difference of this core part of ELI and the APL1 is ELI's adoption of ASCII characters instead of the original APL characters [3]. ELI is available on Windows, Linux and Mac OS (http://fastarray.appspot.com/) and has both 32-bit and 64-bit versions. The 64-bit version comes with a GUI feature called ELI Studio. The homepage of the ELI site lists a group of one-line example codes to illustrate the style as well as the capability of the ELI programming language. The documentation section of the ELI site provides the main language reference [5] and two tutorials. One tutorial [6] aims to teach young students mathematics and computer programming together, utilizing the interactive ELI system, dove tails with Ken Iverson's original intension in inventing APL: to teach applied mathematics at Harvard.

Later APL evolved from *flat-arrays* (i.e. each array cell element is either a number or a character) to *nested arrays* where an array cell element can be another array as exemplified by the IBM APL2 programming language [4]. ELI, following the earlier ***k*** language which also traces its origin from classical APL (*k* later evolves into the Q language [7]) of ***kdb*** system (www.kx.com), does not have nested arrays; instead it has *lists* for handling non-rectangular or non-homogeneous data. In addition, ELI and Q have six temporal data types which APL2 does not have. We shall illustrate the usage of these temporal data types by two examples in the next section. However, APL2 as well as ELI have complex numbers while Q does not provide complex numbers;

complex numbers are quite important in many scientific and engineering applications (we note that Q's intended area of application is in financial analysis and algorithmic trading).

A *dictionary* in ELI and Q is a special kind of two items list, a domain and a range together with a *map* between the two. We note that there are also *dictionaries* in the programming language Python, and in Perl it is called *hashes*. The *transpose* of a *column dictionary* (to be explained in Sec. 3) in ELI is a *table* and this is the foundation of a native database component in ELI, following that of Q of the *kdb* system. In Sec. 4, we show that ELI has a set of SQL-like statements, ***esql***, to manipulate data in the tables of a database just like in other RDBM systems. The most important thing is that *esql* statements are implemented as ELI functions. In Sec. 5, we discuss the file processing facilities of ELI such like processing as well as writing CSV, XML and JSON files in addition to standard ELI scripting files.

We remark that both APL1 and APL2 have no *control structures* while ELI has *control structures* just like those in the C language (other modern APL system such as Dyalog APL has control structures and Q has *if*-statement). This makes it easier to write complex software such as compilers in ELI while still provides remarkable productivity due to the succinctness of code.

We shall discuss the features of an ELI compiler [8] (it only covers flat arrays) in Sec. 6 and its role in providing competitive performance in term of run-time. We also will make relevant comments on ELI in comparison with other programming languages relevant to software engineering in that next to last section. The last concluding section summarizes what we have presented.

2. TWO ILLUSTRATIVE EXAMPLES

In this section we use two examples involving temporal data to show the usage of some important ELI primitive functions and thus illustrate the ELI programming style. There are 12 ***atomic*** *data types* in ELI: ***numbers***, which includes *boolean*, *integer*, *floating point* and *complex number*, ***symbols***, ***characters*** and 6 ***temporal data*** types which includes *date*, *time*, *month*, *minute*, *second* and *datetime*. A single data item is called a *scalar* while a group of data items of the same type forms an array. Each *array* a has a *shape* (#a) and can be *reshaped* (sv#a). There is a system variable []IO which by default is 1 but can be reset to 0. The primitive function *interval* !n generates n integers:

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

ELI has a system variable *timestamp* []TS which reports the current *datetime* and the other temporal data types can be seen as a different parts of a *datetime* type data:

```
        []TS
2016.08.22T21:47:39.312
```

The portion before the letter T is the *date* while the portion after T is the *time* part of a *datetime* value and 21:47:39 is the *second* part of the current time.

Two distinctive features of ELI/APL are i) a line of code executes from ***right to left*** but respects parentheses, ii) primitive functions in ELI are either *monadic* (which takes a *right* argument) or *dyadic* (which takes a *right* and a *left* argument) and they all (including *defined functions*) have ***equal precedence***. These two rules greatly encourages a *dataflow style* of ELI/APL programming. Two consecutive dashes (//) start a *comment* to its right. Before we present the first example, we describe the *compress* function b/a where b must be a Boolean vector and the right operand a must be a vector of equal length or a matrix with its width equal to the length of b; the result is a vector of elements in a corresponding to 1s in b, or a matrix of columns in a corresponding to 1s in b:

```
      0 1 0 0 1 0 0 0 1 1/!10 //[]IO=1
2 5 9 10
```

The first example is to print out a weekly calendar of August 2016, knowing that 2016.07.30 is a Saturday. We proceed as follows:

```
5 7#2016.07.30+!5*7 //35 days after 7/30/2016 and reshape it into a 5 by 7 matrix of dates

2016.07.31 2016.08.01 2016.08.02 2016.08.03 2016.08.04 2016.08.05 2016.08.06
2016.08.07 2016.08.08 2016.08.09 2016.08.10 2016.08.11 2016.08.12 2016.08.13
2016.08.14 2016.08.15 2016.08.16 2016.08.17 2016.08.18 2016.08.19 2016.08.20
2016.08.21 2016.08.22 2016.08.23 2016.08.24 2016.08.25 2016.08.26 2016.08.27
2016.08.28 2016.08.29 2016.08.30 2016.08.31 2016.09.01 2016.09.02 2016.09.03

w<-+. 5 7#2016.07.30+!5*7 //format fn +. turns above into its character representation
#w //w is a 5 by 76 character matrix

5 76

b<-76#(6#0),5#1 //b is a boolean vector of length 76 with 6 0s followed by 5 1s

b

0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0
0 1 1 1 1
1 0 0 0 0 0 0 1 1 1 1 1 0 0 0 0 0 0 1 1 1 1

b/w //the '2016.0' parts have been eliminated from w

7.31 8.01 8.02 8.03 8.04 8.05 8.06
8.07 8.08 8.09 8.10 8.11 8.12 8.13
8.14 8.15 8.16 8.17 8.18 8.19 8.20
8.21 8.22 8.23 8.24 8.25 8.26 8.27
8.28 8.29 8.30 8.31 9.01 9.02 9.03

hd<-'Sun Mon Tue Wed Thu Fri Sat ' //prepare heading
```

```
#hd
34
#b/w
5 34
w1<-b/w
w1[1;!4]<-' ' //blank out 7.31
w1[5;20+!14]<-' ' //blank out Sept days
w1
8.01 8.02 8.03 8.04 8.05 8.06
8.07 8.08 8.09 8.10 8.11 8.12 8.13
8.14 8.15 8.16 8.17 8.18 8.19 8.20
8.21 8.22 8.23 8.24 8.25 8.26 8.27
8.28 8.29 8.30 8.31

hd,.b/w1 //glue the heading to w1 by the catinate along 1st axis ,. Func.
Sun Mon Tue Wed Thu Fri Sat

8.01 8.02 8.03 8.04 8.05 8.06
8.07 8.08 8.09 8.10 8.11 8.12 8.13
8.14 8.15 8.16 8.17 8.18 8.19 8.20
8.21 8.22 8.23 8.24 8.25 8.26 8.27
8.28 8.29 8.30 8.31
```

At the heart of the second example is the special primitive function *execute* !.tx which executes its operand tx as a text input to the ELI/APL interpreter. In the simplest case, !. can be regarded as the inverse of the *format* function +., i.e. it turns the character representation of a data item into a data which the operand character string represents:

```
!.'2.3'
2.3
!.'19:20:55'
19:20:55
!.'3+2'
5
```

We note here that current ELI compiler covers most features of ELI corresponding to APL1 with the *execute* function as the most notable exception. While this function can easily be avoided in scientific and engineering applications it is essential in implementing ELI's ***esql*** statements (see Sec. 4).

The second example is a daily scheduler: to run different tasks, i.e. calling various trading modeling functions written in ELI with appropriate parameters every 15 minutes from 9:00 to 21:00. The code is as follows where dt2S is a short-form function (see §4.2 in [5]) and tasktab is a 49-row text of possibly distinct tasks stored in the file jobs, i.e. function calls with specified parameters, to be executed at 15 minutes interval:

```
[]IO<-0
```

```
{dt2S: !._4!.11!.+.x}                        //convert a datetime to a second type
timelst<-09:00:00+15*60*!49
)fcopy jobs                                  //import jobs file which contains tasktab

j<-0
while ((j<49) { //execute a preset task every 15 minutes
while (timelst[j]>dt2S []TS) */1000#@1       //multiply pi 1000 times to spin
!.tasktab[j;] //when the time comes, execute the j-th job
j<-j+1
}
```

3. Dictionaries, Tables And Keyed Tables

ELI offers ***list*** for non-homogeneous or non-rectangular data. The *shape* (=*count*) of a *list* is its length, and an element in a list can be a scalar, an array or another list. For example,

```
      L1<-('ABC';`s1 `s2;3 4#1 5 6)
      L1
<ABC
<`s1 `s2
<1 5 6 1
 5 6 1 5
 6 1 5 6
     L2<-(1.2 11 3.5;4 8.1;!10)
     L2
<1.2 11 3.5
<4 8.1
<1 2 3 4 5 6 7 8 9 10
```

A most important operator on lists is the ***each*** operator ”: for a function f, f”L applies f to each element of list L. For example,

```
      #"L1
<3
<2
<3 4
      +/"L2
<15.7
<12.1
<55
```

where +/ is the derived function *sum*. The function *f* can be a user-defined function if it is well-defined on each elements of the list L.

A ***dictionary*** D is a two item list, a *domain* d and a *range* r of equal count and a correspondence between the two by the dyadic function ***map*** **:**, D<-d:r; a pair of system functions then return its components: key(D) gives the domain d and value(D) the range r. The domain d must be a simple list of unique elements such as a vector of symbols, characters or integers with no duplicates, the range r is a list of the same count as that of d whose items can be scalar, array or list of any type. For example,

```
    M<-1 10 20:('Washington';'Hamilton';'Jackson')
    M

1 |  Washington
10| Hamilton
20| Jackson

    M[10]

Hamilton

    M[1 20]

<Washington
<Jackson
```

Hence, a dictionary is a generalization of a list, instead of indexing by position values are indexed by keys in its domain. A dictionary whose domain is a vector of symbols and whose range is a list of equal length vectors (some of them can be a scalar or a character matrix with the number of rows matches that length) is called a ***column dictionary***. For example,

```
   D<-(`sym `price `hq:(`appl `ibm `hp `goog;449.1 108.2 24.5 890.3;4 2#'CANYCACA'))
  D

sym | appl ibm hp goog
price| 449.1 108.2 24.5 890.3
hq | 'CANYCACA'
```

The *transpose* of a column dictionary is called a ***table***. A *table* can also be directly defined by listing its columns:

```
  &.D
sym price hq
-------------
appl 449.1 CA
ibm 108.2 NY
hp 24.5 CA
goog 890.3 CA

     T<-([]sym<-`appl `ibm `hp `goog;price<-449.1 108.2 24.5 890.3;hq<-4 2#'CANYCACA')
    T
sym price hq
-------------
appl 449.1 CA
ibm 108.2 NY
hp 24.5 CA
```

```
goog 890.3 CA
```

Note that with ‘(’ followed immediately by ‘[’ it signifies the definition of a table so T is not a mere list but a table. Column values can be accessed by symbol indexing and updated:

```
        T[`price]
449.1 108.2 24.5 890.3
        T[`price]<-T[`price]+1 2 _3 0.5
        T
sym price hq
-------------
appl 450.1 CA
ibm 110.2 NY
hp 21.5 CA
goog 890.8 CA
```

The *shape* of a table is defined to be the number of *records* in it which is equal to the length of its column vectors. A group of columns in a table *t* is said to be the ***primary key*** of table *t* if values in that group uniquely define records in table *t*. Let *t*k be the (smaller) table composed of the primary key column(s) of *t* and *t*v defined to be the table with the rest of columns in *t*. Then a ***keyed table*** k*t* is defined to be a *mapping* k*t*:v*t* between these two tables. We can also enter a *keyed table* directly as a table by putting *key columns* inside []:

```
        kT<-([sym<-`appl `ibm `hp `goog] price<-449.1 108.2 24.5 890.3;hq<-4 2#'CANYCACA')
       kT
sym | price hq
----|---------
appl| 449.1 CA
ibm | 108.2 NY
hp | 24.5 CA
goog| 890.3 CA
```

To access a record or records in a keyed table we do indexing by value(s) from the table k*t* of keys (not by position):

```
       kT[`ibm]
price| 108.2
hq | 'NY'
       kT[(`appl;`goog)]
price hq
--------
449.1 C
890.3 A
```

Records in a keyed table just as in an ordinary table can be updated similar to indexed assignment (see **§5.3** [5]). For a variable u of unique symbols and a variable v with values belong to u, there is a construct called *enumeration* `u: of u over v and `u:v the enumerated values (see **§5.1** [3]). This can be extended to that of a table pT with a primary column as *domain* and entries in a column of another table *t*, whose values all belong to that of pT, as its *range*; this is denoted as `pT:. Hence, we define a ***foreign key*** *column* in a table as an enumerated value over a *t*.

```
        pT<-([cux<-1 3 5] sex<-'fmm';age<-35 18 51)
        pT

cux| sex age
---|--------
1 | f 35
3 | m 18
5 | m 51

        sals<-([] mech<-`p1`p2`p3`p4`p2;amt<-2.3 1.2 5 20.1 50.7;cust<-`pT:3 5 1 5 3)
        sals

mech amt cust
--------------
p1 2.3 3
p2 1.2 5
p3 5 1
p4 20.1 5
p2 50.7 3
```

A foreign key establishes a ***relation*** between the enumeration domain kt and the enumerated value table dt, i.e. it has a column fk<-`kt:…; then the values of a column c0 in kt can be accessed via fx with the dot notation dt.fk.c0 from dt, and this is called a virtual column.

```
        sals.cust.age
18 51 35 51 18
```

4. QUERIES: ESQL AND AN EXAMPLE DATABASE

ELI provides a basic set of *query* statements, ***esql***, which is quite similar to standard SQL statements for traditional relational database management systems. While not every SQL statement has a ready counterpart in *esql* yet, some *esql* statements can be more powerful than their corresponding SQL statements. One prepares an *esql* statement as a text string, and then executes that string by applying the *reserved* function do to it: do txt.

The ***create*** statement is of the following form:

CREATE TABLE tbl (fields;types;width)

where *tbl* is the name of the *empty* table to be created, *fields* is a list of column names separate by blanks, *types* is a character string indicating the types of column values and *width* is a vector indicating the width of character valued columns. For example,

```
      do stm10c<-'CREATE TABLE t1 (a b c d e f;"SICEIC";8 1)'
table t1 created.
      t1
```

```
a b c d e f
-----------
```

The ***load*** and ***insert*** statements are of the following form:

LOAD TABLE tbl (fld1s<-val1;…; fldn<-valn)

INSERT INTO tbl (val1;…; valn)

One can also load a table using the method of setting up a table in the previous section.

The ***select*** statement is of the most used query in any database system and it is of the following form:

SELECT [*fields***] [BY** *group***] FROM** *tbl* **[WHERE** *wherespe***]**

where each expression inside **[...]** is optional, *tbl* is the name of the table we are selecting data from; *fields* selects the columns of *tbl* while *wherespe* selects which rows of records in *tbl* to be included in the final result. The **WHERE** clause is processed first, if it is absent then all rows of *tbl* are selected; similarly, if *fields* is empty then all columns are selected corresponding to the statement **SELECT * FROM** *tbl* … in SQL. *wherespe* is of the following form:

constraint **[,***constraint1***[**, *constraint2,..* **]]**

See **§6** [5] for details of *fields*, *group* and *constraint*; the meaning of **BY** *group* clause is similar to that of **GROUP BY** in SQL.

We illustrate the use of *esql* by running query examples against a sample database of a *store* in the ws directory of *eli* distribution: after type)fload store2 in ELI session, we see the following:

```
customer<-([cux<-!tc] name<-na;sex<-se;age<-ag;addr<-ad;cardn<-cn)
stock<-([stx<-!ts] m_name<-mn;in_stk<-in;uni_prc<-up)
employee<-([empx<-!te] e_name<-en;e_age<-ea;e_phone<-ep)
sales<-([salx<-sa]cust<-`customer:cu;stk<-`stock:it;amount<-am;payment<-py;dat_time<-dt;
empl<-`employee:sp)
```

One can type do 'SELECT FROM sales' to see the content of the main table sales with salx as its primary key and cust,stk,empl as foreign keys pointing to tables customer,stock,employee respectively. We can do the following:

```
do stm2c<-'SELECT yng_sal<-amount, dat_time FROM sales WHERE 25 > cust.age'
```

```
yng_sal dat_time
------------------------------
7.1 2012.05.25T09:50:21.000
45 2012.05.25T10:37:03.000
23.5 2012.05.25T11:56:45.000
28.5 2012.05.25T12:00:22.000
51.2 2012.05.25T13:17:04.000
23.1 2012.05.25T13:23:41.000
```

```
26.5 2012.05.25T15:30:05.000
```

```
    do 'SELECT amount, dat_time FROM sales WHERE 50 > amount, stk.m_name IN `hammer
    `screw `tape'

amount dat_time
------------------------------
36.5 2012.05.25T10:13:42.000
28.5 2012.05.25T12:00:22.000
18.5 2012.05.25T13:43:25.000
30.2 2012.05.26T12:43:46.000
23.1 2012.05.25T13:23:41.000
39 2012.05.25T14:10:23.000
26.5 2012.05.25T15:30:05.000
48.2 2012.05.26T14:30:26.000
17.5 2012.05.25T15:33:42.000
```

```
    do stm10c<-'SELECT gender<-FIRST cust.sex, total<-SUM amount BY purchase<-cust.cux
    FROM

sales'
purchase| gender total
--------|-------------
0 | m 30.6
1 | f 138.3
2 | m 174.3
3 | f 398.4
4 | m 296.8
5 | f 253.1
6 | f 150.1
```

The ***exec*** statement in *esql* is of the same format as that of the *select* statement simply with the key word **EXEC** replacing **SELECT**. Instead of returning a table as in the case of the *select* statement, the *exec* statement returns a dictionary corresponding to the table for the alike select statement; for a one column table it just returns the column values. Through this statement, we can bring the powerful set of primitive functions and operators on array/list in ELI to the data involved for further processing and analysis outside the constraint of *esql*.

The ***update*** statement is also of the same format as that of the *select* statement but with *fields* being replaced by *updfields* which is a group of one or several assignments to column names:

UPDATE [*updfields***] [BY** *group***] FROM** *tbl* **[WHERE** *wherespe***]**

Tables can be saved and reloaded as part of a workspace, or they can be exported to an esf file (see §4 [5]). Finally we remark that *esql* statements are implemented as ELI defined functions but packaged by the ELI system to make them as system function calls instead of as ordinary function calls. In the ELI source code for various *esql* statements, the ELI primitive function *execute* !. played a crucial role. In addition, two monadic primitive functions *where* ?b and *grouping* >.m proved to be very handy in implementing corresponding *esql* statements of the similar name. We explain the first one *where* (?) here and refer the second one *grouping* (>.) to §2.2 [5]. For a Boolean vector b, ?b gives the positions of 1s in b depending on []IO:

```
     w
13 75 45 53 21 4 67 93 38 51
     w<50
1 0 1 0 1 1 0 0 1 0
     ?w<50          //[]IO=0
0 2 4 5 8
    w[?w<50]
13 45 21 4 38
```

5. File Handling Facilities In Eli

The basic work unit of the ELI system, as in APL, is called a ***workspace***. In **§3** of [3], we described the *scripting file* facility of ELI. In short, ordinary text file of type txt containing ELI data (with prescribed heading indicating variable name, type and shape information), defined function definitions and executable ELI statements can be *loaded* or *copied* into an ELI *workspace* by the ELI system command)fload or)fcopy as a *.esf file. Conversely, the content of an ELI *workspace* such as variables and/or defined function definitions can also be put out into an *.esf file. Hence, the tables of a ***database*** built in an ELI workspace are permanently *saved* as a *.esf file; subsequently, these saved tables in an ELI database can be loaded or copied into an active ELI workspace for data accessing, manipulation and analysis.

In this section, we briefly describe additional file facilities of ELI, which accepts several commonly used file formats as well writing the data context of a particular ELI workspace to files of these common file formats. ELI has a construct called ***file handle*** to support writing data to files and there are two system functions []open and []close to deal with file handles:

```
    h<-[]open 'file.txt'
    []close h
```

where the first statement opens the file file.txt and save the file handle to variable h and the second statement closes h, return 0 if h is successfully closed, return 1 otherwise.

The system function, []get, is used to read a raw text file as a whole while the system function, []CSV, is used to load a *csv* file into a workspace as a table. On the other hand, a table in ELI can be outputted to a file of certain common data formats like CSV, XML, JSON and TXT. Suppose contact is a table in an ELI workspace containing a database, The table *contact* can be saved into *contact.csv* by the system command []write. We can try three more commands to save the table into three different formats as follows:

```
    []write 'contact.xml'
contact.xml
    []write 'contact.json'
contact.json
    []write 'contact.txt'
contact.txt
```

More details on ELI file handling facilities can be found in **§4.3** of ELI Primer [5].

6. Discussion On ELI And Related Programming Languages And Systems

Since ELI inherits its core from APL, it is an array-oriented interpreter-based programming language system with obvious similarities to FORTRAN and MATLAB. FORTRAN, as the first high level programming language developed before APL, is a compiler based language system; later when FORTRAN evolved into the current version FORTRAN90, it borrowed many, but not all, array features from APL. FORTRAN is still actively used in scientific research, especially in HPC (high performance computing) circle since it has the best run-time performance and especially, FORTRAN is the only stable programming language tool available on many parallel supercomputers. MATLAB, on the other hand, is interpreter-based but its coding style follows that of FORTRAN; MATLAB is currently far more popular than FORTRAN. Naturally, both APL and MATLAB suffer greatly in run-time efficiency. To close that performance gap, in 1980s the second author developed the APL/370 compiler [9] (later it evolved into the APL-to-C compiler [10]) at IBM T.J. Watson Research Center which has been shipped to some selected IBM customers with reports of good performance [11]. Similarly, ELI offers an ELI-to-C compiler which covers language features corresponding to APL1 with the notable exclusion of the *execute* function (!.) whose argument value can only be known at run-time. MATLAB took a route quite different from the traditional compiler approach followed by [9], [10]: starting in 2004 it developed the JIT (*just-in-time*) compilation technology to seamlessly accelerate MATLAB applications in its interpreter based environment (especially those written in the scalar looping form) [12]. We have not yet made a comparison between the run-time performances of compiled code on a good sample of equivalent code in ELI and MATLAB under [8] and [12]. We intend to carry such a study soon.

Unlike FORTRAN and MATLAB, APL and ELI's application areas are not limited to scientific and engineering areas (we listed many of these in [3]). One good example to illustrate the versatility of APL/ELI is the fact that both APL compilers [9], [10] are written in APL and the ELI compiler [8] is written in ELI. One difference between the compilers in [10] and [8] is that the ELI compiler has been self-compiled; hence it operates independent of the ELI interpret environment, i.e. as a compiled C object code operates on a text file representing the ELI source code intended for compilation while in [8], [9] the compiler is a saved workspace (in this effort, the new scripting file facility of ELI played a crucial role). We hardly would image a compiler of such complexity being written in FORTRAN or MATLAB. We further note that the control structures in ELI, which is absent in APL1, greatly helps in writing complex software such as compilers. Thus ELI can be a suitable implementation language for complex software in many application areas.

Compared with C/C++, ELI is not as versatile since it is not a good system implementation language due to its lack of links to low level system calls as well as its poor run-time efficiency. Of course, ELI/APL is far more productive in term required manpower to complete an application of considerable complexity such as financial processing and logistics systems. This is largely due to its succinctness and the interactive nature of programming for easy testing and experimentation. Moreover, we note that ELI is not an *oo* (object-oriented) programming language such as C++, Java or Python. ELI utilizes a well-thought out group of high level primitives operating on arrays and lists in an integrated fashion to reduce program complexity and speed up program construction. An *oo* language uses *classes* to offer a modular approach to manage large program complexity. ELI offers conceptual simplicity while *oo* languages offer scalability in building large scale system software. This discussion brings us to the current most popular teaching language in computer science, Python. Python is also interpreter-based. It has array features in addition to being an *oo* language. Moreover, Python can be linked to many low level system features to build versatile systems. In comparison with ELI, Python greatly depends on abundance of many available packages. For example, to do scientific computation in Python, one imports the NumPy or SciPy module. In ELI these are already built into the ELI system itself. However, we are not aware of efforts to compile a substantial part of Python to increase its run-time efficiency.

The most important feature of the ELI programming language system is the existence of a native RDMS system as a group of related tables and a set of SQL-like statements, *esql*, applicable to these ELI tables just like SQL applied to tables in a RDMS system such as IBM DB2, Oracle and MySQL. We acknowledge the fact that we followed the direction of k/Q of *kdb* in this effort, and we also point out that k/Q are not mainly aimed financial/trading applications. The programming language *R* has *dataframes* similar to *tables* in ELI but not exactly the same and no corresponding SQL-like statements to manipulate them. Python has packages to do data analytics but none of these has true tables as in a relational database system with SQL-like statements. Many programming languages such as Python and Perl offer DBI package to interface with an existing database system such as MySQL to get data for further processing/analysis under the particular programming language interfaced to. In ELI, the database, i.e. a group of related tables, exists inside an ELI workspace as native data objects ready to be processed by *esql* statements or further manipulated by other ELI code. This integration of a database and its analysis code greatly speed up data analysis in terms of speed and effort. However, we should point out that the database system inside an ELI system has its limitations. First, *esql* contains most used SQL statements is not as complete as SQL in a fullfledged RDMS system. Second, it lacks *locking* facility. Hence, databases built inside ELI are more suitable for large data analysis than for transactional applications.

The ELI system itself is implemented in C++; in particular, each ELI primitive function is written in C++. The most intriguing fact about ELI having a native database subcomponent is that all *esql* statements are implemented in ELI as ELI defined functions and later packaged as system calls. In implementing *esql* statements, many ELI primitive operations on arrays and lists are used, and most importantly the *execute* function is used. This presents a challenge in case we attempt to speed up *esql* statements through compilation since our current ELI compiler method excludes the *execute* function. This challenge will guide our future effort in improving the run-time efficiency of more ELI applications.

7. CONCLUSIONS

We have presented the ELI programming language system. After tracing its heritage from APL we have shown two examples to illustrate some of its newer features. We described lists, dictionaries and tables in ELI as the base of having a native database component in ELI. Later, we introduced SQL-like statements, *esql*, in ELI together with an example database. We discussed ELI in comparison with several related programming language systems from many different angles. Finally, in explaining the implementation of *esql*, we brought our attention to a challenge to our effort in extending the coverage of ELI compilation.

REFERENCES

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

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

[3] H. Chen, W.-M. Ching, ELI: a simple system for array programming, vol. 26, No.1, Sept. 2013, 94-103, Vector,

[4] IBM APL2 Programming: Language Reference.

[5] W.-M. Ching, A Primer for ELI, a system for programming with arrays, 2011-2016,http://fastarray.appspot.com/.

[6] W.-M. Ching, ELI for kids, a novel way to learn math and coding, 2015-2016,http://fastarray.appspot.com/.

[7] Jeffery Borror, q for Mortals version 3, an introduction to Q programming, q4m LLC, New York, 2015.

[8] *ecc*: the ELI-to-C Compiler User's Guide,2013, http://fastarray.appspot.com/.

[9] W.-M. Ching, Program analysis and code generation in an APL/370 compiler, IBM Jour. Res. Dev. , vol.30, no.6, 594-602, 1986.

[10] Ching, W.-M, Ju, D.: An APL-to-C compiler for the IBM RS/6000: compilation, performance and limitations, APL Quote Quad 23(3), 15-21, 1993.

[11] Bates, J.: Some observation on using Ching's APL-to-C translator, APL Quote Quad 25(3), 47-48, 1995.

[12] The MathWorks, The MATLAB JIT-Accelerator, 2004.