A Primer for ELI

a system for programming with arrays

by Wai-Mee Ching

with assistance of Hanfeng Chen

revised 2012-11-28,2013-1-28,4-10,6-23,11-26

Copyrighted 2011

Preface

ELI is an array programming language system derived from APL but uses ACSII characters for easy communication. ELI has almost all functionalities of ISO APL [1], i.e. all flat array operations, but it also has features not prescribed in [1] such as complex number, symbol, temporal data, list, dictionary, table, esql, control structures and scripting files. We kept ELI simple since we want to make a clean and succinct way of doing array computing, unburdened by the legacy of FORTRAN, accessible to general public. We make such a tool freely available to let more people appreciate the fact that simplicity of rules and notation in a programming language leads to greater programming productivity.

The Primer is short; it is neither a tutorial nor a reference manual but does cover everything in the current ELI system. A careful reading of this Primer should enable a new comer to get sufficiently familiar with ELI to start using it. Thus, anyone with basic mathematics background can quickly pick up ELI to explore programming with arrays.

Currently, ELI is available on Microsoft Windows platform as well as on Linux and Mac OS. The directory structures are different in Windows and Linux/Mac OS. In Windows, under the directory `eli`, there are two subdirectories `bin` (which contains the `eli.exe` and documents) and `ws` (which contains workspaces and script files). In Linux/Mac OS these two are merged into one directory `elix/elim`. The user interface under Windows is through an ELI window while under Linux/Mac the interaction is command line based.

The project started around 2000 [2] but was dormant for almost a decade; during 2009-2012 ELI has been revived with new features such as lists, complex numbers and temporal data, and made publicly available in 2011 (current site: http://fastarray.appspot.com/). In 2013, we added dictionaries and tables as well as *esql*. Both developers are based in US and we welcome suggestions and comments from users around the world to further improve ELI.

1. Arrays, Primitive Functions and Operators

1.1 getting started

You get an Eli executable, *eli.exe*, either through an e-mail or download from web. You put *eli.exe* in your *desktop* or some directory in *my document*. Click on *eli.exe* icon and you see

```
ELI version 0.2 (C) Rapidsoft
    CLEAR WS
```

You *type* (the lines displayed with an indentation), and you see the system *responses* with a line:

```
    !10
1 2 3 4 5 6 7 8 9 10
    100+!10
101 102 103 104 105 106 107 108 109 110
    v<-100+!10
    v
101 102 103 104 105 106 107 108 109 110
    w<-2*v
    w
202 204 206 208 210 212 214 216 218 220
    2*100+!10
202 204 206 208 210 212 214 216 218 220
    w+v
303 306 309 312 315 318 321 324 327 330
    v+w<-2*v<-100+!10
303 306 309 312 315 318 321 324 327 330
```

First, `!` is a function which when applied to *n* generates a vector from `1` to *n*. `100` is then added to each elements in that vector. This vector can then be assigned to a variable `v` , and after multiplied by 2 we store the value in another variable *w* , whose value can also be generated by the line below (in that line + is done before * because + is to the right of *). We see here two ***basic*** principles in Eli:

i) *a line executes from right to left and all functions have equal precedence;*

ii) *for an arithmetic function **f**, A **f** B works if either one of the operand is a single or both are of equal shape (length).*

Rule i) also applies to defined functions, and ii) extends to scalar functions and arrays. Eli provides 65+ *primitive functions*. They are neither library functions nor are they represented by mnemonics. Rather, they are denoted by *special symbols* consist of one or two ASCII characters. For a *two-character symbol,* no blank is allowed in between; such a symbols either ends in a '.' or the combination carries clear meaning. A *function symbol* can denote either a *monadic* (i.e. has only a *right operand*) or a *dyadic* (has *left* and *right operands*) *function* depending on its *context* as shown in the following table.

Table of special symbols and the functions they represent

monadic function	*symbol*	*name*	*dyadic function*
	.	dot	
type	:	column	map/enumeration
unique	=	equal	equal
negate	-	minus	subtract
ravel	,	comma	catenate/laminate
raze	,.	comma dot	catenate on 1^{st} axis
pi	⍜	at	circle functions
	⍜.	del	
where	?	epsilon	member
roll	?.	random	deal
signum	*	star	multiply
exponential	*.	power	power
reciprocal	%	percent	divide
natural logarithm	%.	log	general logarithm
absolute value	|	bar	residue
factorial	|.	gamma	binomial
shape	#	rho	reshape
matrix inverse	#.	domino	matrix divide
reverse	$	turn	rotate
reverse along 1^{st} axis	$.	turn dot	rotate on 1^{st} axis
grade_up	<	left	less
enclose	<.	pack	encode
grade_down	>	right	greater
grouping	>.	unpack	decode
conjugate	+	plus	add
format	+.	format	special format
interval	!	iota	index of
execute	!.	exclaimation	drop
count	^	cup	and
first	^.	cup dot	take
	&	ampersand	or
transpose	&.	flip	general transpose
	_	high_minus	
floor	_.	lower	minimum
not	~	tilde	match
ceiling	~.	upper	maximum
partition_count	||	double bar	partition
	~=		not equal
	>=		greater or equal
	<=		Less or equal

reduce	*/ slash*	*compress*
reduce along 1^{st} *axis*	*/. slashdot*	*compress on* 1^{st} *axis*
scan	*\ backslash*	*expand*
scan along 1^{st} *axis*	*\. backslashdot*	*expand on* 1^{st} *axis*
	<-	*assign*
	->	*branch*
	// comment	
	[] quad	
	[) bare_quad	

In the table, the *names* in the middle column are the names of the *primitive symbol* next to it. Some are without names because they denote one function only, and the name of that function is the name of that symbol. For each row, the left column is the name of the *monadic function* which it represents while the name in the rightmost column is the names of the *dyadic function* which it represents. We shall go through all of them later in this Primer. Since the primitive functions are represented by symbols composed of special characters, there are no *reserved words* in Eli, other than 7 words for ***control structures*** which we'll introduce in the next chapter.

For a reader who is already familiar with APL, he can now go directly to **sect**. **1.11** on control structures and **chapter 3** on system facilities other than note that there is a simple type *symbol* described in the next section. We remark here that currently Eli only covers *real* numbers, i.e. no *complex* numbers; and Eli has no *nested* arrays but it has *lists* to accommodate heterogeneous data (see **sect. 1.11**).

// put everything to the right of it as a comment. To ***log off*** from Eli, simply type

```
     )off
```

1.2 data types, variables, shape, reshape and count

Eli deals with three ***types*** of data: *numeric, character* and *symbol* (we'll add *temporal* data later). A *numeric* data can be a *boolean*, an *integer*, a *floating point* number or a *complex* number. A boolean data is either 0 or 1, which represent **false** and **true** respectively and can result from a comparison

```
     2=2 3
1 0
```

or a logical operation on boolean data

```
     1 1 0 1 0 0&0 1 0 0 1 0
1 1 0 1 1 0

     1 1 0 1 0 0^0 1 0 0 1 0
0 1 0 0 0 0
```

where `&` is the ***or*** function and **^** is the ***and*** function. Boolean data can participate in all arithmetic operations as integers. But there are primitive functions (we'll see later) whose left operand must be boolean.

Character data are entered by paired (single) quotes but displayed without quotes; and to put a '' into a character string, write `''`:

```
     A<-'BC'
     ch<-'abcde''1234'
     ch
abcde'1234
     A
BC
     A+1
domain error
```

We see that `A` is a variable of character type, and apply addition to it results in a *domain error*, i.e. it is out of the domain where + is defined. If we do later,

```
     1+A<-10
11
```

Now `A` takes a numeric value 10 and addition is ok. Hence, in Eli variables can be assigned values on the fly *without first declaring their type (or dimensions)*, and can even change its type (even though it may not be a good idea to do so). But a variable must have been *assigned* value before you can use it:

```
     n+1
value error
```

A data of *symbol* type is entered with a back-tick ` followed by a character string (possibly empty) as

```
     a<-`abc
     a
`abc
```

A character string that is allowed to form a symbol should not have blanks; and not all symbols are name-like as `+ and `>= are also legitimate symbols.

A *name* in an Eli workspace denotes either a *variable* or a user *defined* functions. A *name* must start with a letter followed by alphanumeric characters plus the '_' character (but not as an ending character). There are also *special names,* i.e. names prefixed by a '_', reserved for system variables and system functions. As in Unix, names are *case-sensitive*. Clearly, there must be a *blank* between data, and between data and user function, i.e. numbers, character strings and names; but a blank is not required between a data item and a special symbol denoting a primitive function, or between symbols. So `a-3` and `a - 3` are the same.

To see what variables or defined functions are in a workspace, you type the system commands

```
     )vars
     )fns
```

In a *clear workspace*, there is none. Numbers are written in decimal form

```
     12 1.2 0.5 1.0
12 1.2 0.5 1
```

A floating point number involves a '.' but it can't start or end with a '.'

```
     .5
syntax error
```

Negative numbers are prefixed with '_'. Note that the monadic function '-' applies to each number to the right of it, and '_' is not a primitive function

```
     -1 2.5 3   (same as 0-1 2.5 3)
_1 _2.5 _3
```

Note that '_' is treated similar to a decimal point in a number, so it must be immediately followed by a digit (i.e. no blank is allowed between them). For scaling, both 'e' and 'E' are acceptable:

```
     1.2e2
120
     1.2E2
120
```

A complex number is of the form `RjI`, where `R` is the real part and `I` is the imaginary part, each written as an integer or a floating point number and there should be no space before or after `j`. For example the square root of `_1` is

```
     _1*.0.5
0j1
     2j5+3j2.5
5j7.5
     2j5*3j2.5
_6.5j20
```

Our discussion so far is on how to enter *literal* data, or assign a literal data to a *variable.* We call all literal data, variables and values resulted from evaluation of *expressions* simply *data*. A data in Eli is either a ***scalar***, an ***array*** or a ***list***. A *scalar* is a single *number*, a single *character*, a single *symbol*, a single *temporal data*, or a *variable* holding such a value. A one dimensional array is called a *vector*, a two dimensional array is called a *matrix*, and a *scalar* can be regarded as a 0-dimensional array. There are higher dimensional arrays, and the dimension of an array is called its' ***rank***. A vector is a rank 1 array, and a matrix is a rank 2 array and so on. The length of a vector is the number of elements in that vector, and a length 0 vector is called an *empty vector*. Surprisingly, there are *empty arrays* other than *empty vector*, i.e. one of its dimensions is 0. For any data, we can apply the ***shape*** function # to get its' *shape,* i.e. its' *dimensions*

```
     sa<-#a<-1
     sa
           (sa, the shape of a, is an empty vector)
     #sa
0           (its length is 0)
     b<-1 3 8 9 12
     #b
5
     #'K'
            (again an empty vector)
     ##'K'
0
     ##b     (b is of rank 1, i.e. a vector)
1
```

We can also create empty vector in two ways:

```
     #!0
0
     #''
0
```

Up to this point we may just wonder: *how do we input a matrix*, and other higher dimensional arrays, into Eli console? First, let us try to input a 2 by 3 matrix:

```
     a<-9 2 3
     4 5 6
4 5 6
     a
9 2 3
```

We see that multi-line doesn't work as expected. To this end we introduce the dyadic form of `#`, the ***reshape*** *function* `S#A`, where `S` must be a non-negative integer or vector while `A` can be any data:

```
     a<-2 3#9 2 3 4 5 6
     a
9 2 3
4 5 6
     #a
2 3
```

What the dyadic `#` (called ***rho***) does is to put the right side operand into an array of shape specified by the left operand. The left operand must be a vector whose elements are all non-negative integers; but that vector can have just 1 element or more that 2 elements.

```
     7#'wy'
wywywyw
     a3<-2 3 4#!21
     a3
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21  1  2  3
     #a3
2 3 4
```

We see that in case the right operand does not have enough elements to fill up the required result array or vector, it would reuse its elements in turn. The *shape* of the result from a *reshape* always equals to the left operand of the reshape. Moreover, we can create empty vector and empty matrix by a reshape

```
     #empv<-0#a
0
     #empm<-0 10#'A'
0 10
```

An empty vector usually is used to initialize a list, and an empty matrix is typically used to initialize a table of certain length (say `10` as in the above case).

There is a monadic function '`,`' called ***ravel*** acting like an *inverse* of reshape, which turns any data into a vector whose length is the product of shape vector (the dimensions) of the operand array

```
     ,a
9 2 3 4 5 6
     ,a3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 1 2 3
```

```
     #,a3
24
     #av<-,a0<-8
1
```

We see that the ravel of a scalar turns it into a *one-element vector*, i.e. a vector of length 1 of the same value. Often they behave the same in computations, but there could be subtle differences as one is of rank 0 while the other is of rank 1.

The dyadic function denoted by ‘,’ is called *catenate* which glues two pieces of data together: for `A` and `B` both to be either scalar or vector, `A,B` just attach `B` to `A`.

```
     0,!10
0 1 2 3 4 5 6 7 8 9 10
      (!10),_1
1 2 3 4 5 6 7 8 9 10 _1
     'WATCH OUT','!'
WATCH OUT!
     'WATCH OUT',' GUYS!'
WATCH OUT GUYS!
     100 200,2 3 5
100 200 2 3 5
```

We note that the difference between a character string (a vector) and a symbol (a scalar) here:

```
     #S<-'abc'
3
     #`abc
                    (again an empty vector)
     #,`abc
1
     #s3<-`abc `ddl `comp
3
     2 2#s3
`abc  `ddl
`comp `abc
```

For a vector `v`, the shape of `v`, `#v`, gives the number of elements in `v`; for a general array `a`, multiplying the elements in `#a`, i.e. the product of dimensions of `a` (`*/#a`), gives the number of elements in `a`. There is a monadic primitive function ***count*** `^`, for which `^w` gives the number of elements in `w`, and for a scalar `w`, `^w` is `1` (this saves us the need to *ravel* a scalar in cases for counting purpose). So we have,

```
     ^S
3
     ^`abc
1
     ^s3
4
     ^a3
24
```

We note that when entering a numeric vector a blank space is obviously needed between two neighboring numbers, but this is not needed in case of entering a vector of symbols:

```
     syms<-`a`b`c
     syms
`a `b `c
```

1.3 indexing and indexed assignment

Before we discuss indexing, let us revisit the monadic function `!`*n* which gives a vector `1…n`. There is actually a *system variable* `[]IO` called *index origin*, which by default is 1, but can be changed to 0 in a workspace.

```
CLEAR WS
     []IO
1
     !12
1 2 3 4 5 6 7 8 9 10 11 12
     []IO←0
0
     !12
0 1 2 3 4 5 6 7 8 9 10 11
```

Indexing a vector or an array depends on `[]IO`. For `[]IO=0`, it operates like in C. For `w` holding

```
3 1 ¯7 9 11 6 0 ¯2
```

```
     w[0 4 7]
3 11 ¯2
     w[8]
index error
     []IO←1
1
     w[8]
¯2
```

Eli signals an `index error` if any one of the *indices* used is *out of bound* and that depends on `[]IO`, i.e. each index must be an integer between `[]IO` and `(#v)+[]IO-1` for any vector v. What inside `[..]` is called an *index expression*. Elements in an index expression need not to be distinct nor necessary to be a scalar or vector, it can be an array as long as each of its elements is within the bound specified above:

```
     w[3#4 7]
9 0 9
     3#4 7
4 7 4
     w[2 3#4 7]
9 0 9
0 9 0
```

The shape of the result from an indexing is the shape of the index expression, i.e.

`#v[I]` ←→ `#I`

For matrices and high dimensional arrays, we can index into individual elements of an array, or we can get some of its sub-array by indexing certain slices.

```
     x
ABCD
EFGH
IJKL
     x[2;4]
H
     x[2 3;4]
HL
     x[2 3;1 4]
EH
IL
```

```
     x[;4]
DHL
     x[;,4]
D
H
L
     x[1 2;]
ABCD
EFGH
```

where an *empty expression* before or after a ';' indicates that all elements in the corresponding *axis* are to be taken. We also notice the difference between indexing by a scalar `4` and by a one element vector `,4`. The first results in a vector while the second results in a 3 by 1 matrix. In general, if `A` is an k-dimensional array, then

`I` ←→ `I1;I2;…;Ik`

is a ***valid*** *index expression* for `A` provided each `Ij` is either empty or is an integer expression whose value (or value of its elements in case of array) lies within `!(#A)[j]`; each `Ij` is called a component of the index expression `I`, and the shape of

```
A[I]
```

is the Cartesian product of shapes of `Ij`'s. For example,

```
     A<-2 3 4#!24
     A
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
     A[1;2 3;1 2 4]
5  6  8
9 10 12
     A[;1 3 2;2 4]
 2  4
10 12
 6  8

14 16
22 24
18 20
```

If one of the elements in `Ij` is not in `!(#A)[j]`, the system responses with a `index error` message. If the number of '`;`' is not equal to `(##A)-1` then we have a *rank error*

```
     A[2;3 4]
rank error
```

We have already seen *simple assignment*

```
anv<-expression
```

where `anv` is a variable, which may or may not need to have appeared before, and `expression` is a line of code to the right of `<-`. Eli evaluate this line from right to left as we stated in the **getting started** sect. and put the

resulting value into *anv*; and that resulting value is also the result of the assignment. Hence, we can assign some (initial) value to a group of variables, `A, B, C`, … all at once

```
     A<-B<-C<-!0
```

and remember the *equal precedence* rule, which can be overwritten by parentheses

```
     3*2+1
9
     (3*2)+1
7
```

To set up a variable `E` with the same number of `0`s as the length of another vector `V`,

```
     E<-#V#0
```

would have `E` holding the value of `V` if `V` is an integer vector, but the following will do the intended

```
     E<-(#V)#0
```

An ***indexed assignment***

```
     A[I]<-B
```

changes the values of certain elements of an existing array `A`, specified by the index expression `I`, to a new set of values denoted by `B`. For this to be a valid assignment, first `I` must be a valid index expression with respect to *A* as explained earlier in this section, and second, the shape of `B` must be the same as that of `I` or `B` is a scalar. The result of the indexed assignment is the replacement of the values of elements in `A` specified by `I` with the corresponding elements from `B` or the single scalar value `B`. In case there are duplicates in `I`, the later ones in `B` overwrite earlier ones.

```
     av<-!12
     av[2 3]<-12 13
12 13
     av[2*!6]<-0
0
     av
1 0 13 0 5 0 7 0 9 0 11 0
     a3<-2 3 4#!21
     a3[1;2 3;3 4]<-2 2#!4
1 2
3 4

     a3
 1  2  3  4
 5  6  1  2
 9 10  3  4

13 14 15 16
17 18 19 20
21  1  2  3

     a3[2;;1 3 4]<-1
1
     a3
1  2 3 4
5  6 1 2
9 10 3 4
```

```
1 14 1 1
1 18 1 1
1  1 1 1
```

Indexed assignment is the only kind of assignment with expressions other than a variable name allowed on the left side of '<-' in Eli.

1.4 arithmetic and circle functions

A monadic primitive function *f* is called a ***scalar function*** if *f(s)* is again a scalar for a scalar *s*, and that *f(A)* has the same shape as that of *A* for an array *A* plus that any element *(f(A))ij* of the result equals to *f(Aij)* (so ! and # are not monadic scalar functions). A dyadic primitive function *f* is called a *scalar function* if *z f s* is again a scalar for scalars *z*, *s*; otherwise for *B f A* , *A* and *B* must be *compatible*, i.e. either one of them is a scalar or they must be of the same shape *S*, then *B f A* is of shape *S* plus that any result element *(B f A)ij = Bij f Aij,* where either *Bij* or *Aij* could be the reshape of a scalar.

Hence, the application of a scalar function to an array, or arrays, is a natural extension of its application to each (pair of) element(s) of the array(s). Thus, a scalar function preserves the shape of its operand(s). All arithmetic, logical and relational functions are *scalar functions*. Other functions are called *mixed functions*. Arithmetic functions operate on numbers while logical functions operate on boolean data. We have see the addition `+`, subtraction `-` and multiplication `*` functions. The monadic form of `*` is the *signum function*:

`*0` is `0`, `*p` is `1` for any number `p>0`, and `*n` is `_1` for any number `n<0`.

Suppose `V` and `P` are the volumes and prices of a stock trades in the first minute of market opening, `P0` (=`20`) is the closing price of that stock in the previous day. To mark the volumes of up-trades positive and down-trades negative:

```
     V
86 25 55 48 78 95 36 36 14 65
     P
17.3 22.3 24.3 17.5 21.4 18.4 17.2 15 20.8 21.8
     *P-P0<-20
_1 1 1 _1 1 _1 _1 _1 1 1
     V**P-P0
_86 25 55 _48 78 _95 _36 _36 14 65
```

The *power function L*`*`*.R* raises the base to the *R*th power of *L*

```
     2 _3 10 0.5 25 *. 2
4 9 100 0.25 625
     10 *. _1 0 1 2 3
0.1 1 10 100 1000
     2*.!10
2 4 8 16 32 64 128 256 512 1024
     4 9 100 0.25 625 *.0.5
2 3 10 0.5 25
```

We see that for a negative number *–p* on the right *n*`*`*.-p* is the inverse of (*n*`*`*.p*); and that *v*`*`.0.5 is taking the square root of *v.*

The *monadic* form of `*.` is the mathematical *exponential* function.

```
     e<-*.1
     e
2.7182818
     *. _1 0 1 2 3
0.36787944 1 2.7182818 7.3890561 20.085537
     e*. _1 0 1 2 3
0.36787944 1 2.7182818 7.3890561 20.085537
```

We see that `e` above is the famous transcendental number ***e***, and that `*.`*v* is just a convenient way to write *e**.*v* for any numerical (scalar or array) *v*.

The *divide* function is denoted by `%`

```
     (!6)%2
0.5 1 1.5 2 2.5 3
     24%!6
24 12 8 6 4.8 4
```

The monadic form of `%` is the *reciprocal* function; `%v` is simply `1%v`.

```
     %!6
1 0.5 0.33333333 0.25 0.2 0.16666667
```

Now to take cubic root of numbers or various roots of a number, we do

```
     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
```

The next pair of arithmetic functions is the *log* functions `%.` : for dyadic `B%.L`, it is the `B` based log of `L`, and the monadic `%.L` is the *natural logarithm* function, i.e. the same as the dyadic one with `B` fixed to be `=e`. So, `%.` is the inverse function of `*.` just like `%` is the inverse function of `*` .

```
     10%. 1 10 100 1000 10000
0 1 2 2 3 4
     10%. 1 10 100 1000 10000 100000
0 1 2 3 4 5
     2%. 0.5 1 2 4 8 16
_1 0 1 2 3 4
     2 3 10%. 4 9 100
2 2 2
     e<-*.1
     e
2.718282
     %.1 10 100 1000 10000 100000
0 2.302585 4.60517 6.907755 9.21034 11.512925
     e%.1 10 100 1000 10000 100000
0 2.302585 4.60517 6.907755 9.21034 11.512925
```

The monadic `|` is the *absolute value* function, i.e. `|p` equal to p if `p>0` or `=0`, and `|p` equal to `-p` if `p<0`.

```
     | _1 2 _3.2 5 0 _10
1 2 3.2 5 0 10
```

The dyadic function A|B is the *residue* function, i.e. its value is the remainder of B after divided by A.

```
     2|!12
1 0 1 0 1 0 1 0 1 0 1 0
     2 3 4.5|9
1 0 0
     3.2|3 6.4 7
3 0 0.6
```

The monadic form of _. is the *floor* function, i.e. the value of _. v is the largest integer i which is less than or equal to v.

```
     _. 9.2 3 10.5 _1.5 _100
9 3 10 _2 _100
```

The dyadic form of _. is the *minimum* function, i.e. the value of *a*_. *b* is the smaller of *a* and *b*.

```
     3 5 7 9.4_.2 6 8 20
2 5 7 9.4
```

The functions denoted by ~. are the mirror images of that for _. . The monadic form of ~. is the *ceiling* function, i.e. the value of ~.v is the smallest integer i which is greater than or equal to v.

```
     ~.9.2 3 10.5 _1.5 _100
10 3 11 _1 _100
```

The dyadic form of ~. is the *maximum* function, i.e. the value of a~. b is the greater of *a* and *b*.

```
     3 5 7 9.4~.2 6 8 20
3 6 8 20
```

The last of arithmetic functions are the ***circle functions*** @. The monadic @n is just *pi* times *n*.

```
     @ 1 2 3
3.141593 6.283185 9.424778
```

The dyadic L@R actually represents a group of functions (mostly *trigonometric* with several others), where *L* can take any integer value from _12 to 11; we have

Table of Circle Functions in ELI

L	L @ R	L	L @ R
		0	(1 - R*.2)*.5
_1	arcsin R	1	sine R
_2	arccos R	2	cosine R
_3	arctan R	3	tangent R
_4	(_1 + R*.2)*.5	4	(1 + R*.2)*.5
_5	arcsinh R	5	sinh R
_6	arccosh R	6	cosh R
_7	arctanh R	7	tanh R
_8	-(8 @ R)	8	-(_1 - R*.2)*0.5 for R>=0
			(_1 - R*.2)*0.5 for R<0
_9	R	9	real R

_10	**+R**	10	**|R**
_11	**0j1*R**	11	**imaginary R**
_12	***.0j1*R**		

For example,

```
     4@3 0.5 1
3.162278 1.118034 1.414214
```

Finally, we remark here that we only used scalar and vector data in our examples in this section as well as in the next section. This is only for convenience and succinctness of presentation. Since they are all scalar functions, they apply to matrices and arrays of higher dimension in a similar manner.

So far our examples of arithmetic functions have only touched real numbers, but we already introduced complex numbers in **sect**. 1.2, i.e.

```
     _1*.0.5
0j1
```

and dyadic +, -, *, %, *. , %. as well as monadic `-, %, *.` just follow the usual rule of complex numbers mathematical operations. We also have specified how some circle functions apply to complex numbers. We now discuss how monadic +, the *conjugate* function, and monadic *, the *signum* function apply to complex numbers. For a real number `a, +a` is just `a`; for a complex number `+ajb` is `aj_b`

```
     +2j3
2j_3
```

For a complex number `c` , `*c` is an unit length complex number in the same direction as that of `c`:

```
     *3j4 0j1 0j_1
0.6j0.8 0j1 0j_1
```

We also have the examples of monadic scalar functions applied to complex numbers

```
     %3j4
0.12j0.16
     |3j4
5
     _.2.1j3.2 _1.2j2.5        //floor
2j3 _1j2
     ~.2.1j3.2 _1.2j2.5        //ceiling
2j4 _1j3
     @1
3.141592654
     *.0j1*@1                  // e^{iπ} = −1
_1
     %._1
0j3.141592654
```

1.5 relational and logical functions

Relational functions are also scalar functions; they are all dyadic, and all yield boolean results. They are the six primitive relational functions, *equal, not equal, greater, greater or equal, less, less or equal* :

```
=   ~=   >   >=   <   <=
```

(note that for a legitimate primitive symbol in Eli, no blank is allowed in a 2-character symbol such as <=). The first two functions test for equality, and for which `A fn B` is valid as long as *A* and *B* are shape compatible. For the rest 4 functions, left and right operands must be either both numeric or characters. A mixed *order* comparison will result in a `domain error`. The order of numeric data is their mathematical order. The order of character data is their *ascii-character* order.

```
     'A'=1
0
     'A'>'1'
1
     'ABb'<'B'
1 0 0
     'ABb'='B'
0 1 0
     'ABb'>='B'
0 1 1
     14 76 46 54 22 5 68 68 94 39 < 50
1 0 1 0 1 1 0 0 0 1
     52 84 4 6 53 68 1 39 7 42 > 14 76 46 54 22 5 68 68 94 39
1 1 0 0 1 1 0 0 0 1
     `abc='abc'    (since `abc is a symbol while 'abc' is a character vector of length 3)
0 0 0
```

For two symbols they can only be compared for *equal* or *not equal*.

There are three *logical* functions ~ (*not*), ^ (*and*), & (*or*) in Eli which operate on boolean data only:

```
     ~ 1 0 1 0 1 1 0 0 0 1
0 1 0 1 0 0 1 1 1 0
     1 0 1 0 1 1 0 0 0 1 ^ 1 1 0 0 1 1 0 0 1 1
1 0 0 0 1 1 0 0 0 1
     1 0 1 0 1 1 0 0 0 1 & 1 1 0 0 1 1 0 0 1 1
1 1 1 0 1 1 0 0 1 1
```

1.6 operators and their derived functions

Eli provides 4 ***operators*** to apply to *dyadic scalar functions* to produce ***derived functions*** *f*; they are ***reduce*** **/** , ***scan*** ****, ***outer product*** **.:** and ***inner product*** **:** . For **/** and **** the derived function, *f*/ or *f*\, is monadic. When *f* is + , +/ is the *summation* function, and if *f* is ~. , ~./ is the *total maximum* function, _./ the *total minimum* function, */ the *product* function:

```
     +/!100
5050
     +/14 76 46 54 22 5 68 39
324
     ~./14 76 46 54 22 5 68 39
76
     _./14 76 46 54 22 5 68 39
5
     */14 76 46 54 22 5
290727360
```

In general, the result of `f/V` is always a scalar for a vector `V`; i) for *s* a scalar or one element vector, the value *f*/*s* is *s* or a scalar holding value of *s*; ii) if V is a vector of length n, then

```
f/V  ←→  V[1] f V[2]….V[n-1] f V[n]
```

i.e. the value of `f/V` is computed by inserting `f` between all elements of `V`.

```
     -/3 4 5
4
     3-4-5      (remember to evaluate from right to left)
4
```

A special note: to compare a variable b with the – reduction or negation of a, a blank is required

```
     a<-!10
     b<--/a
     b
_5
     b<-/a
left argument missing
     b< -a
1 1 1 1 0 0 0 0 0 0
     b< -/a
0
```

(iii) what happens if `V` is *empty*? The result then is the *identity element* `w` of the function *f*, that is the value which is *neutral* to the function *f* in the sense that `a f w` is `a` for all `a`:

```
     +/!0
0
     */!0
1
     +./0
0
     &/!0
0
     ^/!0
1
```

(iv) for `A` being a matrix or higher dimensional array, `f/A` is evaluated by applying `f/` to the vector along the last axis of `A` to form a resulting vector or array

```
     m
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
     +/m
10 26 42
58 74 90
     +/m[1;;]
10 26 42
```

For a commutative function *f*, i.e. `a f b = b f a`, to get the total *f*, you can apply ‘,’ to `m` then do reduce.

```
     +/,m
300
```

or for a matrix operand, you can apply *f/* twice

```
     +/+/m[1;;]
78
```

What if we want to *reduce along the first axis*, i.e. columns in case of a matrix? /. would do just that

```
     +/.m[1;;]
15 18 21 24
     +/.m
14 16 18 20
22 24 26 28
30 32 34 36
```

The ***scan*** operator \ also takes a left argument *f*, which is a dyadic scalar function, to produce a monadic *derived function f*. When *f* is +, +\ is the *partial sum* function, and if *f* is ~. , ~.\ is the *cumulative maximum* function, _.\ the *cumulative minimum* function, *\ the *partial product* function:

```
     +\14 76 46 54 22 5 68 39
14 90 136 190 212 217 285 324
     ~.\14 76 46 54 22 5 68 39
14 76 76 76 76 76 76 76
     _.\14 76 46 54 22 5 68 39
14 14 14 14 14 5 5 5
     *\14 76 46 54 22 5
14 1064 48944 2642976 58145472 290727360
```

`&\` turns bits in a boolean bits vector into 1's once it sees a 1, and ^\ turns bits into 0 once it sees a 0.
We'll see later that combine with other primitives, these two derived functions are very useful.

```
     &\0 0 0 1 0 1 1 0 0 0 1
0 0 0 1 1 1 1 1 1 1 1
     ^\1 1 1 1 0 1 1 0 0 0 1
1 1 1 1 0 0 0 0 0 0 0
```

For the cases (i) and (iii) described above, the result of f\ is the same as that of f/. For a vector `V`

`(f\V)[i]` ←→ `(f/V[!i])`

assuming `[]IO=1`, i.e. the value of `f\V` at position `i` is the reduce of `f` on `V[1…i]`. This rule extends to matrices and other arrays. Hence, *scan* preserves shape, i.e. `#f\A` equals `#A`. For the `m` above,

```
     +\m
 1  3  6 10
 5 11 18 26
 9 19 30 42

13 27 42 58
17 35 54 74
21 43 66 90
     +\m[1;;]
1  3  6 10
5 11 18 26
9 19 30 42
     +\.m
 1  2  3  4
 5  6  7  8
 9 10 11 12

14 16 18 20
```

```
22 24 26 28
30 32 34 36
     +\.m[1;;]
 1  2  3  4
 6  8 10 12
15 18 21 24
```

where the *scan along 1st axis* ***operator*** \. is similarly defined as that of /. *reduce along first axis*.

Let `C` be a coefficient vector of a polynomial, and set `[]IO=0`, i.e. `C` represent the polynomial in `x` of degree `n`

`C[n]x`^{n} `+…+C[2]x`^{2} `+C[1]x+C[0]`

To evaluate this polynomial at a point `B` we do

```
+/C * 1,*\(_1+#C)#B
```

We note that the expression to the right of scan is a vector of `B`'s of length `n=(#C)-1`, and the scan puts that into a vector of successive powers of *B*.

The other two ***operator***s, *outer product* and *inner product*, take one (*f*) and two (*f,g*) dyadic scalar function(s) respectively to produce dyadic derived functions. The outer product operator is denoted by .: and applies to a function *f* on its right .:*f ;* for two vectors `A` and `B`,

(`A`.:*f* `B`)`[i;j]` ←→ `A[i]` *f* `B[j]`

```
     A<-15 18 21 24
     B<-100 200 2 3 5
     A.:*B
1500 3000 30 45  75
1800 3600 36 54  90
2100 4200 42 63 105
2400 4800 48 72 120
     A.:%3 5 2
5 3  7
6 3  9
7 4 10
8 4 12
     (!3).:<!5
0 1 1 1 1
0 0 1 1 1
0 0 0 1 1
```

To see how 1000 dollars will grow under 3%, 5% and 8% of annual interests 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
```

In general, then `#(A.:f B)` ←→ `(#A),#B` as we see for the m above

```
     m.:+100 200 300 400 500
101 201 301 401 501
102 202 302 402 502
103 203 303 403 503
104 204 304 404 504
```

```
105 205 305 405 505
106 206 306 406 506
107 207 307 407 507
108 208 308 408 508

109 209 309 409 509
110 210 310 410 510
111 211 311 411 511
112 212 312 412 512

113 213 313 413 513
114 214 314 414 514
115 215 315 415 515
116 216 316 416 516

117 217 317 417 517
118 218 318 418 518
119 219 319 419 519
120 220 320 420 520

121 221 321 421 521
122 222 322 422 522
123 223 323 423 523
124 224 324 424 524
     cv.:*cv<-2j3 0j2 _2.3    //outer product of *
  _5j12   _6j4    _4.6j_6.9
  _6j4    _4        0j_4.6
_4.6j_6.9  0j_4.6 5.29
     +cv.:*cv<-2j3 0j2 _2.3   //conjugate of outer product
  _5j_12 _6j_4  _4.6j6.9
  _6j_4  _4       0j4.6
_4.6j6.9  0j4.6 5.29
```

The *inner product operator* `:` takes two dyadic scalar functions *f* and *g* to produce a dyadic derived function *f*`:`*g*; (we remind APL people *f.g* is ***not*** inner product in Eli) for vectors `V` and `W`, `V` *f*`:`*g* `W` is `f/V`*g*`W`. For example,

```
     V<-2 5 1 4
     W<-1 9 6 7
     V~.:+W
14
     V+:~.W
24
     m1<-3 4#!12
     b2<-2 3#!6
     m1
1  2  3  4
5  6  7  8
9 10 11 12
     b2
1 2 3
4 5 6
     b2+.*m1
length error       (oops, inner product is +:* not +.*)
     b2+:*m1
38 44  50  56
83 98 113 128
```

So we see that `+:*` is our usual matrix multiplication, but inner product is more useful than that. Let `STS` be a *slice* of data from students' record representing their home states, `ST6` represents six Northeast states, and `&.ST6` is the transpose of the character matrix `ST6`.

```
     STS
NY
NJ
CT
NY
MA
PA
NY
CT
MA
RI
PA
NJ
NY
NJ
CT
     ST6
CT
MA
NJ
NY
PA
RI
     &.ST6
CMNNPR
TAJYAI
     STS^:=&.ST6
0 0 0 1 0 0
0 0 1 0 0 0
1 0 0 0 0 0
0 0 0 1 0 0
0 1 0 0 0 0
0 0 0 0 1 0
0 0 0 1 0 0
1 0 0 0 0 0
0 1 0 0 0 0
0 0 0 0 0 1
0 0 0 0 1 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 1 0 0 0
1 0 0 0 0 0
     +/.STS^.:=&.ST6
3 2 3 4 2 1
```

The result of inner product ***and*** *with* ***equal*** is a 15 by 6 matrix of 0 and 1's; each row has one 1 points to a *matched* state and `+/.` gives the number of students from each of the six states.

There is a fifth operator, the ***axis operator***, `[x]`, applies to some structural primitive functions, which we will explain in section **1.8**.

1.7 member of, index of, where, compress, expand and unique

Two very useful mixed functions in Eli are ***member of*** (`A?S`) and ***index of*** (`V!B`). For the first function, both arguments `A` and `S` can be any scalar or array; for each elements `a` in `A` it tests whether `a` is a member of the set consisting of elements in `S`, the result in that corresponding position is `1` if it is, and `0` if not. Hence `#A?S` ←→ `#A`. In particular, if `A` is empty then the result is empty, and if `S` is empty then the result is `(#A)#0`. For example,

```
     1 2 3?!0
0 0 0
     (!0)?2 3
                    (empty vector)
     #(!0)?2 3
0
     5?!6
1
     a
ABCD
EFGH
IJKL
     s<-'A BOOK'
     a?s
1 1 0 0
0 0 0 0
0 0 1 0
```

For the *index of* function `v!c` the left argument `v` must be a vector while the right argument `c` can any scalar or array. For `c` a scalar, `v!c` gives the index of the first element in `v` which equals to `c` if `c?v` is 1, or 1 plus the index of the last element of `v` if `c?v` is 0. This function obviously depends on `[]IO` which we assume to be 1 here. For example

```
     2 4 6 7 8!6
3                          (if []IO=0, it would be 2)
     2 3 6 7 8!5
6                          (if []IO=0, it would be 5)
```

For general `B`, `#V!B` ←→ `#B` and each element of `V!B` gives the index of the corresponding element of `B` in `V` as in the scalar case. For example

```
     s!a
1 3 7 7
7 7 7 7
7 7 6 7
```

The *member of* function `?` tells us whether a particular element (of an array) is ***in*** the set of another array. The *index of* function `!` tells us more, i.e. the *position*, in case the array we check against is a vector. So, *index of* function gives a kind of *associative search* capability in Eli. Please note that the *roles* played by the left and the right arguments in *index of* (`!`) is the *reverse* of that in *member of* (`?`).

The monadic form of (`?`) is the ***where*** function: for a boolean (vector) operand `B`, `?B` gives the indices of elements in `B` which are `1`.

```
     ?1 0 0 1 1 0 1 0 0 0 1 0
1 4 5 7 11
```

This function clearly depends on `[]IO` as the above is for `[]IO=1`, and for `[]IO=0` we would have

```
0 3 4 6 10
```

The ***compress*** (`B/A`) function let us select elements of `A` according to the left argument `B` which must be a boolean vector while `A` can be any array. First, suppose `A` is a vector `V`; then we must have

```
(#B) = #V
```

the function then selects those elements of V whose corresponding element in B is 1 to form the result vector

```
     B<-0=2|V<-!10
     V
1 2 3 4 5 6 7 8 9 10
     B
0 1 0 1 0 1 0 1 0 1
     B/V
2 4 6 8 10
    (~B)/V
1 3 5 7 9
     X
14 76 46 54 22 5 68 39
     +/(X>50)/X
198
```

Hence, one typically uses a *B* to represent some proposition *P* about elements of *V* to get those and only those by *compress*; in the above *P* is the proposition that {*can be evenly divided by 2*}, and the result of B/V is the set of even numbers in V. And for a W,

```
          (V?W)/V
```

gives the *intersection* of V and W. Now V may have duplicate elements, but

```
          ((V!V)=!#V)/V
```

yields the unique elements of V. This is because by the definition of dyadic !, only when an element V[i] appears for the first time, the value of *index of* at that position would equal to i. V can be characters or floating point numbers. Indeed, we have a monadic primitive function ***unique*** = to do just that (note that the right operand must be a vector):

```
     = 14 76 14 46 54 22 5 22 68 5 39
14 76 46 54 22 5 68 39
```

For an array A, B/A selects elements of A along the *last axis*

```
     a
ABCD
EFGH
IJKL
     1 0 1 0/a
AC
EG
IK
```

To ***compress*** *along the first axis*, we have the function /.

```
     1 0 1/.a
ABCD
IJKL
```

The function which plays the reversal role to compress is ***expand***, B\A, with B also required to be a boolean vector. For a vector V as A, we must have (+/B) = #V, expand then distributes elements of V to positions corresponding to a 1 in B and fills holes by a typical element of V, which is 0 if V is numeric and a blank if V is a character string.

```
     B
0 1 0 1 0 1 0 1 0 1
     W<-2 4 6 8 10
     B\W
0 2 0 4 0 6 0 8 0 10
     B<-1 1 0 1 1 0 1 1 1 1 1 1
     W<-'itissimple'
     B\W
it is simple
```

`B\A` extends to array A in a similar manner as that for `B/A`, and `\.` is ***expand*** *along the first axis*.

1.8 structural transformations of arrays

In addition to `s#v`, there are 9 more primitive functions that *transform* their array arguments *structurally* but do not change or re-arrange the values of the array elements. They are ***take*** `(I^.A)`, ***drop*** `(I!.A)`, ***first*** `(^.A)`, ***rotate*** `(I$A)`, ***reverse*** `($A)`, ***transpose*** `(&.A)`, ***ravel*** `(,A)`, ***catenate*** `(A,B)` and ***laminate*** `(A,[x]B)`.

When the right argument of a ***take*** is a vector `V`, the left argument `I` must be an integer. For example

```
     3^.1 2 3 4 5 6 7
1 2 3
     _3^.1 2 3 4 5 6 7
5 6 7
     9^.1 2 3 4 5 6 7
1 2 3 4 5 6 7 0 0
     5^.`abc `ddl `comp
`abc `ddl `comp ` `
```

i.e. `S^.V` for `S>0` (resp. `S<0`) takes the *first* (resp. *last*) `S` elements of `V`, and if `(|S)>#V`, additional slots in the result are filled by the *typical element* of `V` (0 for numeric `V`, ' ' for character `V` and ` for symbol `V`). The ***first*** function `^.V` differs from the special case of `1^.V`:

```
     ^.`abc `ddl `comp
`abc
```

in that `V[[]IO]=^.V` is a *scalar* while `1^.V` is a *one-element vector*; and for any `A`, `^.0#A` yields a *typical* (scalar) *element* of `A`. For general `A`, `I` is an integer vector such that `(#I)` equals `##A`, and `I[j]` indicates the number of elements to take along the *j-th axis*. For example

```
     a
ABCD
EFGH
IJKL
     2 3^.a
ABC
EFG
     _1 2^.a
IJ
```

drop `!.` is the opposite of ***take*** `^.`, it drops elements from a vector or array instead of taking them to form the result, yet it operates by the same rule:

```
     3!.1 2 3 4 5 6 7
4 5 6 7
     _3!.1 2 3 4 5 6 7
```

```
1 2 3 4
     1 2!.a
GH
KL
     1 0!.a
EFGH
IJKL
```

If `S>#V`, then `S!.V` results in an empty vector; for `I!.A,` if any `I[j]` is `>` length of `A`'s j-th axis, it results in an empty array. The the ***rest*** function in LISP is the special case of `1!.V`:

```
     1!.1 2 3 4 5 6 7
2 3 4 5 6 7
```

The monadic `$` is the ***reverse*** function, `$A` *reverses* elements in `A` for vectors, or along last axis of `A`:

```
     $1 2 3 4 5 6 7
7 6 5 4 3 2 1
     $a
DCBA
HGFE
LKJI
```

The ***reverse*** function can be modified by the axis operator `[1]([0]` in case`[]IO=0`) to reverse an array along the first axis, or more conveniently, `$.` :

```
     $[1]a      //$.a
IJKL
EFGH
ABCD
```

The dyadic `$` is the ***rotate*** function; `I$A` *rotates* rows (along the last axis) according to integer(s) I. For vector `A, I` is an integer, it rotates `A` by `I` positions to the right (resp. left) right if `I>0` (resp. left).

```
     3$1 2 3 4 5 6 7
4 5 6 7 1 2 3
     _2 $ 1 2 3 4 5 6 7
6 7 1 2 3 4 5
```

For a matrix `A, I` is an integer vector of length equal to the depth of `A`, and each `I[j]` determines the direction and amount the j-th row `A[j;]` takes; for `I` a scalar, then it rotates each row of `A` by that amount. We have

```
     a
ABCD
EFGH
IJKL
     1 2 3$a
BCDA
GHEF
LIJK
     1$a
BCDA
FGHE
JKLI
     _1 1 2$a
DABC
FGHE
KLIJ
```

As in *reverse,* ***rotate*** can also be modified by an *axis operator* `[1]` to rotate around the first axis, or just use `$.`:

```
     1$[1]a    //1$.a
IJKL
EFGH
ABCD
```

To rotate a high dimensional array `A`, `R$A`, `R` must be a scalar or `#R` must equal to `_1!#A`:

```
     A<-2 3 4#!24
     2$A
 3  4  1  2
 7  8  5  6
11 12  9 10

15 16 13 14
19 20 17 18
23 24 21 22
     r  // r<-2 3#!6
1 2 3
4 5 6
     r$A
 2  3  4  1
 7  8  5  6
12  9 10 11

13 14 15 16
18 19 20 17
23 24 21 22
     l<-3 4#!12
     l$.A
13  2 15  4
17  6 19  8
21 10 23 12

 1 14  3 16
 5 18  7 20
 9 22 11 24
```

**rotate* `$[x]` with `x` other than the first or last axis applies to higher dimensional arrays as well, but that has not been implemented at this time.

The ***monadic transpose*** `&.` of a vector `V` just leaves `V` not changed; for a general array `A`, `&.A` reverses the axises of `A`, i.e. it flips `A`, the first axis becomes the last axis and the last axis becomes the first, etc. It is easy to see for matrices

```
     &.a
AEI
BFJ
CGK
DHL
     &.4 6#!24
1  7 13 19
2  8 14 20
3  9 15 21
4 10 16 22
5 11 17 23
6 12 18 24
```

Now we can code an interesting piece of Eli that lists temperatures from `C-4` to `C15` in descending order together with their Fahrenheit degrees:

```
     &.2 20#c,32+1.8*c<-$_5+!20
15  59
14  57.2
13  55.4
12  53.6
11  51.8
10  50
 9  48.2
 8  46.4
 7  44.6
 6  42.8
 5  41
 4  39.2
 3  37.4
 2  35.6
 1  33.8
 0  32
_1  30.2
_2  28.4
_3  26.6
_4  24.8
```

General case for the ***dyadic transpose*** function `I&.A`. has **not** been implemented at this time, but the special case of taking diagonal with left argument being `1 1` is available.

```
     1 1&.a
AFK
     w3
1 2 3
4 5 6
7 8 9
     1 1&.w3
1 5 9
     1 1&.w<-4 6#!24
1 8 15 22
```

We have already introduced the monadic function ravel `(,)` and a limited form of the dyadic function ***catenate*** `(,)` in **sect**. **1.2**. In general, for arrays `A` and `B`, `A,B` is defined with the restriction that `(#A)` and `(#B)` must be the same except the lengths of the last axis, so each row of the result is formed by putting together a row from `A` with a corresponding row from `B`. Suppose `w2<-2 3 4#!24` then we have:

```
     w2,c2<-2 3 2#1
 1  2  3   4 1 1
 5  6  7   8 1 1
 9 10 11 12 1 1

13 14 15 16 1 1
17 18 19 20 1 1
21 22 23 24 1 1
```

If one of the arguments is a scalar, then that scalar is extended to conform to the other operand similar to scalar extension for scalar functions:

```
     'O',a,'X'
OABCDX
OEFGHX
```

```
OIJKLX
```

If one argument is a vector whose length equals the length of the first dimension of the other operand, then catenate can also be carried out:

```
     b1<-'xyz'
     a,b1
ABCDx
EFGHy
IJKLz
```

The ***catenate*** function (,) can also modified by the ***axis operator*** `[1]` (`[0]` in case of index origin = 0) to *catenate* two operands along the first axis, which can also be written as ‘,.’:

```
     b
abcd
efgh
     a,[1]b    //a,.b
ABCD
EFGH
IJKL
abcd
efgh
     b2<-'w',b1
     a,.b2
ABCD
EFGH
IJKL
wxyz
     w2,[2]0
 1  2  3  4
 5  6  7  8
 9 10 11 12
 0  0  0  0

13 14 15 16
17 18 19 20
21 22 23 24
 0  0  0  0
```

For two arrays of identical shape, we can specify a fractional number `F` in an axis specification after ‘,’, `A,[F]B` to get a new array which *joins* the two given arrays `A` and `B` along a *new axis* with its relative position indicated by `F`: for `[]IO=1, 0<F=0.5<1` implies the new axis will be the first axis and the `F=1.5` last axis. This function is called ***laminate***. For example,

```
     (!5),[0.5]10+!5
 1  2  3  4  5
11 12 13 14 15
     (!5),[1.5]10+!5
1 11
2 12
3 13
4 14
5 15
     (!5),[1.5]10
1 10
2 10
3 10
4 10
5 10
```

Now the example above for listing converted temperatures can be written as

```
c,[1.5]32+1.8*c<-$_5+!20
```

1.9 encode, decode and sort

For a vector of integers `R` and a scalar number `A`, (`R<.A`), the ***encode*** of `A` by `R` gives an encoding of `A` for radix `R`, and it is of the same length as that of `R`. For example,

```
     2 2 2<.32.5
0 0 0.5
     2 2 2 2 2 2<.32.5
1 0 0 0 0 0.5
     2 2 2 2<.12
1 1 0 0
     8 8 8<.12
0 1 4
     10 10 10<.12
0 1 2
```

give the binary, octal and decimal representations of the number on right. `R` needs not to be uniform:

```
     24 60 60<.3723
1 2 3
```

means 3723 seconds is the total time of 1 hour 2 minutes and 3 seconds. For a vector `A`, each `j`-th column of `R<.A` is `R<.A[j]`:

```
     2 2 2 2<.3 15 8
0 1 1
0 1 0
1 1 0
1 1 0
```

In general, `#R<.A` ←→ `(#R),#A`.

The ***decode*** function `R>.A` is the inverse operation of *encode*. We see that

```
     24 60 60>.1 2 3
3723
     (3#10)>.0 1 2
12
     (4#2)>.1 1 0 1
13
     2 2 2 2>.4 3#0 1 1 0 1 0 1 1 0 1 1 0
3 15 8
```

`R>.A` reduces the rank of `A` by 1 and the length of the first axis of `A` must equal to the length of `R`.

Eli has two sorting functions ***grade up*** `<V` and ***grade down*** `>V` for vector `V`, `<V` is a *permutation* of the indices of `V` so that `V[<V]` is in a *non-decreasing* order. If `V` is of alphabet characters, it is the alphabetic order.

```
     <'CEA'
3 1 2
     V<-52 84 4 6 53 68 1 39 7 42
     <V
```

```
7 3 4 9 8 10 1 5 6 2
     V[<V]
1 4 6 7 39 42 52 53 68 84
     cc<-'cbA12df'
     cc[<cc]
12Abcdf
```

The grade down function `>V` similarly yields a permutation of indices of `V` so that `V[>V]` is in a non- increasing order. ***Note***: *negation* of *grade down* must be written as `- >`, not as `->` which is designated for the ***branch*** symbol. Usually, no blanks are required between primitive functions, or a primitive function and a literal/variable similar to APL. This case is a rare exception.

1.10 execute, format and other mixed functions

The ***execute*** function `!.S` simply executes a character string `S` which represents a line of legal Eli code.

```
     v<-10
     !.'v<-10*.2'
     v
100
```

It can also turn string of digits into a number and a string which is preceded by a "`" into a symbol:

```
     !.'123'
123
     !.'`abc'
`abc
```

If there is any error during execution of `S`, relevant error message will be displayed. Why should there be such a primitive function, can't we just type `S` into our console? It is important to realize that an Eli session is either in ***execution*** *mode* or in ***function edit*** *mode*; so far for convenience, all our presentations are in execution mode. Hence the function `!.` looks superfluous; but when `!.S` is embedded in a function text with `S` possibly a variable, it would make *execute* a very powerful tool.

So far character data and numbers cannot be mixed together; what if we want to write a report with numeric results? The *monadic* ***format*** function `+.A` when applies to a numeric or a symbol data `A` turns `A` into its display in characters:

```
     #D<-+.1 2 3
5
     'A',D,'B'
A1 2 3B
     +.`abc `ddl `comp
abc ddl comp
     #+.`abc `ddl `comp
12
```

If we want to display a group of numbers in column form, we need to reshape it into a one column matrix first. To display the six states' students' distribution we calculated in sect. 1.6, we do

```
     ST6,' ',+.6 1#3 2 3 4 2 1
CT 3
MA 2
NJ 3
NY 4
```

```
PA 2
RI 1
     +.3 1#`abc `ddl `comp
abc
ddl
comp
```

The dyadic format function `P+.A` takes a left argument `P` to specify how the formatting should be done in detail. At present this is not implemented in Eli yet.

Random number generator is quite useful. For positive integer `N`, the *monadic* function ***roll*** `?.N` *randomly* picks a number from `!N` (so it depends on `[]IO`)

```
     ?.100
14
```

The function also depends on a seed, the system variable `[]RL` which is set to 16807 in a *`CLEAR workspace`*. For a vector `V` of positive integers of length `l`, `?.V` is carried out by executing `?.V[i] l`-times. In particular, if we set `[]IO=0`, then we have a convenient way to generate a bits string of certain length.

```
     ?.100 1000
76 459
     []IO<-0
     ?.16#2
1 0 0 1 1 1 0 1 1 0 0 1 1 0 0 0
```

For the *dyadic* function ***deal*** `A?.B`, both `A` and `B` must be positive integers with `A<=B`; `A?.B` picks `A` *distinct* numbers from `!B` (please **remember** there is a '**.**' after '`?`' for random number generators)

```
     10?.100
41 68 58 93 84 52 9 65 41 70
```

The monadic form of `|.A` is the ***factorial*** *function*, and the dyadic `A|.B` is the ***binomial*** *function*, that of taking `A` distinct items from a collection of `B` items:

```
     |.6
720
     5|.7
21
     |.5 8
120 40320
     7|.10
120
```

We note again that there is a **.** after **|** compared with common mathematical notations ! for factorial and binomial. Also, we see that the *factorial* function |. is actually a scalar function.

The monadic (mixed) function ***type*** denoted by `:x` gives out the numeric designation of the type of its argument; `:x` is 0 if x is boolean, 1 if x is integer, 2 (4) if `x` is a floating point (complex) number, 3 if `x` is character, and 5 if `x` is of symbol type, etc., `x` needs not be a scalar. This function is useful in some applications.

There is another pair of primitive functions, the monadic and dyadic functions of the *domino* symbol `#.` is for ***matrix inverse*** `#.M` and ***matrix divide*** `A#.M` which is useful for solving system of linear equations.

```
     m<-3 3#4 6 0 3 2 2 1 3 4
     m
4 6 0
3 2 2
1 3 4
     #.m
_3.84615e-002  0.461538 _0.230769
 0.192308    _0.307692  0.153846
_0.134615     0.115385  0.192308
     m+:* #.m
1           0           0
0           1           0
0           0           1
```

(*some of the 0s are not exact due to precision). *Matrix divide*, the dyadic form of `#.`, is a generalization of matrix inverse, which can be seen as a matrix divide with the left operand being the identity matrix. For `A#.B` to be defined, both `A` and `B` must have rank no more than 2, and `A#.B` is defined to be that of a matrix of shape `(1!.#B),1!.A` which minimizes `+/,(A-B+:*A#.B)*.2` (columns of `B` must be linearly independent, if `A` or `B` is a vector, then it is regarded as a one column matrix). For two vectors x and y, to compute regression using least-square method, we simple do

```
     y#. x.:*. 0 1
```

The abundance of primitive functionss in Eli may need a bit of time to get used to for C/MATLAB programmers. However, providing commonly used functions as language primitives not only relieves one the need to remember many library function names but offers consistency and clarity and encourages a *dataflow* style of programming as we shall see in examples in the next chapter. Since most primitives in Eli are array primitive, that is, they apply directly to arrays; programming solutions in Eli are often thru arrays, not iterations thru loops as in C/FORTRAN.

2. Lists and related Operations

2.1 lists, enclose and raze

The arrays we dealt with so far are all *homogeneous*, i.e. the elements in an array are all of the same type, *numeric*, *character* or *symbol*. What can we do if we want to operate on *non-homogeneous* data, or data which are not rectagular? Eli provides *lists* to organize such data: a ***list*** is a group of *item*s, each of which can be a scalar, an array, or another list, separated by '`;`':

```
     a<-(`abc `ddl `comp;1 2 3)
     a
<`abc `ddl `comp
<1 2 3
     #(`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
     L1<-(`a `b;L)
     L1
<`a `b
<<`abc `ddl `comp
 <1 2 3
```

We see that a *list* can be *counted* and *indexed* like a vector. Most importantly, a list `L` can be assigned to a *group* of variables all at once, where the number of variables is equal to `#L`. A *list* can be entered with *items* containing expressions or another list:

```
     a<-2 3 4
     (a+1;$'avc';`rr `bb)
<3 4 5
<cva
<`rr `bb
     (2 3 4;(`ab;'abc');;!10)
<2 3 4
<<`ab
 <abc
<
<0 1 2 3 4 5 6 7 8 9
```

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
     Lm<-<.(!3;'abc';`x `y)
     Lm
<<1 2 3
```

```
 <abc
 <`x `y
```

Lists can be *catenated* and have *indexed assignments*

```
     #L0<-L,<.'abc'
2
     L0
<1  2  3  4
 5  6  7  8
<abc
```

reshape **does not** work for lists, but a special case of *reshape* works for allocating a list of `n` *empty* items (denoted by the *underline* symbol _)

```
     #Ln<-3#_
3
     Ln[1 2]<-L0
     Ln
<1  2  3  4
 5  6  7  8
<abc
<
```

take and *drop* work on lists as expected:

```
     3^.L
<1  2  3  4
 5  6  7  8
<
<
     1!.Ln
<abc
<
```

The ***first*** function `^.` also applies to a list `L`, i.e. `^.L` is `L[[]IO];` for `L` as above, we have

```
     ^.L
1  2  3  4
5  6  7  8
```

Hence, if a list `Ll` which has only one item, the function `^.Ll` serves as the ***disclose*** of `Ll.` i.e. the inverse of the ***enclose*** function.

For a more general list `Ll` which is homogeneous, i.e. as defined *recursively* all items in `Ll` are all of the same type, either *numeric*, *character* or *symbol* (and later *temporal*), then the monadic function ***raze*** `,.Ll` is defined to be a vector resulting from concatenating recursively all items in Ll. So *raze* turns a list into a vector if it is homogeneous. For example,

```
     l<-(1;2 3)
     L<-(2.1 3.5 6;7 8;l)
     L
<2.1 3.5 6
<7 8
<<1
 <2 3
     ,.L
2.1 3.5 6 7 8 1 2 3
```

```
     chl<-('ABC';'GH';'WW225E')
     ,.chl
ABCGHWW225E
```

2.2 partition, partition count and grouping

enclose turns a scalar or an array into an one item list. To turn a vector into a more general list, ELI has a dyadic primitive function ***partition*** `y||x`, where `x` is the source vector, and `y` is a positive integer or a vector of non-negative integers specifying how `x` is going to be partitioned into a list. If `y` is a single number, then `x` is cut into a list with each item has `y` elements from x sequentially, except the last item which may have less than `y` elements from `x` if there are less than `y` elements in `x` left to fill. For example,

```
     w<-11?.100
     w
14 76 46 54 22 5 68 94 39 52 84
     3||w
<14 76 46
<54 22 5
<68 94 39
<52 84
```

If `1<^y`, then `x` is cut into a list `L` such that *i-th* item of `L` has the next `y[i]` elements of `x`, `i=1..^y`:

```
     3 4 5 || w
<14 76 46
<54 22 5 68
<94 39 52 84
```

If `x` is a matrix, or an array in general, then the partition `y||x` cuts `x` along the first axis according to `y`:

```
     m<-5 4#!20
     2 3 1||m
<1 2 3 4
 5 6 7 8
< 9 10 11 12
 13 14 15 16
 17 18 19 20
<
```

The monadic form of `||` is the ***partition count*** function; its argument `x` is a boolean vector starting with a `1` and its result `||x` is an vector of integers where each element is the length of a segment starting with `1` and ending with the last `0` before the next `1`.

```
     ||1 0 0 1 0 0 0 1 0 0 0 0
3 4 5
     (' This is an example'=' ')
1 0 0 0 0 1 0 0 1 0 0 1 0 0 0 0 0 0
     ||(' This is an example'=' ')
5 3 3 8
     (||(' This is an example'=' '))||' This is an example'
< This
< is
< an
< example
```

Another primitive function which produces a list based on a vector `x` is ***grouping***: `>.x` is a list of indices of `x` such that each item in the result consists of indices of elements in `x` corresponding to unique values in `x`, i.e. `(#>.x) = #=x`, `(>.x)[i]` is `?x=(=x)[i]` for `i` from `1` to `#=x`. For example,

```
     stk
`ibm `appl `ibm `hp `goog `hp `ibm `ibm `appl `ibm `hp `goog `hp `ibm `ibm `appl `ibm `hp
     il<->.stk
     il
<1 3 7 8 10 14 15 17
<2 9 16
<4 6 11 13 18
<5 12
```

For a vector `V` and a list `I` whose elements are legitimate indices of `V` (i.e. for all `i` in each item `I[j]`, `V[i]` is well-defined)

```
        V[I]
```

is a list of the same structure as that of `I`, i.e. `V[I][j]` is `V[I[j]]` for each `j` in `!#I`.

```
     V
14 76 46 54 22 5 68 94 39 52
     I<-(1 2;3 4 5;6 7 8 9;10)
     I
<1 2
<3 4 5
<6 7 8 9
<10
     V[I]
<14 76
<46 54 22
< 5 68 94 39
<52
```

Suppose that

```
     prc
109.1 523.6 107.2 37 358 35 102.3 103.2 551 99.8 33 370.1 31.2 98.9 101.9 588 100 30.5
```

is a vector of corresponding trading prices of companies whose stock symbols appeared in stk above; then the following

```
     prc[>.stk]
<109.1 107.2 102.3 103.2 99.8 98.9 101.9 100
<523.6 551 588
<37 35 33 31.2 30.5
<358 370.1
```

is a list of stock prices *grouped by* trading symbols. With *each* operator in the next section, we have

```
     prices<-prc[>.stk]

^"prices
<8
<3
<5
<2
```

2.3 the each operator

The ***each*** operator ⍤ operates on lists, or a scalar/array and a list. For a *monadic* function *f*, and a list `L`, `f⍤L` is evaluated by applying `f` to each component `L[j]` of `L` to result in a vector or another list `Z` of the same structure as that of `L`: `#f⍤L` is of the same (length) as `#L` and each element of `Z[j]` is `fL[j]` which must be well-defined for all `j`. For `V` and `I` as in the previous section, we have

```
     V1←-V-22
     W←-V1[I]
     *⍤W
<¯1 1
< 1 1 0
<¯1 1 1 1
< 1
```

Since this result is homogeneous, we can apply *raze* to it:

```
     ,.*⍤W
1 1 1 1 0 1 1 1 1 1
```

We note here that the function `f` associated with operators we discussed earlier must be scalar functions. In case of the *each* operator, `f` can be a mixed function, a derived function or even a defined function (to be introduced in the next chapter). For example,

```
     ,.#⍤W
2 3 4 1
```

For numeric `L`, `+/⍤L` is a vector of sums of each item in `L`:

```
     ,.+/⍤W
46 56 118 30
```

`+\⍤L` is a list of same structure as that of `L` with each item the partial sum of the corresponding item of `L`. `avg⍤L` is similarly defined for a defined function `avg`.

For a *dyadic* function `g`, a scalar `s` and a list `L`, `s g⍤ L` is defined by applying (`s g`) to each item of `L` as (`s g L[i]`). For example, `1+⍤L` adds a `1` to each items in `L`; `1^.⍤L` will result in a vector or list, depending on whether `L` is homogeneous or not, of length `#L` consisting of all first elements of items in `L`. For an vector `a` of the same length as that of `L`, `a g⍤ L` is defined if `a[i] g L[i]` is defined for each `i` in `!#L`. For an one-item list `l`, `l g⍤ L` is defined if `(^.1) g L[i]` is defined for each `i` in `!#L`. `L g⍤ r` is similarly defined for a scalar/vector or one item list `r`. For two lists `L1, L2, L1 g⍤ L2` is defined if

```
L1[i] g L2[i]
```

is defined for each `i` in `!#L1` where `(#L1)=#L2`, and as in the monadic case, `g` can be a mixed function, derived function or a defined function, but for each `i, L1[i] g L[i]` must be well-defined. We have

```
     L←-(1 2;3 4 5)
     1+⍤L
<2 3
<4 5 6
     L2←-(7 8;1 2 3)
     L2+⍤L
```

```
<8 10
<4 6 8
     #"L
<2
<3
     (2 3;5;2 3 4)#"!6
<1 2 3
 4 5 6
<1 2 3 4 5
<1 2 3 4
 5 6 1 2
 3 4 5 6

 1 2 3 4
 5 6 1 2
 3 4 5 6
     L*"2
<2 4
<6 8 10
```

Suppose we want to produce a group of 4 consecutive integers but with different starting points, we do

```
      0 _1 1 2+"<.!5
<1 2 3 4 5
<0 1 2 3 4
<2 3 4 5 6
<3 4 5 6 7
```

Note that it is important to *enclose* `!5`, and in the each reshape below *enclose* `2 3` is necessary:

```
     a<-(1 2;3 4 5;7)
     2^."a
<1 2
<3 4
<7 0
     _1!."a
<1
<3 4
<
     10,"a
<10 1 2
<10 3 4 5
<10 7
     a,"10
<1 2 10
<3 4 5 10
<7 10
     #"a
<2
<3
<
     ^"a
2 3 1
     3#"a
<1 2 1
<3 4 5
<7 7 7
     2 3#"a
length error
     2 3#"a
        ^
     (<.2 3)#"a
```

```
<1 2 1
 2 1 2
<3 4 5
 3 4 5
<7 7 7
 7 7 7
     (<.2 3)?"a
<1 0
<0 1
<0 0
     a?"<.2 3
<0 1
<1 0 0
<0
     ?"(0 1 0 0 1;1 0 0;0 0 1 0 0 0 1)
<2 5
<1
<3 7
     6_."a
<1 2
<3 4 5
<6
```

The *each* operator can also apply to *derived functions* as well as *defined functions* (see next chapter).

```
     +/"a
<3
<12
<7
     ~./"a
<2
<5
<7
```

2.4 temporal data

There are six types of ***temporal data*** in Eli to handle date and time: **`date, time, month, minute, second, datetime`**:

```
     d<-2012.04.02              //type date
     d
2012.04.02
     d+1
2012.04.03
     t<-14:37:07.123            //type time
     t+100
14:37:07.223
     23:11:12+10*1 2 3          //type second
23:11:22 23:11:32 23:11:42
     dm<-2012.04m               //type month
     dm+1 2 3
2012.05m 2012.06m 2012.07m
     2012.04.02T14:37:07.123+55 //type datetime
2012.04.02T14:37:07.178
     23:30+30                   //type minute
00:00
     23:30-15
23:15
     23:15<22:10
0
```

We see that addition and subtraction to temporal data are interpreted according to their type, and no other arithmetic function applies to temporal data. Furthermore, unlike numeric data, both operands temporal data must be of the same type. This last rule also applies to relational functions comparing two temporal data. The format function `+.` can be used to convert a temporal data into its character representation:

```
     #dtc<-+.2012.04.02T14:37:07.178
23
     10^.dtc
2012.04.02
     11!.dtc
14:37:07.178
```

In fact, there are two functions in based on this, `dt2d` and `dt2s` which converts a datetime to its character representation in date and second while `dt2dat` and `dt2sec` which converts it into type *date* and *second* in the *standard* library (see **sect.4.5**); and a function there to convert a date after 2000.01.01 into a weekday:

```
     t
2013.06.20T19:00:16.478
     dt2d t
2013.06.20
     dt2s t
19:00:16
     dt2dat t   //a scalar
2013.06.20
     dt2sec t   //a scalar
19:00:16
     dat2wdc dt2dat t
Thu
```

There are also functions converting date or seconds into numeric vectors in the *standard* library:

```
     dt2ymd t
2013 6 20
     dt2hms t
19 0 16
     #dt2ymd t
3
     #dt2hms t
3
```

Temporal data are used for storing time stamps in database systems. We note that APL systems in general, other than **kdb**, only provide a system function `[]TS` to record a time stamp, not temporal data types as data of first class citizen.

3. Defined Functions and Sequence Control

3.1 defined functions and evaluating one line

In the last chapter we introduced all primitive functions and operators provided by the ELI system. In this chapter, we shall describe how a user can *define* one's own function. To define a function, you must be in ***edit mode***; to switch from *execution mode* to *edit mode*, you type in the *del symbol* `@.` followed by a *name* for the function you want to define

```
     @.add
```

The system switches to edit mode by creating a new window in Microsoft Windows, and you type in the definition of your function

```
[0]  z<-a add b
[1]  z<-a+b
[2]  @.
```

type another `@.` when you finish defining your function and the system switches back to ***execution mode***. In Linux/Mac OS, you can type either just the *del* symbol `@.` or `@.` followed by the whole function head to switch to edit mode. Please note that since Linux/Mac OS version is line-based, once you typed in your head line you cannot go back to make corrections; in the first case you start defining your function by typing in the head line, and in the second case you continue by typing in the first line of your function. When you finish your function definition, just as in Windows, types a matching `@.` in a new line to switch back to ***execution mode***. We can check to see whether the function is there, see its' definition and try it out.

```
     )fns       (to see all defined functions that are available in this workspace)
add
     3 add 4
7
     {add}
---------add--------
 [0] z<-a add b
 [1] z<-a+b
--------------------
```

Another way to see a defined function's text is using the system function `[]CR`; the argument to `[]CR` is the function name and the result is the character matrix of the function text:

```
     []CR 'add'
z<-a add b
z<-a+b
     #[]CR 'add'
2 10
```

A *defined function* can take one (right) or two arguments (left and right) just as primitive functions do, i.e. it can be a ***monadic*** or ***dyadic*** function; but a defined function can have no argument, such a function is called ***niladic***. A defined function can return a result or return no result. The result returned by a defined function can be an array as well as a scalar. The first line (line [0]) of a defined function is of the form

function-header <;local-variable-list>

function-header must be one of the six forms below:

type	valence	have result	no result
niladic	**0**	R<-FN	FN
monadic	**1**	R<-FN B	FN B
dyadic	**2**	R<-A FN B	A FN B

where ***valence*** is the number of arguments the function takes, R is the name of the returned result, FN the name of the function, A is the name of the left argument and B is the name of the right argument (you can, of course, give different names to each one of them). The arguments can be arrays and lists, not just scalars. We can use an *explicit list notation* on the right to effectively input more than 2 arguments:

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

In fact, the right argument can consist of several sub-arguments without formally entering it as a list and unpacking its components in the first line of the function. For an example:

```
-------margfn-------
[0]   z<-le margfn (a;b;c)
[1]   []<-c                  //output c
[2]   z<-le+a+:*b
--------------------
     x<-3 4#!24
     y<-4 2#_1 2 3 _5 8 10 1 0
     x
1  2  3  4
5  6  7  8
9 10 11 12
     y
_1  2
 3 _5
 8 10
 1  0
     1000 margfn (x;y;'a test for multi-arg function.')
a test for multi-arg function.
1033 1022
1077 1050
1121 1078
```

The **local-variable-list** in the function header-line is *optional*; it is there only if you wish to ***localize*** a group of variables. The list starts with a ';', and variable names in the list are separated by ';'. If a variable named V1 is *localized* in a function AF, then first during execution of AF, Eli will look for the value of V1 assigned inside AF regardless whether a variable also named V1 exists in the *environment* where AF is called; second, when AF finishes execution the value of V1 obtained in AF will *disappear*. In other word, any variable named V1 in the environment is *protected* by the localization of V1 in AF. We illustrates with the following example

```
@.BF:…
  …
  V1<-1
  AF
  …
@.

@.AF;V1
  …
```

```
  V1<-V1+1
  …
@.
```

Function `BF` calls `AF` after assigns value `1` to `V1`. However, if there is no assignment to variable `V1` before execution of `AF` reaches the line of adding 1 to `V1`, a *value error* will result. After the function `AF` finishes and returns to its caller `BF`, the variable `V1` still holds its old value 1 before the invocation of `AF`. The code in `AF` involving `V1` accesses and modifies its *private* copy of a variable named `V1`. Since the *localization*, or *shadowing*, of variables in Eli depends on the *calling sequence* of functions, it is called ***dynamic binding*** of identifiers. This is in contrast to the static binding of identifiers in PASCAL.

A function's result, if there is any, and function parameters, i.e. arguments, are considered to be localized automatically. Hence, *parameter passing* in ELI is ***by-value***. For example, when the function `FN1`

```
@.Z<-A FN1 B;C
  …
  …
@.
…
Z<-bigA FN1 bigB
```

is called by the line above, where `bigA` and `bigB` are two large arrays, then first these two arrays are copied into `A` and `B` at the start of the function `FN1`, and their values would not be changed at the of the end of the function call. Second, if `FN1`, thru some path, exits without assigning `Z` any value then the line will result in a *value error* after function exit. Copying large arrays in parameter transmission certainly is inefficient. There are programming techniques to put such arrays as global variables, or use sophisticated compiler techniques to avoid copying in case there is no modification to their elements in `FN1`; but we are not going into detail discussion of this issue in the Primer.

With the classification of (user) defined functions done, we can now give a more *formal* definition on how to ***evaluate*** *one line* of code *L* in Eli as follows. Start from the right end of *L*, it must begin with a literal data, a variable or a niladic function with a result. For a variable get its value, or evaluation stops with a *value error*; if it is a niladic function, execute that function and get its return value. In each case, we get a value *cv*. If there is nothing to the left of it, *cv* is the final value of the line *L*; otherwise what to its left must be a primitive function *pf* or a defined function *df* . For a monadic *df*, we execute *df cv* to get a new *cv* and keep going. If *df* is a dyadic function, we look to the left of *df* to get the value *lv* of its left argument; in case there is nothing to the left of *df*, we get a *valence error*. What to the immediate left of *df* must be a literal data, a variable with value, a niladic function with a result or a right parenthesis ')'. In the first three cases, we get *lv* as before and execute *lv df cv* to get a new *cv* and keep going. In case of ')', we recursively evaluate the code inside (…) to get a value *lv* and proceed as in the previous 3 cases. For a *pf*, if *pf* can only represent a monadic functions or a dyadic function, then we do the same as what we described for *df*. If *pf* can either be dyadic or monadic, we decide by checking whether there is literal, variable, niladic function with result or a right parenthesis to the left of *pf* and proceed accordingly.

3.2 one line examples

We have already seen several examples of one line code in chapter 1 in the process of introducing various primitive functions and operators. We shall now add two more examples from the perspective of implementing

specific tasks thru a process of refinement. The strategy is to first layout a function which has the basic idea that works, then make it more efficient, or works for more general case. From these examples, we also gain an understanding of the *dataflow style* of programming (this refers to writing long line of code where output from one expression is immediately used as input to the next expression) subtly encouraged by Eli, and come to appreciate the difference between APL, Q, Eli in one camp and conventional array languages such as MATLAB, R, Octave and SciLab in the other.

Our first example is that of **finding prime numbers up** to N. The idea is to setup a multiplication table of 2..N by 2..N, then numbers not in that table are primes:

```
     {prim}
--------prim--------
 [0] z<-prim n
 [1] z<-(~v?v.:*v)/v<-1!.!n
--------------------
     prim 100
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
```

The program prim is clearly inefficient, but basic idea of having `v` as the vector `2..n` and using `~…?` multiplication table is clear enough. We can improve on several fronts. First, other than 2, all primes are odd numbers; second a smaller multiplication table of square root of n by one third of n should be large enough to cover all numbers up to n. Hence, we have

```
     {prime}
--------prime-------
 [0] z<-prime n;v
 [1] z<-2,(~v?(2!.!_.n*.0.5).:*2!.!~.n%3)/v<-1+2*!_.(n-1)%2
--------------------
     prime 300
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131
137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263
269 271 277 281 283 293
```

Here `v` is the vector of odd number from 3 to near `n`, and `2` is added to the result list after all odd primes up to `n` have been selected from elimination of multiples.

The second example is **word counting**: give a text `TX` which is a matrix of characters consisting of English alphabets, punctuation marks and blanks; we would like to count the number of words in `TX`. We can assume that if a word ends in a punctuation mark not on the text boundary, it is always followed by a blank. Hence to count words in `TX` is somehow related to counting blanks in `TX`. First let us make a simplifying assumption that each word ends in precisely one blank. So, we just count the number of blanks in `TX` by

```
     wc<-+/+/b<-' '=TX
```

where `b` a boolean matrix of the same shape as that of `TX` which has a `1` at each position corresponding to a blank in `TX`. `+/b` is a list of integers indicating the number of blanks in each row of `TX`, and `+/+/b` is the total number of blanks.

The simplifying assumption has three problems: 1) a word may end in the right side boundary, 2) there may be more than one blank following a word, and 3) in the beginning of a line we may have blank(s) to start with. To overcome these problems, we construct another boolean matrix btl by removing the first column of b and gluing

a column of 1's to its right. This has the same effect as putting a column of blanks at the right boundary to end a word but ignoring the first column. A blank which truly ends a word is then a blank whose predecessor is not a blank that is a 1 in btl but a 0 in b. For example

```
     TX
 ABC CFM
N, GH  D
F ABC GF
     b<-' '=TX
     b
1 0 0 0 1 0 0 0
0 0 1 0 0 1 1 0
0 1 0 0 0 1 0 0
     btl<-(0 1!.b),1
     btl
0 0 0 1 0 0 0 1
0 1 0 0 1 1 0 1
1 0 0 0 1 0 0 1
     btl^~b
0 0 0 1 0 0 0 1
0 1 0 0 1 0 0 1
1 0 0 0 1 0 0 1
```

Hence, the function we want is

```
     @.z<-COUNT TEXT;b
      z<-+/+/((0 1!.b),1)^~b<-' '=TEXT
     @.
     COUNT TX
8
```

In contrast to the C program solution in sec.1.5.4 in [3], we see that the style of C programming is *sequential* in the sense the solution is reached through a loop while in Eli it comes from a calculation of global characteristics. The Eli approach is what we call *array oriented programming*. It takes time to gain proficiency in writing this style of code for a person comes from C/C++ but it accrues advantage in parallel processing which we will not going into details to explain here.

3.3 transfer of control by branching and stopping

So far we have only seen only examples of straight-line code, and presumably we can execute one line after another in a multi-line function code. How do we write code with alternate choices and loops? A simple mechanism to do that in Eli is ***branching*** which is of the form

```
-> branch-expression
```

When we execute a function `F` line by line in their *line number (textual) order* until we hit a branch expression; we evaluate that expression just like we evaluate any other Eli line. If the expression evaluate to an empty vector, execution continues with the next line in F. If it is 0, i.e.

```
-> 0
```

or any number out of range of `F`'s line numbers, function `F` returns to its' caller which can be Eli system or another function `G`. If it is the *line number* of a statement in `F`, that statement is to be executed next. A statement

in a function text can be prefixed by a label, and that label has the value of the line number of that statement. For example

```
[5]LABEL1: X<-X+1
```

The label `LABEL1` has value 5. Frequently, one uses the following form

```
->(~BE)/ELSE
…
ELSE:…
```

to jump to the `ELSE` line in case `BE` is false. Of course, `ELSE:` can be a line precede the branch line thus forms a *while* or *for* loop. In general, a way to do multiple branching is as follows

```
->(BEV)/L1,L2,…,Ln
```

where `BEV` is a boolean vector and the branch goes to the line label `Lk` if k-th bit is the first set bit of `BEV`. Another frequently used branching pattern, assuming `[]IO=0`, is

```
->(L0,L1)[BEX]
```

which branches to either `L0` or `L1` depending on whether `BEX` is false or true.

If the ***branch-expression*** after `->` is a *character string* instead of a line number, the execution would ***stop*** after display the character string. For example,

```
->'L0 and L1 are not equal'
```

Note that this differs from output the message and then `->0`. For in the second case, execution would continue after return of the function where the message is triggered.

3.4 control structures

For numerical computations, where need of branching is infrequent once many loops would have been eliminated by array primitives, transfer of control by *computed goto* is tolerable. But for implementing complex systems such as a modern compiler that is quite inconvenient. *Eli has ***control structures*** with seven ***reserved words***:

`if` `else` `case` `for` `while` `break` `continue`

and we define a statement in Eli as follows:

statement: *simple-statement* *or* {*list of statements*}
if-statement: **`if`** (*bool expression*) *statement* [[**`else if`** (*bool expression*)]" **`else`** *statement*

case-statement: **`case`** (*case expression*) {*case-lists* [**`else`** *statement*]}
case-list : v1[*,v2..vn*]*: statement*
for-statement: **`for`** (*idxv*:*for-forment*) *statement*
for-forment: *strv*;*endv*[;*step*] or *idxlist*

while-statement: **`while`** (*bool expression*) *statement*

`break` and **`continue`** can only present inside a **`for`** statement or a **`while`** statement, and their effects are to *exit* the enclosing statement or to *start* a new iteration. *case expression* must be an expression of discrete scalar type, i.e. it must result in an integer, a character or a symbol:

```
@.z<-tst_cas x
  case (:x) {
    0,1:z<-x+10
     2:z<-x*10
     3:z<-x
     5:case (x) {
       `select:z<-'SELECT'
       `exec:z<-'EXEC'
       else z<-'***'
       }
     else z<-x+100
  }
@.
     tst_cas 9
19
     tst_cas 1.2
12
     tst_cas 'ABC'
ABC
     tst_cas `ABC
***
     tst_cas 20:55
22:35
```

A partial ordering **<** on a set `S` is a relation among elements of `S` such that

```
a, b, c in S,     a<b, b<c →   a<c
```

Let us introduce an ordering among nodes of a directed graph `G`:

```
I<J if G[I;J]=1
```

This is a partial ordering; a **topological sort** of `G` then is a linear ordering of `G` compatible with that partial ordering. An algorithm (from Wirth[4]) to do a topological sort is the following:

L ← Empty list that will contain the sorted elements
S ← Set of all nodes with no incoming edges
while ***S*** is non-empty **do**
    remove a node ***n*** from ***S***
    insert ***n*** into ***L***
    for each node ***m*** with an edge ***e*** from ***n*** to ***m*** **do**
        remove edge ***e*** from the graph
        if ***m*** has no other incoming edges **then**
            insert ***m*** into ***S***
if graph has *e*dges **then**
    output error message (graph has at least one cycle)
else
    output message (proposed topologically sorted order: ***L***)

We note that for a node `N`, `G[;N]` indicates all nodes with edges to `N`, and `G[N;]` all nodes having edges from `N`. The code in Eli is as follows

```
[0] L<-TSORT G;N;ML
[1] L<-!0
[2] S<-(0=+/.G)/GL<-!1^.#G
[3] while (0<#S) {
[4]   L<-L,N<-S[1]
[5]   S<-1!.S
[6]   ML<-G[N;]/GL
[7]   G[N;ML]<- 0
[8]   S<-S,(0=+/.G[;ML])/ML
[9]  }
[10]if (0<+/+/G) []<-'Graph G is not acyclic.'

     G
0 1 0 1 0 1 0 0
0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 1
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 1
0 0 0 0 0 1 0 0
     TSORT G
3 5 7 1 8 2 4 6
```

We remark here that `[]CR 'fn1'` always displays a defined function `fn1` as entered while {`fn1`} only displays a function after the function `fn1` has been executed at least once and where control structure statements have already been translated into standard branch forms. For example,

```
     )fload TSORT
saved 2012.11.01 23:05:56 (gmt-5)
     )fns
TSORT
     []CR 'TSORT'
Z<-TSORT G;N;ML
L<-!0
S<-(0=+/.G)/GL<-!^.#G
while(0<#S){
L<-L,N<-S[1]
S<-1!.S
ML<-G[N;]/GL
G[N;ML]<-0
S<-S,(0=+/.G[;ML])/ML
}
if(0<+/+/G)Z<-'Graph G is not a cyclic'
else Z<-L
     {TSORT}
--------TSORT-------
--------------------
     )vars
G
     TSORT G
3 5 7 1 8 2 4 6
     {TSORT}
--------TSORT-------
[ 0]  Z<-TSORT G;N;ML
[ 1]  L<-!0
```

```
[ 2]  S<-(0=+/.G)/GL<-!^.#G
[ 3]  L0:->(~0<#S)/L1
[ 4]  L<-L,N<-S[1]
[ 5]  S<-1!.S
[ 6]  ML<-G[N;]/GL
[ 7]  G[N;ML]<-0
[ 8]  S<-S,(0=+/.G[;ML])/ML
[ 9]  ->L0
[10]  L1:->(~0<+/+/G)/L2
[11]  Z<-'Graph G is not a cyclic'
[12]  ->L3
[13]  L2:Z<-L
[14]  L3:
--------------------
```

3.5 recursive functions

Eli support ***recursive*** functions. An example is that of `rprime`: to find prime numbers up to N recursively.

```
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
}

     rprime 300
2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103 107 109 113 127 131
137 139 149 151 157 163 167 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263
269 271 277 281 283 293
```

What this function does is that it first stores the primes up to `100` in `p`, and then it checks the size of the parameter `n`. If `n` is less than `100`, then just select primes from `p`; otherwise recursively calls `rprime` with square root of `n`. Then take out multiples of primes found so far. This is a much more efficient function than our earlier `PRIME` function even though it is recursive. This function also illustrates how control structures improve program readability when there are alternate choices and irregular iteration, while ELI boolean array operations still support a succinct dataflow style coding.

Finally, we recall that we stated in **sect**.2.3 that the *each* operator ″ operates on defined functions and lists. For example,

```
     rprime"(10; 100; 200)
<2 3 5 7
<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
<2 3 5 7 11 13 17 19 23 29 31 37 41 43 47 53 59 61 67 71 73 79 83 89 97 101 103
     107 109 113 127 131 137 139 149 151 157 163 167 173 179 181 191 193 197 199
     {add:x+y}
add
     (3;4) add" (6 7 9;10 11)
<9 10 12
<14 15
```

4. System Facilities

4.1 save and reload your work*

You can save your work by rename your *workspace* and then save

```
     )wsid NEW1
     )save
saved 2011-08-18 11.30.26
```

Once you saved your workspace, you can reload it at a later time

```
     )load NEW1
```

A workspace you saved may contain errors or intentional stops, and this is indicated by the ***state indicator***

```
     )SI
```

For example, if you finished writing a function `FNT`, and when executed it results in an error at line 3.
Then the result of the system command above is

```
     FNT[3]
```

When you load such a workspace, you can resume the work you left and start debugging.

Unlike in APL, a saved workspace cannot be copied into an existing workspace with a command

```
     )copy NEW1
```

To achieve the effect of a `)copy`, i.e. replacing any variable or function in the existing workspace with the one copied in when there is a name collision, we use `)fcopy file1` to be explained later in this section. In particular, system variables such as `[]IO` would be refreshed by the one exists in the copied in file. Please note that ELI **does not allow** a *variable* to be replaced by a *function* with the same name, nor a *function* replaced by a *variable* of the same name from another workspace.

You can also output the functions and variables in a workspace, or some portion of it, to a file; but your workspace's suspensions, i.e. `)SI` indicating unresolved bugs, will not be copied over.

```
     )out file1
     )out file2 FN1 FN2 A BC
```

In the second case, only those items listed after the file name `file2` would be sent to the output file. These so called ***eli script files*** have file extension **.esf* while that of workspaces have extension **.eli*. For the same group of variables and functions, an *esf* file is noticeably smaller than an *eli* file. An *esf* file, or portion of it, can be copied in or loaded with the system commands

```
     )fcopy file1
     )fcopy file2 FN1 FN2 A BC
     )fload file1
```

In Windows, after installing ELI, there would be a subdirectory `eli` created in your `Program Files` directory which likely resides in `C` disk; `eli` has two subdirectories, `bin` and `ws`. The Eli executable resides in `bin`.

Saved workspaces, outed or edited files and `esf` script files all reside in `ws` directory, and one need to put a prepared (text) file into the `ws` directory before one can `)fload` or `)fcopy` it. In Linux/Mac OS, `bin` and `ws` are combined into one directory: `elix/elim`. To see all workspaces and script or text files that are there, you type `)lib`

```
     )lib
RPRIME.esf    TSORT.esf    standard718.esf standard.esf  edb828.esf    edb828.eli    dbdata.txt
dbdata.eli    edb.esf  edb.eli  test.eli
```

To see only those workspaces or files starting with specific characters, you type

```
     )lib ed
edb828.esf    edb828.eli    edb.esf      edb.eli
```

4.2 loading and copying script files

You don't need to input your functions in the edit mode of Eli; nor is it convenient or practical to input large variables through an interactive ELI session. You can prepare as an ordinary text file, or what usually call a script file, of extension **.esf*. (extension **.txt* also works if no *.esf* file of the same name exists). The script file can contain not only function definitions as in RPRIME example we see in the last chapter but also values for variables. To do that you put

```
&A I 2 50 80
 …
&
```

The first line is the variable name followed by a character denoting its *type* (`B`:boolean; `I`:integer; `E`:float; `J`:complex number; `C`:character; `S`: symbol, `D`: temporal data, `L`: list; we note here for an item `v` in Eli the monadic function `type :v` gives an integer assigned to that type from 0,1,2,4,3,7,6 corresponding to the order above) and the *rank* and *shape* of that variable, then followed by the value of the variable in *ravel* order for all arrays. For a scalar, instead of *rank* and *shape,* one writes a `0:`

```
     &a E 0
     0.5…
     &
```

However, the specification for a list is *recursive* since it is not a simple array or scalar, i.e. each item in the list needs to be specified one after another. For example, we have file `ts2.txt` as follows

```
&lt L 1 3
I 2 2 3
1 2 3 4 5 6
C 1 5
'abcde'
E 1 1
0.5
&
     )fcopy ts2    //copy ts2.esf into ELI interpreter
     lt
1 2 3
4 5 6
abcde
0.5
```

For a function FN1 with two arguments A, B and result Z, it is written similar to what we type in function edit mode

```
@.Z<-A FN1 B;C
  …
@.
```

If FN1 calls FN2, definition of FN2 must appear before FN1 if FN1 is in short form (*see below). To load in a file named S1, type

```
)fload S1
```

You can even put in executable statement such as 'RPRIME 120' in a script file; that statement would then be executed at the time of loading (or copying).

There is a similar command with respect to copy in a script file S

```
)fcopy S
```

It differs from)fload in that it would not start from a CLEAR workspace. Existing functions and variables in the current workspace would remain unless there are functions and/or variables with the same names exist in S. In this case, they would be replaced by those from S. An *esf* file can also include other system commands such as)load …,)fload …,)fcopy ….

ELI also provides a ***short-form*** function definition facility as follows

```
{fnam: …}
```

where fnam is the name of a function and either z or the last expression is the result of the function, while x is the right argument, and y is the left argument if present; all other variables are assumed to be local. All statements must be a *simple* or *if* statement; two statements can be separated by ';' in one line. Statements in short-form definition **cannot access** any global variable other than system variables. Also comment line(s) must be outside of the function body {..}. Moreover, a short-form definition of a defined function can be entered in *execution mode* of ELI. For example

```
//average of a numeric vector
{avg: (+/x)%#x}
```

For a more interesting example, let x be an annual interest rate in percent, y be the length of years of a loan, P the principal of a loan. Then

J = monthly interest rate = x/(12*100)
N = number of months over loan = y*12
M= monthly mortgage payment = P * J/[1-(1+J)$^{-N}$]

```
//monthly payment for a loan of one dollar with annual interest rate of x% over y years
{mthpaym1: J%1-(1+J<-x%12*100)*.-y*12}
```

Then for a loan of P amount, the monthly mortgage payment at rate x over y years is

```
M<-P*y mthpaym1 x
```

4.3 input and output

We have already discussed input and output files with functions and variables. But in ELI, for simple output, we can do

```
      []<-'Hello World!'
Hello World!
```

and an appearance of `[]` in code will *wait* for user input. '`[)`' is for *bare-output*, i.e. the next output will not start from a new line.

4.4 pre-defined functions in the standard script

ELI system provides a script `standard.esf` which contains many frequently used functions. Hence,

```
     )fcopy standard
```

will copy in the script file, then a group of pre-defined functions are avalable. For example, they include

```
{avg: (+/x)%_1^.#x}                 //average of a num. vector or row-wise average of an array
{median: ((0.5*w[m]+w[m+1]),w[m<-_1+[]IO+~.0.5*#x])[[]IO+2|#w<-x[<x]]}
{gmean: (*/x)*.%#x}                 //geometric mean of a numeric vector
{stddev: ((+/(x-avg x)*.2)%#,x)*.0.5} //standard deviation of a vector
{intersect: (y?x)/y}                //y intersects x, those of y which are in x
{union: x,(~y?x)/y}                 //y union 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
{trim: triml trimr x}               //trim leading and trailing blanks
{diag: x*(!1^.#x).:=!_1^.#x}        //get the diagonal matrix of matrix x
```

One can study the `standard.esf` file to see the functionalities of the pre-defined functions, or do a copy to bring in all processed pre-defined functions in the `standard` file. For example,

```
     )fcopy standard
    avg 3 5 78 9
23.75
     median 3 5 78 9
7
```

For character strings, we have

```
     lower 'ABC'
abc
     upper 'abc'
ABC
```

The call `s substr x` takes a character string s, and produces a substring located at `x[1]` of length `x[2]`

```
     'abcdefghijklmn' substr 5 6
efghij
```

To get up the diagonal, upper or lower triangle part of a matrix `m`, we do:

```
     m<-5 5#!25
     diag m
1 0  0  0  0
0 7  0  0  0
0 0 13  0  0
0 0  0 19  0
0 0  0  0 25
     triu m
1 2  3  4   5
0 7  8  9  10
0 0 13 14 15
0 0  0 19  20
0 0  0  0  25
     tril m
 1   0  0  0  0
 6   7  0  0  0
11 12 13  0  0
16 17 18 19  0
21 22 23 24 25
```

To get a `n` by `n` identity matrix, we call `(n=5)`

```
     eye 5
1 0 0 0 0
0 1 0 0 0
0 0 1 0 0
0 0 0 1 0
0 0 0 0 1
```

The function `det m` gives the determinant of matrix `m`

```
     m
 3  5  3  1
 0 11 12  7
_5  6  8 10
 9  3  5  3
     det m
_1428
```

We have the random number generator `?.` which stochastically picks an integer from a range.
Often, we like to generate `n` random numbers from `[0,1]` for that we can call `rand n`:

```
     rand 5
0.131538 0.755606 0.458651 0.532768 0.21896
```

and `randm n` gives an `n` by `n` matrix with entries from `[0,1]`

```
     randm 5
0.047045 0.678865 0.679297 0.934693 0.383503
0.519417 0.830966 0.034573 0.053462 0.529701
0.67115   0.007699 0.383416 0.066843 0.417486
0.686773 0.588977 0.930437 0.846167 0.526929
0.091965 0.653919 0.416    0.701191 0.910321
```

We recall from **sect**.1.8, that the primitive function ^. when applies to a matrix right argument the left argument must contain the length of its first axis (resp. last axis) when we only care about the last axis (resp. first axis). `standard.esf` provides the function `take` (resp. `take1`) to do what we want more conveniently:

```
     m<-3 5#!15
     m
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
     2 take m
 1  2
 6  7
11 12
     _4 take m
 2  3  4  5
 7  8  9 10
12 13 14 15
     2 take1 m
1 2 3 4  5
6 7 8 9 10
     _4 take1 m
 0  0  0  0  0
 1  2  3  4  5
 6  7  8  9 10
11 12 13 14 15
```

For two items `A` and `B`, to glue these two pieces along the first axis in a flexible way, i.e. if they are of different lengths in their last dimension, we pad one with fill elements to make them conformable for catenation, requires a bit work. But there is a pre-defined function `cat` in standard to do just that:

```
     c1
AAAA
AAAA
     c2
BBBBB
BBBBB
BBBBB
     c1 cat c2
AAAA
AAAA
BBBBB
BBBBB
BBBBB
     c2 cat c1
BBBBB
BBBBB
BBBBB
AAAA
AAAA
     100 cat nm<-3 6#!18
100  0  0  0  0  0
  1  2  3  4  5  6
  7  8  9 10 11 12
13 14 15 16 17 18
     nm cat 90+!7
 1   2  3  4  5   6  0
 7   8  9 10 11 12  0
13 14 15 16 17 18  0
91 92 93 94 95 96 97
```

The function `y replic x` replicates elements of `x` according to non-negatives numbers in `y`:

```
     x<-1 8 9 3 2 1 7
     y<-1 1 4 3 2 0 3
     y replic x
1 8 9 9 9 9 3 3 3 2 2 7 7 7
```

The function `y nwise_sum x` gives `y`-wise sum of vector `x` while `y movin_sum x` gives `y`-moving sum of vector `x`, the function `y nwise_avg x` gives `y`-wise average of vector `x`, the function `y nwise_min x` gives `y`-wise minimum of vector `x` while `y movin_min x` gives `y`-moving minimum of vector `x`. We have `y nwise_max x` and `y movin_max x` similarly defined for maximum:

```
     w
14 76 46 54 22 5 68 94 39 52
     3 nwise_sum w
136 176 122 81 95 167 201 185
     3 movin_sum w
14 90 136 176 122 81 95 167 201 185
     3 nwise_avg w
45.33333333 58.66666667 40.66666667 27 31.66666667 55.66666667 67 61.66666667
     3 movin_min w
14 14 14 46 22 5 5 5 39 39
     3 movin_max w
14 76 76 76 54 54 68 94 94 94
```

The function `stddev` is the *standard deviation* of a numeric vector, with `w` as above, we have

```
     stddev w
26.6608327
```

Studying the code in the script file `standard.esf` (most functions are defined in short function form) helps one who is new to ELI to learn how to program in ELI through one-line code as well as using control structures. One can also discover other pre-defined functions not sampled here. We note that each of these functions has implicit requirement on its' input(s). For example, `avg` and `median` only apply to numeric vectors while `lower,upper` and `substr` only apply to character vectors. `det` clearly only takes only numeric matrix argument.

5. Dictionaries and Tables

5.1 dictionaries and enumerations

A ***dictionary*** `D` is a two items list, a *domain* `d` and a *range* `r` of equal count, `(d;r)`, and a correspondence between them set up by the dyadic function ***map*** **:**, `D←d:r`. Once that a dictionary is set, a pair of system functions will return its components: `key(D)` gives `d`, the domain, and `value(D)` the range of `D`. 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. Elements in `d` are called ***keys***; for each key `k` in `d` the *lookup* function `D[k]` will yields the corresponding item, which can be a scalar, an array or a list in `r` as the result. *Note that we assume* `[]IO=0` *in this chapter and the next*. For example

```
     tD←`nodeid `parent`lson `rson: 0 _1 1 2
     tD
nodeid| 0
parent| _1
lson  | 1
rson  | 2
     C←'ABC':((`usd;1;'USA'); (`pound;0.66;'Britain');(`yuan;6.27;'China'))
     M←-1 10 20:('Washington';'Hamilton';'Jackson')
     M
1 | Washington
10| Hamilton
20| Jackson
     key(tD)
`nodeid `parent`lson `rson
     value(tD)
_1 0 1 2
     key(C)
ABC
     value(C)
<<`usd
 <1
 <USA
<<`pound
 <0.66
 <Britain
<<`yuan
 <6.27
 <China
     value(M)
<Washington
<Hamilton
<Jackson
     tD[`nodeid]
0
     C['A']
<`usd
<1
<'USA'
     C['AC']
<<`usd
 <1
 <'USA'
<<`yuan
```

```
 <6.27
 <'China'
     M[10]
Hamilton
```

Hence, dictionary is a generalization of list, instead of indexing by position `0..n`, they are indexed by keys in their domains. Moreover, for a `k` not in `key(D)`, `D[k]` results in an unknown represented by `` `0N ``, not an ***error***.

```
     M[100]
`0N
```

As for lists, for key `k` in a dictionary `D`, `D[k]` can be *indexed assigned*, i.e. its corresponding value can be *updated* if `k` is in `key(D)` or *inserted* if `k` is not in `key(D)` to start with

```
     tD[`rson]<-4
```

As an array can be indexed by a subset of positions to get a sub-array, a dictionary can be indexed by a subset of its domain to get a sub-dictionary with corresponding sub-range: `C['BC']` is the dictionary

```
 'BC':((`pound;0.66;'Britain');(`yuan;6.67;'China'))
```

However, when that subset is a singleton `s` we need to enlist it to get a sub-dictionary with count 1 for both its domain and range

`C1<-C[<.'A']` is the same as `C1<-(<.'A'):<(`usd;1;'USA')`

Since *dictionary* is a generalization of *list* that instead of a map from an integer interval `!n` to `L` it is a map from a general discrete set, the *shape* `#d` of a dictionary `d` is defined as `#key(d)` or `#value(d)`.

Two dictionaries `D1` and `D2` can be *join*ed: `D1,D2`. `key(D1,D2)` is the union of `key(D1)` and `key(D2)` which must be of the same type. If `k` is only in the domain of one dictionary then the corresponding value of `k` in `D1,D2` is the corresponding value of k in that dictionary. If `k` is in both domain, then the corresponding value of `k` in `D1,D2` is that of `D2`, the right operand of the *join*.

If the ranges of dictionaries `D1` and `D2` are numeric and are conformable, then *each* of `f`, `D1 f" D2` is defined for a primitive dyadic arithmetic, logical or relational function `f` just as that for lists with elements belong to the common range of `D1` and `D2`, and this is the same if one of `Di` is a numeric scalar. For a primitive monadic function `g`, `g"D1` is similarly defined; by convention, we drop `"` from `f"` for dictionaries.

```
     tD1<-`nodeid `parent`lson `rson: 1 0 3 4
     tD+tD1
nodeid| 1
parent| _1
lson  | 4
rson  | 6
     tD1+10
nodeid| 11
parent| 10
lson  | 13
rson  | 14
```

We can *drop* keys `ks` and their corresponding values from a dictionary `D`, `ks!.D`; `` `parent`lson!.T `` results in a new dictionary (this feature is missing at present)

```
     `nodeid `rson: 0 2
```

If some element `k0` in `ks` is not in `key(D)`, it would have no effect to the result.

A dictionary `D` whose domain `key(D)` consists of symbols and whose range `value(D)` is a list whose elements are 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***. Suppose `k` is a key, then `D[k]` is a vector, and we can index it by i to get its i-th element, `D[k][i]`, or as in a matrix `D[k;i]` (later we also access this element by `D.k[i]`). An example of column dictionary is the following:

```
     tDc<-`nodeid `parent`lson `rson:val<-0 _1 1 2+"<.!5
     tDc
nodeid| 1 2 3 4 5
parent| 0 1 2 3 4
lson  | 2 3 4 5 6
rson  | 3 4 5 6 7
```

There is an *incarnation* of the ***map*** function **:** when the left operand is the *name* (symbol denoting a name) of a variable `u` whose value is a vector of unique items of symbol type and the right operand `v` is a scalar or vector of symbols all belong to the set represented by `u` which results in an ***enumeration*** instead of a *dictionary*

```
     env<-`u:v
```

(recall that for `lop:rop` to be a dictionary `lop` must be a vector or a list while `` `u `` here is a scalar). With a fixed `u`, the (monadic) function `` `u: `` is called an ***enumeration*** by `u`, `u` is the *domain* of enumeration and `` `u:v `` is called the *enumerated value* over `u` (we assume `[]IO=0` in this section).

```
     u<-`ibm `hp `appl
     v<-`ibm `ibm `appl `hp `ibm `hp `appl `hp `ibm `appl `hp `ibm `appl `hp `ibm `ibm
     ev<-`u:v
`u:`ibm `ibm `appl `hp `ibm `hp `appl `hp `ibm `appl `hp `ibm `appl `hp `ibm `ibm
     v[2]
`appl
     ev[2]
`u:`appl
```

The reason to have *enumeration* instead of just store the variable `v` is that to operate on each item of `v` we need to check each symbol in v globally; with `u: we only need to check each item in `v` against those in `u`. For an enumerated variable the values is actually stored as an index vector of `u`

```
     0 0 2 1 0 1 2 1 0 2 1 0 2 1 0 0
```

We can change an item in `v` and the corresponding `ev` would be changed as well; *index of* and *where* functions give the same results on `v` and `ev`.

```
     v[2]<-`hp
     ev[2]
`u:`hp
     v!`hp
2
     ev!`hp
2
     ?v=`ibm
0 4 8 11 12
```

```
     ?ev=`ibm
0 4 8 11 12
```

Updating an enumeration makes change to some base symbol in `u` automatically forces corresponding changes in `v`.

```
     u[1]<-`ge
     ev
`u: `ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm
     v
`ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm
```

Without enumeration to affect the same change in `v`, we need to do

```
     v[?v=`hp]<-`ge
     v
`ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm
```

On the other hand, to append an enumeration requires bit more work than just append an item to `v`.

```
     v<-v,`gm
     v
`ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm `gm
     ev<-ev,`gm
type error
```

What happened is that `` `gm `` is not a member of `u`; so we need to do

```
     u<-u,`gm
     ev<-ev,`gm
     ev
`u:`ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm `gm
```

To recover `v` from its enumerated value, simply apply the value function as in the case of dictionary

```
     value ev
`ibm `ibm `appl `ge `ibm `ge `appl `ge `ibm `appl `ge `ibm `appl `ge `ibm `ibm `gm
```

Finally we note that enumeration `` `u:v `` still works as above if `u` is relaxed to be a variable of distinct values of a types different from the symbol type.

5.2 tables

The *transpose* of a column dictionary `D` defined as `` `k0 `k1…`kn:(v0;v1;…;vn) `` is called a ***table*** `T<-&.D`, under the hook `T` is still stored as `D` with a mark for output purpose. We can also set up a table directly by

```
     T<-([] k0<-v0;k1<-v2;…;kn<-vn)
```

where `k0,k1,…,kn` are symbols and `v0,v1,…,vn` are vectors (some of them can be a scalar); the value columns `v0,v1,…,vn` can be stored in variables `c1,c1,…,cn` first. Then we can define `T` as

```
     T<-([] c0;c1;…cn)
```

`ki` is simply the symbol of the variable name of `ci, i=0,1,…,n`. Let `D` be the dictionary

```
     D<-`ticker `price:(`ibm `hp `appl;207.2 16.8 457.01)
```

```
     D
ticker| ibm hp appl
price | 207.2 16.8 457.01
     &.D
ticker price
-------------
ibm    207.2
hp     16.8
appl   457.01
```

and `T` the transpose of `D`, then `T` can also be defined as a *table* directly as follows:

```
     T<-([] ticker<-`ibm `hp `appl; price<-207.2 16.8 457.01)
     T
ticker price
-------------
ibm    207.2
hp     16.8
appl   457.01
```

Note that with '`(`' followed immediately by '`[`' it signifies the definition of a table so '`[]`' is not the quad for output and `T` is not a mere *list* but a *table*. In case we have done

```
     c0<-`ibm `hp `appl
     c1<- 207.2 16.8 457.01
```

first, we can define a table as follows

```
     T0<-([] c0;c1)
     T0
c0   c1
-----------
ibm  207.2
hp   16.8
appl 457.01
```

then the column variable names will be used as symbol names for columns.

Records in a table can be accessed like in a list, and column names can be used as indices in a 2-dimensional array; and their values can be updated:

```
     T[;`ticker]
`ibm `hp `appl
     T[2;]
ticker | `appl
price  | 457.01
     T[2;`price]
457.01
     T[`price]
207.2 16.8 457.01
     T[`price]<-T[`price]+1 2 _3
     T
ticker price
-------------
ibm    208.2
hp     18.8
appl   454.01
```

We can also access column values of a table with dot notation commonly used in database systems; and access rows of table as a vector of records forming a table or a dictionary in case of a single row:

```
     T.ticker
`ibm `hp `appl
     T[2]
ticker| appl
price | 457.01
     T[0 1]
ticker price
------------
ibm    207.2
hp     16.8
```

Two tables with similar columns can be *joined* together by the *cantenate* primitive ‘,’:

```
     T2<-&.`ticker `price:(`ibm `hp;198.8 20.1)
     T2
ticker price
------------
ibm    198.8
hp     20.1
     T,T2
ticker price
-------------
ibm    207.2
hp     16.8
appl   457.01
ibm    198.8
hp     20.1
```

In particular, we can *append* a record to a table:

```
     `ticker `price:(`appl;468.3)
ticker| appl
price | 468.3
     T,&.`ticker `price:(`appl;468.3)
ticker price
-------------
ibm    207.2
hp     16.8
appl   457.01
appl   468.3
```

When at least one column is a vector, a column which is a scalar will be extended to match the lengths of others

```
     T1<-([] c0;c2<-2013;c1)
     T1
c0   c2   c1
------------
ibm  2013 207.2
hp   2013  16.8
appl 2013 457.01
```

We also note that a character matrix can be used as a vector of rows in defining a column as long as the length of its first axis is the same as that of the other columns in the table:

```
     compnam<-3 8#'Int BusMH P CompApple   '
     T1<-([] ticker<-`ibm `hp `appl;price<-207.2 16.8 457.01;name<-compnam)
     T1
```

```
ticker price  name
----------------------
ibm    207.2  Int BusM
hp     16.8    H P Comp
appl   457.01 Apple
```

To retrieve column names of a table, Eli provides the function `cols`

```
     cols T1
`ticker `price `name
```

The ***shape*** of a table `T`, `#T`, is defined to be the *number of records* in `T`

```
     #T
3
```

We note that the *transpose* of a table results back into a *column dictionary*, i.e. `&.T` is `D`.

5.3 primary keys and keyed tables

In database systems, one typically designates a column, or a group of columns, in a table `T` to be the ***primary key*** of that table which usually uniquely identifies the records in that table `T`. That is, we have a smaller table `Tk` composed of the primary key column(s), and `Tv`, the table defined to be the rest of columns in `T`. Then a ***keyed table*** `T` is defined as a *mapping* between these two tables:

```
     Tk<-&.(<.`eqx):<.101 103 110
     Tv<-&.`ticker `price:(`ibm `hp `appl; 207.2 16.8 457.01)
     kT<-Tk:Tv
     kT
eqx| ticker price
---|--------------
101| ibm    207.2
103| hp     16.8
110| appl   457.01
```

For now we assume that the primary key is a single column. In the first line above, it is necessary to enclose the single items both in the domain and value of `Tk` to create a dictionary. To enter a keyed table directly as a table we do the following to achieve the same effect

```
     kT<-([eqx<-101 103 110] ticker<-`ibm `hp `appl; price<-207.2 16.8 457.01)
```

To access a record in the keyed table `kT` we provide *indexing* by a value from the table of keys, i.e. value from key column(s):

```
     kT[103]
ticker| hp
price | 16.8
```

To retrieve multiple records, it is a bit more elaborate as we can't simply do

```
     kT[101 110]
length error
```

We need to enclose each index as `kT[(<.101),<.110]` or more conveniently

```
     kT[(101;110)]
```

```
ticker price
-------------
ibm    207.2
appl   457.01
```

Keyed table can also be updated as tables but the indices are from items of its primary key table:

```
     kT[(101;110);`price]
-------------
ibm    207.2
appl   457.01
     kT[(101;110);`price]<-kT[(101;110);`price]*1.03
     kT
eqx| ticker price
---|----------------
101| ibm    213.416
103| hp     16.8
110| appl   470.7203
```

We also note that positional indexing for non-keyed table such as `kT[1]` no longer works here unless `1` happens to be a key column value. Since a keyed table is a dictionary like mapping, the functions `key` and `value` apply to it:

```
     key kT
eqx
---
101
103
110
     value kT   //kT before update
ticker price
-------------
ibm    207.2
hp     16.8
appl   457.01
```

If we already have an ordinary table which has a column of unique values, the function `xkey` with the table as the right argument and the intended key column name as the left argument converts it into a keyed table (*note: if* `[]` *is missing, it would result in a list, not a table*):

```
     T0<-([] eqx<-101 103 110; ticker<-`ibm `hp `appl; price<-207.2 16.8 457.01)
     T0
eqx ticker price
-----------------
101 ibm    207.2
103 hp     16.8
110 appl   457.01
     `eqx xkey T0
eqx| ticker price
---|--------------
101| ibm    207.2
103| hp     16.8
110| appl   457.01
```

And a keyed table can be converted to an ordinary table by supplying an empty left argument to `xkey`:

```
      () xkey kT
eqx ticker price
-----------------
101 ibm    207.2
```

```
103 hp    16.8
110 appl   457.01
```

All these operations will leave the original table unchanged.

In a situation when only combinations of several columns uniquely determine a record in a table `T`, `T` is said to have a *compound primary key*, i.e. a primary key consists of several columns. Since a keyed table is a mapping between two tables, this just makes the domain table to have more than one column:

```
     k2T<-([eqx<-101 101 110;city<-`ny `ln `ny] ticker<-`ibm `bp `appl; price<-207.2 16.8 457.01)
     k2T
eqx city| ticker price
--------|--------------
101 ny  | ibm    207.2
101 ln  | bp     16.8
110 ny  | appl   457.01
```

To retrieve record(s) from `k2T`, we do

```
     k2T[<.(101;`ny)]
ticker| ibm
price | 207.2
     k2T[((101;`ny);(101;`ln))]
ticker price
------------
ibm    207.2
bp     16.8
```

Notice that in forming list items for retrieving record(s) the order of values must be in the same order as the order of key columns in the definition of the compound keyed table. Given record(s) in the value portion of a keyed table, we can also do reverse lookup of their indices in the key column(s).

```
     kT!(<.(`ibm;207.2)),<.(`appl;457.01)
eqx
---
101
110
```

To get the column name(s) of a keyed table, we apply the `keys` function

```
     keys kT
`eqx
     keys k2T
`eqx `city
```

5.4 foreign keys and virtual columns

The construct of an *enumeration* `` `u: `` of a variable u of unique symbols over a variable v with values belong to u at the end of sect.**5.1** can be extended to that of a table `pT` with a primary key column `p` as *domain* and entries in a column of another table `T`, whose values all belong to that of `p`, as its *range*; this is denoted as `` `pT:. Therefore, we define a ***foreign key*** *column* in a table as an *enumerated value* over a *keyed table*.

```
     pT<-([cux<-1 3 5 8] sex<-'fmmf';age<-35 18 51 45)
     pT
cux| sex age
---|--------
```

```
1  | f   35
3  | m   18
5  | m   51
8  | f   45
     sales<-([] mech<-`p1 `p2 `p3 `p4 `p5 `p2;amt<-2.3 1.2 5 20.1 50.7 11;cust<-`pT:3 5 1 5 8 3)
     sales
mech amt  cust
--------------
p1   2.3  3
p2   1.2  5
p3    5    1
p4  20.1  5
p5   50.7 4
p2   11    5
```

A foreign key establish a ***relation*** between the enumeration domain table `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 `fk` with the dot notation `dt.fk.c0` from `dt`, and this is called a ***virtual column***.

```
     sales.cust.age
18 51 35 51 45 18
```

There is a function **`meta`** which gives the meta data of a table indicating the types of a column (sect.**4.2**) and which column is a foreign key.

```
     meta pT
c  | t f a
---|------
cux| I
sex| C
age| I
     meta sales
c   | t f  a
----|-------
mech| S
amt | E
cust| I pT
```

At times one would like to get actual values instead of the enumerated values, to do that simply apply the `value` function

```
     sales.cust
`pT:3 5 1 5 8 3
     value sales.cust
3 5 1 5 8 3
```

A built-in function `fkeys` applying to a table return a dictionary whose domain is the set of foreign keys of the table and whose value consists of the primary key table names:

```
     fkeys sales
cust| pT
```

As in ordinary vector `v`, the *index of* function `v!a` gives the position of `a` in `v`, for a table `t` and a record `b` , the *reverse lookup* function `t!b` gives the position(s) of record(s) `b` in table `t`.

```
     kTe<-([]eqx<-101 103 110 102 108; ticker<-`ibm `hp `appl `ge `gm; price<-207.2 16.8 457.1 23
11.2)
     kTe
```

```
eqx ticker price
----------------
101 ibm    207.2
103 hp     16.8
110 appl   457.1
102 ge     23
108 gm     11.2
     kTe!(110;`appl;457.1)
2
     kTe!((108;`gm;11.2);(110;`appl;457.1))
4 2
```

6. Queries: esql

6.1 create, load and insert statements

ELI provides a small set of *query* statements, ***esql***, which looks very similar to the standard SQL statements for traditional relational database management systems. Its functionality covers the basics of *q-sql* in the **kdb** database system (see *www.kx.com*) but it does not have the completeness of *q-sql* at present. Hence, while not every SQL statement has a ready counterpart in *esql*, 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 (one should check that `standard.esf` from ELI distribution is in `ws` subdirectory):

```
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 (I for integer, C for character, S for symbols and D for date-time) and *`width`* is a vector of integers indicating the width of character valued columns and may be empty if there is no column of character type while the number of columns and that of types must be equal. Note that the corresponding order is maintained between these three parameters.

```
     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*** statement is of the following form:

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

where `tbl` is the name of the table to be loaded, $\texttt{fld}_i$ is the *i*-th column name and $\texttt{val}_i$ is the *i*-th column value, *i=1..n*.

```
     w1
`r `r1 `r2 `s1 `s1
     w2
30 90 100 114 210
     do 'LOAD TABLE T1 (a<-w1;b<-w2)'
table T1 loaded.
     T1
a  b
------
r  30
r1 90
r2 100
s1 114
s1 210
```

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

```
INSERT INTO tbl (val1;…; valn)
```

where `tbl` is the name of the target table of insert, val_i is the value, *i=1..n*, to be inserted to the *i*-th column of `tbl`. Of course, each val_i must be compatible with the existing values in that column, i.e. of the same type and in case of character type, of the same width; val_i can be all scalars (with character columns being vectors), or vectors of the same length (with character columns being matrices having length of the first dimension equal to the length of other vectors). For example,

```
     do 'INSERT INTO T1 (`r1 `s2 `s3;50 64 88)'
new value inserted into table T1.
     T1
a  b
------
r  30
r1 90
r2 100
s1 114
s1 210
r1 50
s2 64
s3 88
```

We note that this statement inserted 3 records into table `T1`. Also, we can set/load a table as we did in the previous chapter without using **create** or **load** statement of *esql*.

6.2 select, exec and update* statements

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,.. ]]
```

i.e. it is one or several *constraint*s separated by a ‘,’; each *constraint* is of the form

```
lop cfn rop
```

where either *lop* or *rop* is a column name while the other then must be a literal constant, *cfn* is one of the comparison functions such as ‘=’ or ‘<’ in Eli or ‘IN’ (stands for the membership function ‘?’ in Eli). The presence of multiple constraints means a nested *where*, i.e. all constraints are *and* together.

fields is of the form:

columnexp **[**,*columnexp*$_1$**[**,*columnexp*$_2$,.. **]]**

i.e. it is one or several *columnexp*s separated by a ‘,’; each *columnexp* is of the form

[*newnam* <-**[***agrfn***]]** *colnam*

where *colnam* is the name of the source column in *tbl*, which can possibly be a virtual column if *tbl* has a foreign key, and *newnam* is the name giving to the resulting column in the output table; *agrfn* is the name of an aggregate function, in case the **BY** *group* clause is present. The possible aggregate function names are: **SUM, COUNT, MAX, MIN, FIRST** with the usual meanings of the indicated function applying to each group yielding a scalar value.

The meaning of **BY** *group* clause is similar to that of **GROUP BY** in SQL, it groups records by their value specified in *group* which is of the form:

[*newnam* <-**]** *colnam*

where *colnam* is the source column whose value is used to group records and *newnam* is the name of the resulting column. We give several examples of the select statement here:

```
     do 'SELECT FROM T1 WHERE b<100, a IN `r `r1 `s2'
a  b
-----
r  30
r1 90
r1 50
s2 64
SELECT successful.
     T<-([] n<-`x `y `x `z `z `y; p<-0 15 12 20 25 14)
     T
n p
----
x 0
y 15
x 12
z 20
z 25
y 14
     do ‘SELECT m<-MAX p,s<-SUM p BY name<-n FROM T’
name| m  s
----|------
x   | 12 12
y   | 15 29
z   | 25 45
SELECT successful.
```

The ***exec*** statement is of the same format as that of the *select* statement. 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; and in case of a one-dimensional table, it just returns the column values.

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

For example,

```
     do 'EXEC n FROM T'
`x `y `x `z `z `y
```

```
EXEC successful.
     do 'EXEC n,p FROM T'
n| x y x z z y
p| 0 15 12 20 25 14
EXEC successful.
```

Using the ***exec*** statement offers the advantage of directly utilizing a returned value into an Eli program for further processing. This is in sharp contrast to the situation in ordinary database systems where one need to fetch data from a RDMS selected by a group of SQL statements to be further processed by another programming language such as Perl or Python. More than conceptual simplicity in programming, this integration greatly improves processing efficiency.

The ***update*** statement* is of the same format as that of the *select* statement (*at present it is not well implemented) but with *fields* being replaced by *updfields* which is a group of one or several *updf* separated by ',' and each *updf* is of the form *colnam<-expr* where *colnam* is the name of a source column whose value is going to be replaced by *expr*:

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

6.3 query examples against a database

We shall illustrate the use of *esql* by running query examples against a sample database of a store. Assume we have already have `standand.esf` file in `ws`; then after issuing `)fload store2` (which is included in the `ws` directory of *eli* distribution) to bring in the data, we see the following lines to build a database of four tables directly (set `[]IO=0`):

```
     tc<-^.#na
     ts<-^.#mn
     te<-^.#en
     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 table `sales` or any other tables similarly. `sales` is the main table with `salx` as its primary key and `cust,stk,empl` are foreign keys pointing to tables `customer,stock,employee` respectively. `m_name` is the name of a merchandize in store and `in_stk` indicating it is in stock. We see several *esql* statements and their resulting tables here while the meaning of each is self-explanatory as it is the same as in a corresponding standard SQL statement.

```
     do 'SELECT dat_time FROM sales WHERE 25 > cust.age'
dat_time
-----------------------
2012.05.25T09:50:21.000
2012.05.25T10:37:03.000
2012.05.25T11:56:45.000
2012.05.25T12:00:22.000
2012.05.25T13:17:04.000
2012.05.25T13:23:41.000
```

```
2012.05.25T15:30:05.000
SELECT successful.
     do '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
SELECT successful.
     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
SELECT successful.
     do 'SELECT FROM sales WHERE 50 < amount,cust.sex=''f'''
salx cust stk amount payment dat_time             empl
-----------------------------------------------------------
4    1    2   68.2   cr     2012.05.25T11:30:24.000 1
7    3    1   92.5   cr     2012.05.26T16:10:27.000 0
8    3    2   62.4   cr     2012.05.26T17:23:48.000 0
11   3    0   57    cr     2012.05.25T12:23:43.000 2
20   3    3   74.2   cr     2012.05.25T15:03:44.000 2
23   5    4   82.5   cr     2012.05.26T19:43:47.000 3
24   5    5   61.4   cr     2012.05.26T20:57:08.000 3
27   5    3   61    cr     2012.05.25T15:57:03.000 5
31   6    4   77.5   cr     2012.05.26T21:30:27.000 3
SELECT successful.
     do '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
SELECT successful.
```

References

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

[2] W.-M. Ching, The Design and Implementation of an APL dialect, ELI, APL 2000Berlin Conf., Berlin, 2000.

[3] B. Kernighan and D. Ritchie, The C Programming Language, 2nd ed., Prentice Hall, 1988.

[4] N. Wirth, Algorithms + Data Structures = Programs, Prentice Hall, 1976.