ELI for kids, a novel way to learn math and coding

by Wai-Mee Ching

June, Aug, Sept. 2015, Jan., June 2016

Contents

By relieving the brain of all unnecessary work, a good notation sets it free to concentrate on more advanced problems, and in effect increases the mental power of the race.

-A.N. Whitehead, quoted in Ken Iverson's Turing Lecture

Introduction

One difficulty in teaching mathematics to youngsters is the lack of instant response to show a correct answer unless the student has a private tutor, and even an experienced tutor cannot provide answers to complicated formulae instantaneously. With the advent of modern computer and its associated ensemble of software programs, it becomes possible to teach mathematics to young students by interactive computer software. However, most math teaching software packages on the market concentrate more on fairly simple arithmetic operations with great effort in showing fancy graphic images such as adding two apples with three apples. Rarely such packages teach serious mathematical computations, not to say modern mathematical concepts. On the other hand, there are books which intend to teach youngsters how to program. A good example is the recent book "Python for Kids, a playful introduction to programming" by Jason Briggs. Of course, Briggs' book aims solely at teaching kids how to code not to learn math. The purpose of this book is to teach kids real math and good coding at the same time based on an interactive programming system called ELI we developed which is freely available at http://fastarray.appspot.com/.

ELI is based on an old programming language called APL which was based on a set of mathematical notations developed by Ken Iverson at Harvard in the late 1950s to early 1960s for the purpose of communicating computational algorithms in an exact fashion while he was teaching applied mathematics there. Iverson later joined IBM Research and continued his work on APL with Adin Falkoff. In 1966 with the help of Larry Breed and collaborators at IBM T.J. Watson Research Center, APL became an executable programming language. Since then it has been used in a variety of fields such as finance, insurance, logistics, electrical engineering, physics and economic data analysis. It was particularly popular on Wall Street during 1980s. But two factors prevented more wide use of APL, especially at schools: 1) APL font requires a special keyboard; 2) APL was quite expensive even though later there were free trial versions. ELI addresses these two issues by: 1) ELI uses one or two ASCII characters to represent a primitive function symbol in an understandable way whereas in APL it is represented by one APL character, thus preserving the one-character one-symbol principle of APL code; 2) ELI is freely available on multiple platforms.

Why ELI? Why not C, Java or Python? First, ELI as a modern version of APL is particularly suited for teaching mathematics to young students as this was the original goal in designing APL by Ken Iverson. More importantly, we would like to teach kids about serious programming with minimum amount of effort. We do not intend to introduce all fashionable stuff in computer programming today such as object oriented programming constructs. Kids can learn those later or they may discover that for many programming tasks OO paradigm is an unnecessary burden. ELI/APL is easy to learn, especially as a first programming language since it is devoid of unnecessary syntactic clutters of many other programming languages and it has a uniformity of design unburdened by numerous add-on functions or features of a language not well thought out in the beginning. This book aims to train a kid's mind, not to help one to find a job with a particular programming language requirement. Nevertheless, after a student finishes studying this book he will find that the basic coding concepts are the same; the other languages just have more syntactic dressing or are more verbose. In short, this book tries to introduce essential coding techniques to kids as quickly as possible.

ELI has four basic types of data: *numbers*, *characters*, *symbols* and *temporal data* (i.e. data relating to time and date); numbers include booleans, integers, real numbers and complex numbers. A single datum in ELI is called a *scalar*; an *array* is a homogeneous rectangular collection of data (i.e. all elements in an array must be of the same type). Each array has a shape, and one-dimensional arrays are called vectors (the shape of a vector is simply its

length). A *list* is a linear sequence of data items where each item can be a scalar, an array or another list; data items in a list can be of different types. And that is all as far as data in ELI is a concern.

ELI provides a large collection of built-in functions, called *primitive functions*, each represented by a symbol consisting of one or two ASCII characters and having one or two arguments. The collection of arithmetic and relational primitive functions is called *scalar functions* since each operates on arrays as a uniform extension of its operation on array elements which are scalars. The other primitive functions are called *mixed functions*; some mixed functions take a portion of an array and some transform a whole array. There are also five *operators* which apply to primitive functions to produce *derived functions*; of particular importance for lists is the *each* operator. A user defined function takes none, one or two arguments and can return a result or return no result. A line of ELI is a chain of operations execute from *right to left* with the output of one operation feeds as an input to the next operation in a *dataflow style* of programming. All primitive functions (and user defined functions) are of the same precedence, thus ensues the simplicity and uniformity of ELI syntax. In fact, ELI/APL code for mathematical computation can be looked upon as a linearization for a formula in mathematics.

We believe that ELI's austere notation with a minimum amount of relevant concepts makes it faster for kids to learn non-trivial computer programming. We assume a student who takes this study has only a bare minimum of math on his/her part, i.e. he/she understands addition and multiplication, whole numbers and fractions. We assume the student has no previous knowledge in coding whatsoever. Able to read English, download files, type on screen and edit an ordinary text file is all the computer experience required to get on to learn ELI. Once a kid is conversant in ELI programming, he/she would appreciate the beauty of its mathematical consistency and uniformity of its rules. These are ELI's inheritance from APL. But ELI also provides some convenience which is absent in classical APL, such as the easiness to input/output code/data files using ordinary text files and control structures common in other modern programming languages. ELI has dictionaries which are present in other languages such as Python and Perl; we will briefly touch on dictionaries here. In addition, ELI has tables and SQL like statements for database management which we'll refer to **A Primer for ELI** on ELI website's *document* section for students who have further interest to explore.

Mount Kisco, New York, 2015

***Note** on Aug. revision*: The change mainly reflects the difference in entering a defined function between ELI version 0.2 and version 0.3 (the current version) which is more like IBM APL2. There are also some minor improvements.
***Note** on Sept. revision*: Change of mirror site and minor corrections.
***Note** on June 2016 revision*: Add chapter 6, Data and Probability.

1. Arithmetic Computations and Algebra

1.1 Getting started

First, we go to the website http://fastarray.appspot.com to download the free ELI executable (for people reside in China please go to the mirror site http://www.sable.mcgill.ca/~hanfeng.c/eli/). Click on the middle box [Download] in the top bar. You would then see two sides in the download page: the left side is for ELI interpreter while the right side is for Ecc compiler. We'll pick the left side.

- Both interpreters and compilers are called *programming language processors*, i.e. they accept the code you wrote in a particular programming language and execute it on a specific computer platform. The difference between an interpreter and a compiler of a programming language is that when one uses an interpreter, he enters one line of code in that language, the interpreter processes that line, i.e. executes it on a particular computer platform and either spits out an error message or returns a result, if the execution is successful (it actually can take multiple lines or a whole program as we will explain in a later section). For a compiler, it takes a whole program, digests it, i.e. checks its correctness, transforms it into some suitable form, and finally turns it into machine code called an *executable* ready for user to execute. Clearly, an interpreter is more convenient for a user, especially if he is a newbie prone to make many mistakes since the compilation process is not instantaneous and usually takes longer time. Why then people bother with a compiler? That is because a compiled program in general runs much faster than an interpreted program and some programs require long time to run while some programs, so called production programs, will run millions of times. In fact, historically a compiler existed before there was an interpreter.

On the left side, we see choices of three platforms: Microsoft ***Windows***, Apple ***Mac OS*** and ***Linux*** (*Ubuntu*). Choose the one you want and download the executable (there are *readme* files to help you install ELI for each platform). On *Windows* as soon as you downloaded the executable, the system will create a subdirectory named `eli` in your `program files` directory and an ELI *icon* will appear in your desktop. Inside the `eli` directory there are two sub-directories: `bin` which contains `eli.exe` and documents, and `ws` which contains *workspaces* and *script files* (we will explain these terms later). In *Mac OS* these two directories are merged into one directory: `elim`, and in *Linux* these two are also merged into one `elix`.

We will assume that we are in the *Windows* platform. The operations in *Mac OS* or *Linux* are essentially the same as those in *Windows* while each platform provides auxiliary features to help users entering code either interactively or in batch mode. In *Windows*, we have the familiar cut and paste facility for the interactive mode and we recommend using *Notepad+* for editing text to enter a code file in batch mode. In *Linux*, one usually uses *Emacs* for text editing.

We click the ELI *icon* in *Windows* to start ELI or execute `eli` in a command line in *Mac OS* or *Linux*. We shall see in *Windows* that a *window* will pop up with:

```
ELI version 0.3 (C) Rapidsoft
        CLEAR WS
```

This means you are starting from a clean slate.

- A ***workspace*** in ELI is a unit to organize a user's work, including variables and functions defined so far, which can be saved after it is given a name, and a saved *workspace* can be recalled into play by loading it. In a *clear workspace* there is no variable or function exists except some predefined system variables.

In Mac OS or Linux, you would not see a window, but you would see the same lines, i.e. in Mac OS or Linux it is line based. To exit from ELI, you simply type

```
      )off
```

If you *type* a line (displayed with an indentation) into the Window, you will see the system *responses* with a line without indentation:

```
      3+2
5
```

You type `10*` in front of `3+2` and hit enter, and you see

```
      10*3+2
50
```

Uh! Should it be `32` since most other programming languages have a *precedence* rule that puts multiplication ahead of the addition? But ELI's ***precedence rule*** is rather *egalitarian*, i.e. all *functions*, including all ***primitive functions*** (the *built-in functions* in the ELI language) are of ***equal precedence*** and a line of code executes from ***right to left***, evaluating one function at a time. One reason for this rule is that ELI has a ***large*** *number* of primitive functions, as we will soon see, not just the usual addition and multiplication etc. Hence, the equal precedence rule of ELI makes life simple and ensures a uniformity of ELI syntax. Now, what if I do want to do `10*3` first? Simple, you just put a pair of parentheses around it. When ELI ***evaluates*** this line starting from the right, it sees the function already has a right operand and its left operand is in a pair of parentheses, so ELI evaluates the value of the expression enclosed inside the parentheses to get the left operand for `+`. Hence,

```
      (10*3)+2
32
```

In ELI, as in other computer languages, we can store a value into a ***name*** and recall it for later use:

```
      (a3<-10*3)+2
32
      a3
30
      a3%5
6
```

Such a *name* is then called a ***variable*** and `<-` is a *two character primitive function symbol* denoting the ***assign*** function, it *assigns* the value of the expression on the right side of `<-` to the variable on the left side of `<-`. Primitive functions in ELI are denoted either by one ASCII character (such as `+`, `*`, `%`) or two ASCII characters like `<-`. For a two character *function symbol* you must not put a blank between the characters since a blank is also a character. A *variable name* in ELI must start with an *alphabetic letter* and possibly followed by additional an *alphanumerical character (a letter* or a *digit*) and possibly interspaced by the character _ (i.e. _ cannot be the ending character of a variable name). So `6b` and `bcd_` are not legitimate names for a variable while `b6`, `b_x` and `b6_x` are. Needless to say that the names are *case sensitive*:

```
      A3%5
value error
      A3%5
        ^
```

The system responses with an *error message* because the name `A3` has not been assigned a value yet; hence it is not a variable. To see how many variables exist in the current workspace, we type

```
     )vars
a3
```

and see that so far `a3` is the only variable in our workspace.

1.2 Basic data types and how data are organized in ELI

There are four basic *types* of data in ELI: ***numbers***, ***characters***, ***symbols*** and ***temporal data***, i.e. time and dates. We have already seen numbers. ***Character*** data are strings (or arrays) of characters (or just one character) enclosed by a pair of single quote characters (`'`) when we enter them; but when we display their values that pair of quotes are not present. For example,

```
      c2<-'Hello World!'
      c2
Hello World!
```

A ***symbol*** is a string of characters qualified to be a variable name (see the preceding section) prefixed by a back-tick character (`` ` ``). For example,

```
      c3<-`Smith
      c3
`Smith
```

To see a temporal datum we type

```
      []TS
2015.01.15T17:31:05.348
```

where `[]TS` is the *system variable* indicating the *current time*. It is of a subtype of temporal data called *datetime*; the part before the character `T` indicates the date while the part after `T` indicates the time: hour, minute, second up to millisecond. We will not go into details on the other subtypes of temporal data; please see § 2.4 of [1].

A single data item is called a ***scalar***, the variables `a3`, `c3` and `[]TS` are all scalars and so are the values they each holds. `c2` is not a scalar because it consists of more than one character (while `'3'` is a scalar). `c2` is an ***array***, a one-dimensional array which we usually call a ***vector*** and we call a two-dimensional array a ***matrix***.

- ELI organizes multiple elements of the *same type* into *rectangular cubes* called ***arrays***; a one dimensional array is called a ***vector*** and a two dimensional array is called a ***matrix***.

We have already seen how to enter a character vector as a string of characters in ELI. To enter a numeric vector, we simply enter each number separated by one or more spaces:

```
      v1<-10 0.5 78 0 1.2
      v1
10 0.5 78 0 1.2
```

We enter a vector of *symbols* and a vector of *dates* (a temporal data type) in a similar fashion:

```
      v3<-`Smith `Jobs `Bates
      v4<-2015.01.03 2015.01.07 2015.01.10 2015.01.15
      v3
`Smith `Jobs `Bates
      v4
2015.01.03 2015.01.07 2015.01.10 2015.01.15
```

Note that we can freely mix integers and fractional numbers in `v1`, but in entering a fraction number less than `1` one must put a `0` in front the '.'. This is because in ELI '.' frequently is part of a two-character symbol representing a primitive function which differs from the one with no '.':

```
      2*0.5
1
      2*.5
32
```

The second expression is `2` to the power `5`.

Each array has a ***shape*** which indicates the *lengths* in each dimension of the rectangular cube constituting the array. For a vector, its *shape* is just its *length*.

- In ELI a primitive function is either ***monadic*** if it has only one argument (i.e. the right argument) or ***dyadic*** if it has two arguments, one on the right and one on the left.

To query the *shape* of an array, we apply the monadic ***shape*** *function* `#` to an array:

```
      #c2
12
      #v1
5
      #v3
3
      #v4
4
```

All these are just lengths of vectors. For a 3 by 4 matrix, its shape should be `3 4`. But how do we enter such a matrix? To do this we deploy the dyadic ***reshape*** *function* `#`:

```
      m2<-3 4#c2
Hell
o Wo
rld!
      #m2
3 4
```

Therefore, whether `#` is interpreted as the *reshape* function or the *shape* function depends on whether there is an argument to the left of `#`. We see that the shape of the result of a reshape is equal to the left argument of the reshape:

`#shp#v` ←→ `shp`

for all data `v` and a non-negative integer vector `shp`. The shape of an array is called a *shape vector*; since each element of that vector denotes the length of one of a cube's dimension, it must be a non-negative integer, i.e. an integer $1_i \geq 0$. For an array *a*, the *shape* of the *shape vector* `#`*a*, i.e. `##`*a*, is called the ***rank*** of *a*; we see that the rank of an array *a* is the *dimension* of *a*. For a vector *a*, its rank is 1, for a matrix the rank is 2 and we'll see later that for a scalar *a*, its rank is `0`.

With reshape, we can enter not only matrices but also high dimensional arrays. For example,

```
      a3<-2 3 4#v3
```

```
      a3
`Smith `Jobs  `Bates `Smith
`Jobs  `Bates `Smith `Jobs
`Bates `Smith `Jobs  `Bates

`Smith `Jobs  `Bates `Smith
`Jobs  `Bates `Smith `Jobs
`Bates `Smith `Jobs  `Bates
```

We observe a property of reshape: that if the right argument `v` does not have enough elements to fill in the required number of elements of the resulting array it then keeps reusing elements of `v`. On the other hand, if the right operand `v` has more elements than what is required for a reshape operation then the tail part of `v` would be left out:

```
      m1<-2 2#v1
      m1
10 0.5
78 0
```

For *non-homogeneous* data, i.e. items in the data are not of the same type, or data of *non-rectangular* shape, i.e. items are of different length even though they are of the same type, ELI organizes them into *lists*. A ***list*** is entered with a pair of parentheses and items in a list are separated by '`;`' and when displayed, each item is prefixed with '`<`':

```
      l1<-(c2;v3;v1)
      l1
<Hello World!
<`Smith `Jobs `Bates
<10 0.5 78 0 1
      l2<-(1;v1;5 9 8)
      l2
<1
<10 0.5 78 0 1
<5 9 8
```

The *shape* of a list is its *length*, i.e. the number of items in it:

```
      #l1
3
      #l2
3
```

The *reshape* function *does* apply to lists in a limited fashion, i.e. a list can't be reshaped into a scalar or an array but can be reshaped into another list. We remark that an item in a list can be a scalar, an array or another list. Hence, a list is the most general data structure in ELI. So, ***scalar***, ***array*** and ***list*** are all there are for organizing data in ELI.

Finally, what is the *shape* of a *scalar*, say `c3` or `8`, we haven't tried that yet?

```
      #c3

      #8
```

Uh, should we get `0` or `1`? To make sure we assign it to a variable and query that further:

```
      ss<-#c3
      #ss
0
```

This tells us that the shape vector of a scalar is a vector of length `0`, that is what we call an ***empty vector***, i.e. a vector of length zero with no element. To get `1` as a resulting shape, we *reshape* `c3` into a vector of one element first:

```
      cv<-1#c3
      #cv
1
      cv
`Smith
```

`c3` and `cv` hold the same value but are of different structure, one is a scalar while the other is a one element vector. This is rather mysterious and annoying. But we'll leave it there and move on as we have just gotten started. Later we shall see that the reason behind such a mystery is to maintain *mathematical consistency*. For many cases, there is no practical difference between a scalar and a one element vector holding the same value.

In fact, there is another way to turn a scalar into a one element vector. This is achieved by the monadic function ***ravel*** denoted by the character ',':

```
      #,c3
1
      ,c3
`Smith
```

If the argument *a* to the *ravel* function is an array then *,a* turns *a* into a vector consisting of all elements of *a* in *row-major* order. The *ravel* function has no effect on a vector or a list. For example,

```
      m1
10 0.5
78 0
      ,m1
10 0.5 78 0
      m3
 1  2  3  4
 5  6  7  8
 9 10 11 12

13 14 15 16
17 18 19 20
21 22 23 24
      ,m3
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24
```

We can also see that the *shape* (i.e. *length*) `#,`*a* of *,a* is the *product* of elements in the shape of *a*: the shape of `,m1` is `2*2` and the shape of `,m3` is `2*3*4`. To this end, we introduce one more monadic function related to the shape function `#`: the ***count*** *function* `^` which is defined as

$$\text{\^{}}a \quad \leftrightarrow \quad \#,a$$

Consequently, we have

```
      ^c3
1
      ^v2
4
      ^m3
24
```

So $\hat{}a$ just counts the number of elements in a whether it is a scalar, a vector or a higher dimensional array.

1.3 Rudimentary set theory

A ***set*** $S=\{a_1,a_2, \ldots\}$ is a collection of distinct elements a_1, a_2, … which can either be *enumerated* or *defined* by other precise means; and for each element a_i in S, we say a_i *belongs* to S, denoted by $a_i \in S$. A set S is ***well-defined*** if given an item a, either a is a member of S. i.e. $a \in S$ or $a \setminus\in S$, i.e. a does not belong to S. We only consider well-defined sets. Two sets are the same if they contain the same elements, in other words the order you list them doesn't matter. A set can contain other sets as its members just like a list can contain other lists as its elements.

A set S is ***finite*** if it contains no element (i.e. an ***empty set***) or only a finite number of elements; otherwise it is ***infinite***. For a finite set S, we can, in principle, list all its elements. For an infinite set we must define it by other means. Usually, we define an infinite set S by

$$S = \{x \mid P(x)\}$$

where P is some ***proposition*** for which we can decide whether $P(x)$ is *true* (then $x \in S$) or *not true* (then $x \setminus\in S$). We have the following examples of sets,

S_e= { } is the empty set
S_1={1} has one element
S_0={{ }}also has one element, i.e. the empty set
S_2={`Smith, `Jobs, `Bates} has 3 symbols as its elements
S_3={`Jobs, `Smith, `Bates} is the same set as S_2
S_4={0,1}
S_5={`Jobs, 2011, {'Apple Company',' Pixar'}}
S_6={x | 0<x<1}
S_7={x | x is a prime number}
S_8={x | x is a prime number and x+2 is also a prime number}

We know that the set S_6 is *infinite* and *not enumerable* (which we will not prove here). We already know that the set S_7 is *infinite* (proved by Euclid, the same Greek guy who gave us the famous geometry text book more than two millennia ago) and we even know how fast that set grows by the ***prime number theorem*** first proved near the end of the 19th century by Hadamard and de la Vallee-Poussin. We don't know yet whether S_8 is infinite, i.e. are there infinite pairs of *twin prime numbers* $\{p, p+2\}$? This is the ***twin primes conjecture***. But we are tantalizingly close to confirm this conjecture due to an amazing result of Yitang Zhang on *bounded gaps* in 2013.

A set A is a ***subset*** of set B, denoted by $A \subset B$ ***if*** the statement: an element a belongs to A ***implies*** a belongs to B, i.e.

$$a \in A \rightarrow a \in B$$

Now we can define precisely that two sets A and B are the ***same*** *set* if $A \subset B$ and $B \subset A$, and we denote it by $A=B$. We define the ***union*** of two sets A and B, denoted by $A \cup B$, as the set whose elements either belong to A or belong to B:

$$A \cup B = \{x \mid x \in A \text{ or } x \in B\}$$

We define the ***intersection*** of two sets A and B, denoted by $A \cap B$, as the set whose elements belong to both A and B:

$$A \cap B = \{ x \mid x \in A \text{ and } x \in B \}$$

Two sets A and B are *disjoint* if they have no elements in common, i.e. their *intersection* is the *empty set*:

$$A \cap B = \{ \}$$

A ***map*** f from a set A to a set B

$$f: A \rightarrow B$$

is an *assignment* from A to B so that for each $a \in A$, there is a unique $f(a) \in B$. This is what we usually called a ***function*** from A to B (There are multi-valued maps fm from A to B where for some $a \in A$, $fm(a)$ consists of more than one element in B; we would not consider such maps here). A is called the ***domain*** of the function f, and the set

$$\{ f(x) \mid x \in A \}$$

is called the ***range*** of f which by definition is a subset of B. If $A=B$, the function Id defined as

$$Id(a) = a \text{ for all } a \in A$$

is called the *identity function* of A, i.e. it just map every element in A to itself. If for a function f, there is an element $c \in B$ such that

$$f(a) = c \text{ for all } a \in A$$

f is then called a *constant function*. We introduced the basic terminology of set theory in modern mathematics here to make our later discussion about computations involving arithmetic operations more convenient.

In addition, we define the ***Cartesian product*** of two sets A and B, denoted by $A \bullet B$, as the set of all pairs (x,y) with first component x belongs both A and the second component belongs to B:

$$A \bullet B = \{ (x,y) \mid x \in A, \ y \in B \}$$

For example, if $A= \{1, 2, 3\}$ and $B= \{`apple, `orange\}$ then

$$A \bullet B = \{(1,`apple), (1,`orange), (2,`apple), (2,`orange), (3,`apple), (3,`orange)\}$$

1.4 Numbers and arithmetic functions

Let us start with the set of *natural numbers* N, i.e. the whole numbers we used to *count* items, be they apples or oranges,

$$N = \{1, 2, 3, \ldots.\}$$

When people realize that counting apples is essentially the same as counting oranges, they developed the abstract idea of *whole numbers* which we now call ***integers***. So, in the long history of human progress, *counting* is the beginning of mathematics. It took people a long time to invent the notion of *zero*, i.e. no items, no apple or no orange, and to denote it by the number 0. Let us add this new number to the set N and denote the new set by N_0, i.e. $N_0=N\cup\{0\}$, or we can define N_0 as the set of all non-negative integers, i.e. whole numbers greater than or equal to 0:

$$N_0=\{0, 1, 2, \ldots.\}$$

The subset $B=\{0, 1\}$ of N_0 is called the set of Boolean numbers, or just *booleans*. In a *clear workspace*, you can type:

```
      []IO
1
```

`[]IO` is a ***system variable*** called the ***index origin***, like `[]TS`, which pre-exists in a workspace set by the ELI system. But unlike `[]TS`, you can reset the value of `[]IO` if you like:

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

And these are the only two values the system variable `[]IO` can take. What is the purpose of having two different values for `[]IO`? The answer lies in the monadic primitive function ***interval generator*** `!`, for a number `n` in N_0, `!n` generates `n` consecutive whole numbers starting from the value of `[]IO`:

```
      !10
0 1 2 3 4 5 6 7 8 9
```

And if we reset `[]IO` back to its original value in a clear workspace, we get

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

So `[]IO` tells where do we start counting, `1` or `0`. We'll see later that the values give out by many primitive functions in ELI depend `[]IO` on in a way similar to the interval generator `!`.

Let us store that vector of ten integers into a variable `v10` and do some operations to it:

```
    100+5*v10<-!10
105 110 115 120 125 130 135 140 145 150
```

We multiply `v10` by `5` and then add `100` to it. Notice that `5` is multiplied to each element of `v10` and `100` is added to each element of that result. For addition, multiplication and many other similar *dyadic primitive* functions *f* in ELI (remember such a function takes a left and a right argument, and is built into the ELI system) the value of a ***singleton*** *operand* (i.e. it is either a scalar or a one-element vector) on one side of *f* would be repeatedly used against the elements in the operand on the other side of *f*. And the other operand involved can be an array in general, such as a matrix:

```
   3 4#v10
1  2 3 4
5  6 7 8
```

```
9 10 1 2
      (3 4#v10)-1
0 1 2 3
4 5 6 7
8 9 0 1
```

This *rule* is called is called ***scalar extension*** and these functions (to be explained more in detail later) where scalar extension hold are called ***scalar functions***. All *arithmetic functions* in ELI are scalar functions.

We take a quick detour to introduce the dyadic primitive function ***deal*** `?.` (this is a *two-character symbol*, a `?` followed immediately by a dot) which is not a scalar function. Both arguments `m` and `n` to the *deal function* `m?.n` must be positive integers and `m≤n`. For such two arguments `m?.n` *randomly* picks `m` *distinct* integers from `!n`. Consequently, the dyadic deal function is `[]IO` dependent. For example,

```
      w10<-10?.100
      w10
14 76 46 54 22 5 68 94 39 52
```

Hence, the deal function provides a convenient way for us to generate an irregular sequence of integers to play with. We call a computation ***F(a,b,…)*** ***functional*** if the result of ***F(a,b,…)*** purely depends on the values of it arguments no matter how one carries out the computation. All the primitive functions in ELI are *functional* except the dyadic *deal* function and its monadic companion, the ***roll*** function `?.`. To see this, we type the deal function two more times

```
       10?.100
84 4 6 53 68 1 39 7 42 69
       10?.100
59 94 85 53 10 66 42 71 92 77
```

This shows that giving the same arguments `10` and `100`, each time the deal function `m?.n` gives out a different answer just like when you throw a dice you most likely would see different results. However, in a clear workspace, the first execution of `10?.100` would always give the same result as that of `w10`. This mystery is due to the fact that in the execution of `m?.n` there is another *participant*, a system variable called `[]RL`, the ***random seed***. In a clear workspace `[]RL` has the value `16807`, and after each execution of `?.` , either dyadic or monadic, this value of `[]RL` would change. For a positive integer `n`, the monadic *roll* function `?.n` randomly picks an integer from `!n`, and for a vector `v` each component of `(?.v)[i]` is defined to be `?.v[i]`. For example,

```
      ?.30 90 100
4 69 46
       ?.30
16
       ?.90
20
       ?.100
5
```

Note that `4 69 46` is not the same as `16 20 5` because each execution of `?.` changes the random seed `[]RL` and hence the result.

Now, let us resume our discussion of arithmetic operations. We can add `w10` to `v10`:

```
       w10+v10
15 78 49 58 27 11 75 102 48 62
```

The result is as expected, i.e. each element in `w10` adds to the corresponding (in terms of its position) element in `v10` to yield a whole result. However, if we type

```
        w10+7 11 10
length error
        w10+7 11 10
         ^
```

We get a *length error*, i.e. ELI informs us that the operation runs into an error because the left and right operands are of different lengths as one is `10` and the other is `3`. Remember that the shape of a vector is its length. In general, for two operands `A` and `B` of a dyadic primitive function `f`, and let `R` be the result (which needs not be assigned to a variable) of

```
         A f B
```

then `R` is *well-defined* and `f` is called a ***scalar function*** if either

1) `A` and `B` are two scalars or two arrays of the same shape (i.e. `(#A)=#B`) then `R` is a scalar or an array of the same shape as that of `A` or `B` with the value of each element R_{ij} in `R` is $A_{ij}fB_{ij}$ calculated from the corresponding elements in `A` and `B`, or
2) one of the operands, `A` or `B` is a singleton, then `R` is of the same shape as the other array operand and each element R_{ij} of `R` is $A_{ij}fB_{ij}$ with either A_{ij} or B_{ij} being of a repeat value of the scalar.

The requirement on `A` and `B` that either they are of the same shape or one of them is a singleton is called that `A` and `B` are ***conformable***. A *monadic* primitive function `f B` can be similarly defined to be a ***scalar*** function if each element R_{ij} of the result `R` is calculated as fB_{ij}. Let us see some more cases of dyadic scalar function's operation:

```
       5*3 4#v10
 5 10 15 20
25 30 35 40
45 50  5 10
      w2<-3 4#w10
      w2
14 76 46 54
22  5 68 94
39 52 14 76
      w2+5*3 4#v10
19  86  61  74
47  35 103 134
84 102  19  86
      w2+3 5
rank error
      w2+3 5
        ^
      w2+3 5#!15
length error
      w2+3 5#!15
        ^
```

The first case is ok because `5` is a scalar and the second case is ok because `w2` and `5*3 4#v10` are of the same shape. The third case runs into problem because `w2` is a matrix, i.e. an array of *rank* (dimension) `2` while `3 5` is a vector, i.e. of *rank* `1`. The last case runs into problem because `w2` and the other side of `+` are of different shape, one is a `3` by `4` matrix while the other is a `3` by `5` matrix.

Let us see some monadic primitive scalar functions. First, we show that the dyadic function ***subtract*** `-`, i.e. *minus*, and the monadic function ***negate*** `-` are both *scalar* functions:

```
       0-v10
_1 _2 _3 _4 _5 _6 _7 _8 _9 _10
      -v10
_1 _2 _3 _4 _5 _6 _7 _8 _9 _10
      -3 -4
1
      -3 4
_3 _4
      -3 +4
_7
```

For the first expression, we see that `0-` is applied to each element in `v10` to yield ten negative numbers. We note that a *negative number* is *prefixed* with an under bar character '_', just like a fractional number such as tenth is prefixed by a '.' in `0.1`; there should be no space between '_' and the digit following it (and for a negative number what follows '_' must be a digit; remember that a fractional number less than `1` must start with a `0` because `_.1` will result in a `1` as we'll see later while `_0.1` is the correct way to input the number negative tenth). The second expression just exemplifies the fact that the monadic function **`-v`** is defined as **`0-v`** for all numeric expression **`v`**, i.e. the *negate* function is simply a shorthand of the *subtract* function with the left argument fixed to `0`. The third expression above looks like two negative numbers in most other programming languages. But for the ELI parser, it looks from ***right to left*** and seeing (`3 -4`) as the first item to evaluate which results in `_1`; it then applies the monadic `-` to that with `1` as the result. In the last expression above the parser sees (`3 +4`) first with `7` as the result and then applies `-` to it getting `_7`. All these look different from that of other programming languages, but are all very *logical* and *succinct*.

We also notice that for a character *c* representing a primitive dyadic or monadic function *f* with left and/or right argument *a* and/or *b*, it is not required to have a *blank* between them to express *a f b*, may *a* and/or *b* be numbers or variables. On the other hand, there is also no harm to insert one or several blanks between them, sometimes for the sake of readability. This rule also applies to functions *f* which are represented by two-character symbols such as the *deal* function `?.`. The rule removes a programmer's burden to check whether there is a blank between *f* and its arguments in his/her code.

The ***divide*** function is represented by the *percent* character '`%`'. For example,

```
      1%v10
1 0.5 0.3333333333 0.25 0.2 0.1666666667 0.1428571429 0.125 0.1111111111 0.1
      %v10
1 0.5 0.3333333333 0.25 0.2 0.1666666667 0.1428571429 0.125 0.1111111111 0.1
      w10%v10
14 38 15.33333333 13.5 4.4 0.8333333333 9.714285714 11.75 4.333333333 5.2
      w10%1
14 76 46 54 22 5 68 94 39 52
      w10*1
14 76 46 54 22 5 68 94 39 52
```

We see first that *fractional numbers*, which are part of so the called *floating- point numbers* in computer terminology, are printed out up to 10 decimal digits even if a number resulting from a division has mathematically many more digits or infinite number of digits such as one third (`1%3`).

The second expression above just indicates the fact that the monadic function ***reciprocal*** `%` is simply a shorthand for the dyadic function `%` with fixed left argument `1` similar to the relationship between the monadic `-` and dyadic `-`. The third expression above just indicates that `%` is a dyadic scalar function and two of its arguments need to be of the same length for two vectors. For a dyadic arithmetic function *f* and a number *b* in ELI, we denote the map

$$a \rightarrow a\ f\ b$$

from domain D to range R by f_b. We call b the ***identity element*** of f if $f_b(a)= a$ for all a in D, i.e. f_b is the *identity map* on D. The last two expressions above just show the fact that `1` is the *identity element* for both dyadic `%` and `*`. And from what we studied earlier, we see that `0` is the identity element for dyadic `+` and `-`(but a is on the right side).

We now introduce the ***equal function*** $a=b$. It is a dyadic scalar function. We can try out for a few examples:

```
      1=1.0
1
      0.25=1%4
1
      0.3333333333=1%3
0
      0.3333333333*3
0.9999999999
      (1%3)*3
1
```

First, we notice that the answer to the query $a=b$ for two singleton operands is either `1` or `0`. In ELI `1` represents the Boolean value ***true*** and `0` represents the Boolean value ***false***. This is why we called the set B={`0`,`1`} the set of Boolean numbers. We shall see in later sections that this representation has a tremendous advantage over using the words *true* and *false* for Boolean values.

Back to inspect examples above, the first expression yields a `1`; it just says that when we write a whole number either as an integer or in a floating point number format, it is the same number. The second expression also yields a `1` because `0.25` is exactly one fourth. In the third expression, it looks like we copied the third item from the print out result of `1%v10` which corresponds to `1%3` but we end up in a `0`, i.e. a *false* indicating that the two sides are not equal. Why? When we multiply the left side of `=` by `3` we get `0.9999999999` which we know it is not exactly `1`, but when we multiply the right side of that `=` by `3` we get `1`. Recall that we remarked earlier that ELI only displays a fractional number up to 10 decimal digits. So the third element displayed in the result of `1%v10` is not its true value; the true value of `1%3` is stored in the computer according to a standard floating point number representation scheme which we are not going into details here. But that true value stored in the computer by ELI certainly contains more decimal digits than ten `3`s. What is stored is still an approximation to the fractional number `1%3`.

Let us define the set Z of all integers to be the union of N_0 (the set of natural numbers plus 0) and its negative counterpart:

$$Z = N_0 \cup \{n \mid -n \in N\}$$

And we defined the set R_0 of all *fractional numbers*, more formally called the set of ***rational numbers*** as follows:

$$R_0 = \{n\%m \mid n,m \in Z,\ m \neq 0,\ n \text{ and } m \text{ have no common factor other than } 1\}$$

Note that the last requirement on n and m in defining R_0 can also be satisfied by canceling out the common factor in m and n if there is one. With this requirement `2%6` would not be an element of R_0 since it is the same number as `1%3`.

A minor puzzle: why ELI doesn't use n/m to denote a fraction? This is because the character '`/`' is used for other primitive function and operators in ELI as we shall see soon. Finally, we define the set R of ***real numbers*** to be:

$$R = \{x \mid x \text{ can be approximated arbitrary close by a } y \in R_0\}$$

This is a good definition except that the phrase '*approximated arbitrary close*' need to be more precisely defined and which we will do in section 1.6. For now we have the following:

$$B \subset N_0 \subset Z \subset R_0 \subset R$$

All sets above are infinite except B, and all sets above are ***countable*** except R. We'll explain the concept of *countable* later. To say R is ***uncountable*** just means that R is a much larger set than R_0 even though both are infinite.

Going back to the definition of the set of rational numbers R_0, one wonders what happens if m=0? A *domain error*? Let us try:

```
      1%0
0w
      _1%0
_0w
      0%0
1
      0w+100
0w
      0w*100
0w
```

Here `0w` represents ***infinity*** (its mathematical notation is ∞). When a positive number n is divided by a smaller and smaller positive number m, the result is getting larger and larger and after a long while people devise the concept of *infinity* (∞) to denote the ultimate limit of this process instead of just saying division by 0 (when m reaches 0) results in an error. And there are infinities in two directions, one positive (`0w`) and one negative (`_0w`). However, `0%0` is `1`. This is consistent with the rule that some number a divided by itself always results in `1`. The last two expressions above are also easy to understand since an infinite is so huge that adding any finite number to it or multiplying a finite number to it results in an infinity. Now we can extend our definition of R_0 above to remove the restriction $m \neq 0$ as follows:

$$R_0^* = \{n\%m \mid n,m \in Z\} = R_0 \cup \{\infty,-\infty\}$$
$$R^* = R \cup \{\infty,-\infty\}$$

1.5 Short functions and how to save your work

Let us have a simple quiz: a man paid $1.1 for a pencil and a piece of paper; the pencil is $1.0 more expensive than the paper. What is the price for the pencil and what is the price for the piece of paper? A quick minded person may answer right the way: one dollar and ten cents for each respectively. But he soon realizes that the pencil then becomes only $0.90 more expensive than the paper. To solve the problem in a more thoughtful way, we lay out the facts in two equations:

$$x + y = 1.1 \qquad (1)$$
$$x - y = 1 \qquad (2)$$

where x stands for the price of the pencil and y stands for the price of the paper. When we add equation (2) to equation (1), and separately subtract (2) from (1) we get the following answer for x and y:

$$2x = 1.1 + 1 \quad \textit{divide both side by } 2 \rightarrow \quad x = (1.1 + 1)\,\%2 = 1.05$$
$$2y = 1.1 - 1 \quad \textit{divide both side by } 2 \rightarrow \quad y = (1.1 - 1)\,\%2 = 0.05$$

So the answer is the pencil costs one dollar plus 5 cents and the paper costs 5 cents. This is just an instance of what we call the ***sum and difference problem***: two items of possibly different sizes, we know the *sum* of the sizes of the

two items as well as the *difference* of the sizes, what is the size of the larger item and that of the smaller item? If we replace 1.1 above by the name *sum* and replace 1 above by the name *diff*, we then have the following:

$$size\_a = (sum + diff)\ \%2$$
$$size\_b = (sum - diff)\ \%2$$

This solution can surely be applied to many similar situations disguised with items and sizes of different names such as knowing the total length of two pieces of wood and how much one piece is longer than the other. It would be handy if we can write an ELI function to remember the formulae involved.

Primitive functions in ELI are the functions provided by the ELI systems and they are represented symbolically by one or two ASCII characters, and many times the same character symbol can represent either a *monadic* or a *dyadic* primitive function depending on whether there is a left argument to that function symbol. So far we have seen the following: `+`, `-`, `*`, `%`, `=`, `#`, `!`, `?`. (only the last is a two character symbol). In ELI one can write one's own function with a piece of code for later use; such a function is called a ***defined function***. A *defined function* is represented by its ***function name***. The rule to form a *legitimate function name* is the same as that for a variable name. Like primitive functions, a defined function can be *monadic*, i.e. it takes a *right argument* only, or *dyadic*, in other words it takes a *left argument* and a *right argument*. But it is ok for a defined function to take *no argument*; a defined function with no argument is called a ***niladic*** function. We note that every primitive function yields a ***result*** (even an empty vector is a result); but it is also ok for a defined function to return *no explicit result* (even though it may print out intermediate values and possibly *changes* the values of other variables).

- a (right) ***parameter*** *p* to a defined function *df* is a place holder in the *body* (i.e. a piece of code implementing *df*) of *df* so when the function is called with an actual argument *a*:

  df a (or *b df a* in case *df* is dyadic)

  the value *a* is assigned to *p* (in case of dyadic *df*, *b* is assigned to *q*, the left parameter at the same time),
- then the body of *df* is executed.

We will explain how to write a defined function in general in Chapter **5**. For the moment, we would like to show how to write a kind of simple defined functions in a ***short function*** form. You start a short function with a '{' followed by the function name and a ':', then one or several expressions separated by ';' and end it with a '}' as in the following (the notation *a* = *b* | *c* below means that *a* is either *b* or *c*):

```
{fnam: expressions}
 expressions = expression | expression;..;expression
```

There are the following assumptions made about the `expressions`:

- It must contain a variable `x` which is the right parameter of `fnam`, i.e. the variable stands for the right argument to the function `fnam` (when the function is called `x` gets the value of the right argument).
- If it contains a variable `y` then `y` is the left parameter of `fnam`. Hence `fnam` is then a dyadic function.
- If it contains a variable `z` then `z` is the result of `fnam`. Otherwise, the result of the last `expression` in `expressions` is the result of the short function `fnam`.

It is easier to see how these rules work by a concrete example. So let us write a function, the one we named for our sum and difference problem. The function has two inputs: the sum and the difference. We designate the *sum* to be

the left *parameter* `y` and the *difference* to be the right *parameter* `x`; and we return a result as a two-element list with the first element the size of the larger item and the second element the size of the smaller item:

```
      {sum_diff: ((y+x)%2;(y-x)%2)}
sum_diff
       1.1 sum_diff 1
<1.05
<0.05
```

We see that after we type in a short function the system responds by printing out the name of the function, signals that there is no error and the function is established in the workspace. One can then apply the function by feeding it a pair of arguments (because it is a dyadic function). And after we tried, the system gives out a result in response; in this case it is a list of two numbers, one for the larger item and the other for the smaller item. Just like primitive functions, we could assign the result of a function application to a variable, say `items`, then the value would not be displayed but a variable would be created holding that value.

There are several *commands* you can type into the ELI interpreter to *query* about your workspace or ask the system to do something, not to evaluate an expression. They are called ***system command***s. The first system command we encountered is `)off` which asks the ELI interactive system to *close* the current session. The second system command we have seen is `)vars` which queries about user defined variables exist in the current workspace. Similarly, the system command `)fns` queries about existing user defined functions in the current workspace. We try

```
      )fns
sum_diff
      )vars
```

That means we now have one defined function named `sum_diff` in the current workspace. We see that the response to `)vars` is empty. What happened? Even though we didn't assign the result to a variable, but what about the variables `y` and `x` we used inside the function body? The reason is that in ELI, the parameters to a defined function as well as the result of a defined function, if it exists, are ***local***; that means they disappear after the execution of a function finishes. In general, a defined function can have variables in its body which are not local as we shall see later. But for a short function all variables in its body *`expressions`* are *all local*, not just the parameters and the result `z`.

All this will go away once we *log off* from the system by typing `)off`. To save what we have done for later use we need the *save* it. The system command for save is `)save`. However, when we just do that

```
      )save
not saved, this is a clear ws
domain error
```

What happened is that we started in a clear workspace and that remains to be the name of the workspace despite the fact that we did some work including adding a defined function. But a *clear workspace* cannot be saved. To save a workspace, you need to give it a name. The system command `)wsid` can be used both to query the name of the current workspace and to change the name of the current workspace:

```
      )wsid
CLEAR WS
      )wsid ABC
was CLEAR WS
      )save
```

```
2015.02.05 01:06:45 (gmt-5) ABC
      )wsid
ABC
```

Once a workspace is saved we can call it back later by the system command `)load` after you logged off. You can also start all over with a clean slate by issuing the system command `)clear`:

```
      )clear
      )wsid
CLEAR WS
      )fns
      )load ABC
saved 2015.02.05 01:06:45 (gmt-5)
      )fns
sum_diff
```

We see that after `)clear` there is no function there since it becomes a clear workspace. But after loading our saved workspace `ABC` the function `sum_diff` is there. This would be the same for variables including `[]IO` if it is modified.

1.6 More numerical functions

There is a dyadic function ***residue*** `a|b` which relates to *division*: while `a%b` gives the result of `a` divided by `b`, `a|b` gives the *remainder* when `b` is divided by `a`, for a pair of single numbers `a` and `b` (notice that the divisor has changed sides). And its definition on numerical arrays is by *scalar extension*. For example,

```
      w10
14 76 46 54 22 5 68 94 39 52
      3|5
2
      2|v10
1 0 1 0 1 0 1 0 1 0
      v10|w10
0 0 1 2 2 5 5 6 3 2
      0.25|v10
0 0 0 0 0 0 0 0 0 0
      0.4|v10
0.2 0 0.2 0 0.2 0 0.2 0 0.2 0
      5|w10+0.1
4.1 1.1 1.1 4.1 2.1 0.1 3.1 4.1 4.1 2.1
```

We see that when the left argument is `2` the result of the *residue* function just indicates which element of the right argument is an *odd* number and which is an *even* number. The residue function is also called the *modulo* function.

The monadic counterpart to the *residue function* is the ***absolute value*** *function* `|`, i.e. given a number `a`, `|a` is equal to `a` if `a≥0` or equals to `-a` if `a<0`. So `|a` is always *non-negative*, it just flips the sign of a number if it is a *negative* number. The usual mathematical notation for the *absolute value* of a number a is $|a|$. For example,

```
      |v10
1 2 3 4 5 6 7 8 9 10
      |_2.1 0 7.5 _10
2.1 0 7.5 10
```

It would be handy if there is a function in ELI which converts a floating-point number `a` to its nearest integer `ai`. Indeed, there are two primitive monadic functions, ***floor*** `_.` , and ***ceiling*** `~.` to do precisely that in ELI. For `a∈` R,

- `_.a` is the *largest number* in Z which is less than or equal to `a`, i.e. `_.a` is the *nearest integer* approximating `a` from below in terms of size.
- `~.a` is the *smallest number* in Z which is greater than or equal to `a`, i.e. `~.a` is the *nearest integer* approximating `a` from above in terms of size.

It is clear that for all `a`$\in Z$, `_.a` and `~.a` *map* `a` into itself. For example,

```
      _. 1.2 2.7 5 _2.3 _10.9
1 2 5 _3 _11
      ~. 1.2 2.7 5 _2.3 _10.9
2 3 5 _2 _10
```

So for a positive number *a*, *_.a* just gets rid of the fractional part of *a*, but for a negative *a*, *_.a* adds `_1` to it after getting rid of the fractional part. On the other hand, for a negative number *a*, *~.a* just gets rid of the fractional part of *a*, but for a positive *a*, *~.a* adds `1` to it after getting rid of the fractional part. To name these two functions *floor* and *ceiling* and assign the two-character symbols `_.` and `~.` to them is quite suggestive for their functionalities. This is much clearer than to name a similar functions such as ***int***(*a*) in many other programming languages.

Now one may wonder what are the dyadic versions of `_.` and `~.`? The answer is that they are the ***minimum*** and ***maximum*** *functions* in ELI. Giving two numbers *a* and *b*, *a_.b* yields smaller of the two as its result; and for two numbers *a* and *b*, *a~.b* yields larger of the two as its result. Naturally, both definitions on a pair of numbers extend to arrays element-wise as scalar functions do. We try out some samples:

```
       v10_.w10
1 2 3 4 5 5 7 8 9 10
       v10~.w10
14 76 46 54 22 6 68 94 39 52

       v10_.w10*0.1
1 2 3 4 2.2 0.5 6.8 8 3.9 5.2
       v10~.w10*0.1
1.4 7.6 4.6 5.4 5 6 7 9.4 9 10
```

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 trade 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 we do the following:

```
      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
```

2. Comparisons, Compress and Operators

2.1 Comparisons of data

We have already briefly seen the *equality* function `=` in section 1.4 during our study of the precision of the divide function where the two operands are numbers. The equality function is a scalar function and all other scalar functions we encountered in the last chapter take only numeric arguments. For the equality function, not only it can take non-numeric arguments but it can also take two arguments of different data types or even different data structures. Let us illustrate this point by the following examples and note that the results are all Boolean numbers:

```
      v10<-!10
      v10=5
0 0 0 0 1 0 0 0 0 0
      'abcde'='awcue'
1 0 1 0 1
      `abcde=`awcue
0
      `abcde='abcde'
0 0 0 0 0
      v10='abcde'
length error
      v10='abcde'
         ^
      v10=10#'abcde'
0 0 0 0 0 0 0 0 0 0
      3 8=3 9
1 0
      3 8=(3;8)
0 0
      2015.02.07=2015.01.31+!7
0 0 0 0 0 0 0 1
      2015.02.07=20150131+!7
0 0 0 0 0 0 0 0
```

- The first comparison is easy to understand as only the fifth element of `v10` is equal to `5`; so is the second comparison since the first, the third and the last characters are the same on both sides.
- The third comparison yields all `0` because the left side is a scalar of type symbol while the right side is a vector of type character, or a string; they are not the same.
- The 4th comparison fails because one side is a vector of length `10` while the other is a vector of length `5`.
- The 5th comparison can be carried out but yields all `0` because one side is of numeric type while the other is of type character.
- The 6th comparison is easy as the first pair of numbers to compare is equal while the second pair is not.
- The 7th comparison gives `0 0` as the left side is a vector of length `2` while the right side is a list.
- In the 8th comparison, the left side is the date of Feb. 7, 2015 which the right side is seven days following Jan. 31.
- The last comparison gives 7 `0`s because the right side is a vector of 7 numbers, not dates.

In ELI, you can also compare two data items by size. There are four dyadic scalar functions to do that: ***less than*** `<`, ***greater than*** `>`, ***less than or equal*** `<=` and ***greater than or equal*** `>=`, , the last two are represented by two character symbols. We can see a few examples:

```
      'abcde'<='awcue'
1 1 1 1 1
      'abcde'<'awcue'
```

```
0 1 0 1 0
      2015.01.31<2015.02.07
1
      3 5 6 20 28>2*!5
1 1 0 1 1
      3 5 6 20 28>=2*!5
1 1 1 1 1
      3 5 6 20 28<2*!5
0 0 0 0 0
      3 5 6 20 28<=2*!5
0 0 1 0 0
      0.3333333333<1%3
1
```

The *comparison* of the *size* of two character data items is carried out according to ***lexicographical*** *order*, i.e. 'a' is ahead of 'b', etc. The *size* of *dates* is determined by that the *older* date is *less than* a *more recent* date.

2.2 Boolean operations

We see from the previous section that all comparisons result in Boolean values, be they single items or arrays. In this section we first introduce three operations among Boolean data which yield Boolean results. The three Boolean functions on Boolean data are the monadic ***not*** function ***~****a*, and the dyadic ***and*** function *a*^*b* and the dyadic ***or*** function *a*&*b*, for Boolean operands *a* and *b*.

From *elementary logic* we have

- ***not*** **true** implies **false** and ***not*** **false** implies **true**. Hence, `~1` is `0` and `~0` is `1`.
- **true** ***and*** **true** implies **true**, **true** ***and*** **false** implies **false** and **false** ***and*** **false** implies **false**. Hence, `1^1` is `1`, `1^0` is `0` and `0^0` is `0`.
- **true** ***or*** **true** implies **true**, **true** ***or*** **false** implies **true** and **false** ***or*** **false** implies **false**. Hence, `1&1` is `1`, `1&0` is `1` and `0&0` is `0`.

For more general vectors, we have

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

There is one more comparison function called ***not equal*** (`~=`) we have not mentioned yet; its meaning is fairly easy to figure out: *a*~=*b* if *a*=*b* is not true for two data items *a* and *b*. In other words, we have the following formula:

$$a\sim{=}b \quad \leftrightarrow \quad \sim a{=}b$$

Combined with other comparison functions introduced in the previous section we have more identities as follows:

$$\begin{aligned} a{>}{=}b &\quad \leftrightarrow \quad \sim a{<}b \\ a{<}{=}b &\quad \leftrightarrow \quad \sim a{>}b \\ a{>}{=}b &\quad \leftrightarrow \quad (a{>}b)\,\&\,(a{=}b) \\ a{<}{=}b &\quad \leftrightarrow \quad (a{<}b)\,\&\,(a{=}b) \end{aligned}$$

for conformable data items (singletons or arrays) *a* and *b*. One certainly can try them out in ELI.

2.3 Boolean selection

Booleans are useful because they can be used as the left argument *b* of the dyadic function ***compress*** *b*`/`*a* to *select* the elements in its right argument *a* which correspond to `1`s in the left argument *b* and eliminates the elements in *a* corresponding to `0`s in *b*. We illustrate this *selection* functionality of `/` by the following examples:

```
      (' '~=cv)/cv<-'from sea to shining sea.'
fromseatoshiningsea.
      (0=2|v10)/v10<-!10
2 4 6 8 10
      ((2<v10)^9>v10)/v10
3 4 5 6 7 8
      ((0=2|v10)&0=3|v10)/v10<-!10
2 3 4 6 8 9 10
```

In the first *compress*, the Boolean expression on left of `/` yields a Boolean value with `1`s indicating a character in `cv` which is not a blank. Hence, it results in the elimination of all blanks in `cv`. In the second *compress*, the expression on the left of `/` gives a `1` to all even numbers in `v10`, i.e. those when divided by `2` end up having a remainder `0`. Hence, the second compress selects all even numbers in `v10`. The Boolean vector on the left of `/` of the third *compress* is b_1 ***and*** b_2 with b_1 indicating elements in `v10` which are greater than `2` and b_2 indicating elements in `v10` which are less than `9`. Hence, it selects elements in `v10` which are between `2` and `9`. The Boolean vector on the left of `/` of the last *compress* is b_1 ***or*** b_2 with b_1 indicating elements in `v10` which are even numbers and b_2 indicating elements in `v10` which can be evenly divided by `3`. Hence, it selects elements in `v10` which are multiples of `2` plus those which are multiples of `3`.

From the examples above it is clear that the left and right operands of the *compress* function must be vectors of the *same* length (later we shall see how to extend this function to the case where the right operand is an array of higher dimension). But the compress function is not a *scalar* function; it is called a ***mixed function***. Its definition is almost like defining a subset *S* of the set S_r represented by the right operand with left operand representing a proposition *P* about elements of S_r:

$$S = \{x \mid P(x) \text{ holds for } x \in S_r\}$$

and the proposition can be a logical combination of several propositions about elements in S_r connected by ***not***, ***and***, ***or***. But we must point out vectors are not sets. First, a vector (of numbers, or characters or symbols) can have duplicate elements while in a set each element is unique. Second, there is the concept of *position* in a vector, and the definition of the compress function very much depends on that of corresponding position of each element in it. We will continue this discussion about the difference between sets and vectors later when we touch on lists. Shift back our attention to more practical matters we notice that the left side of compress almost always has to be enclosed by a pair of parenthesis, unless it is a literal Boolean vector or a variable holding a boolean value. This is due to the fact that ELI executes from *right to left* and if the left argument to a function, primitive or defined, is an *expression* other than a literal or a variable, then that expression must be put inside a pair of parenthesis. This not only applies to `/`, but also to the dyadic functions `^` and `&` as well as results of the monadic function `~`.

Now let us turn to the case that the right argument *a* to compress *b*`/`*a* is an array. For simplicity, we assume that *a* is a matrix. The left argument *b* to the compress function must always be a Boolean vector (it can be the scalar `0` or `1` but it then just extends that to a length suitable for the right argument). We see examples first

```
      m<-4 7#!28
```

```
      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 25 26 27 28
      0 1 0 1 0 0/m
length error
      0 1 0 1 0 0/m
                ^
      0 1 0 1 0 0 1/m
 2  4  7
 9 11 14
16 18 21
23 25 28
      ms<-2 4#`jones `backer `dean `collins `smith
      ms
`jones `backer `dean   `collins
`smith `jones  `backer `dean
      1 0 1 1/ms
`jones `dean   `collins
`smith `backer `dean
```

We see that b/a selects *columns* of a according to positions of `1`s in b. This of course requires that the *length* of b and the *width* of a (for more general array a, the length of the *last dimension* of a) to be equal. Otherwise, a *length error* would result as in the case of the first compress above.

What to do if we want to select rows of a matrix instead of columns? In fact, ELI has a different compress function called ***compress along the first axis*** denoted by `/.` to do precisely that: For a Boolean vector b whose length equals to the first dimension of the *shape* of the right argument a (it is just the height of a in case a is a matrix), $b/.a$ selects sub-arrays of a along the first axis of a (i.e. rows of a in case a is a matrix) which correspond to `1`s in b. We illustrate this new compress function on matrices by the following two examples:

```
      ms1<-5 4#`jones `backer `dean `collins `smith
      ms1
`jones   `backer  `dean    `collins
`smith   `jones   `backer  `dean
`collins `smith   `jones   `backer
`dean    `collins `smith   `jones
`backer  `dean    `collins `smith
      1 0 1 1/.m
 1  2  3  4  5  6  7
15 16 17 18 19 20 21
22 23 24 25 26 27 28
      0 1 1 0 1/.ms1
`smith   `jones `backer  `dean
`collins `smith `jones   `backer
`backer  `dean  `collins `smith
```

2.4 The reduction operator and mathematical induction

The *slash* character symbol `/` used in the preceding section for the compress function has a different meaning in ELI if there is a primitive function symbol, such as `+`, immediately to the left of it. For example,

```
      +/v10
55
```

In this case, `/` is an ***operator***, more precisely the ***reduction operator***, which when applies to a *dyadic primitive scalar* function *f* on its left produces a ***derived function*** *f*`/`. When f is `+` the *derived function* `+/` is what we call the ***sum***(.) function in other programming languages. More precisely, for a dyadic scalar function *f*, the value of the result when derived function *f*`/` applied to a vector *v* with elements $v_1, v_2,\ldots,v_n$ is defined by

$$f/v \quad \leftrightarrow \quad v_1 f v_2 \ldots f v_n$$

Hence, `+/v10` is `1+2+3+4+5+6+7+8+9+10` which is `55`. Of course, *f* can be other dyadic primitive scalar functions. For examples,

```
      */v10
3628800
      w10<-10?.100
      w10
14 76 46 54 22 5 68 94 39 52
      _./w10
5
      ~./w10
94
      ^/0 1 1 0 0 1 1 0
0
      &/0 1 1 0 0 1 1 0
1
```

Hence, `*/` is the ***product*** function, `_./` is the ***total minimum*** function and `~./` is the ***total maximum*** function. And `^/`*b* is `0` unless all elements in *b* are `1`; `&/`*b* is `1` if there is a `1` in one of the elements of *b*.

We should not forget the fact that Boolean numbers are also integers, i.e. $B \subset N_0 \subset Z$. Hence, arithmetic functions, and consequently, *f*`/` for an arithmetic function *f* also applies to Boolean vectors *b*. It is easy to see that `*/` has the same effect on a Boolean vector *b* as that of `^/`. But `+/` (we call it *plus reduction* instead of *sum*) is far more interesting since it counts the `1`s in a Boolean vector *b*, and this is very useful when combined with comparisons:

```
      +/0 1 1 0 0 1 1 0
4
      +/w10<50
5
      cv<-'from sea to shining sea.'
      +/'s'=cv
3
      +/'e'=cv
2
```

The second expression above counts the number of elements in `w10` which is less than `50` while the last two expressions above count the occurrences of the letters `'s'` and `'e'` in the string `cv` respectively.

What happens if the argument *a* to a *derived function f*`/` is an *array*, say a matrix. The answer is that *f*`/` will then apply to each *row* of *a* to yield a vector of length *r* where *r* is the number of rows in *a*, i.e. *r* is the size of the first dimension in the *shape* of *a*. This is called ***reduction along the last axis*** of *a*. And similar to the case of compress, *f*`/.`*a* for an array *a* is called ***reduction along the first axis*** of *a* and this is defined by applying *f*`/.` to each *column* of *a* to yield a vector of length *c* where *c* is the number of columns in *a* for a matrix *a*. We illustrate these two cases by the following examples,

```
      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 25 26 27 28
      +/m
28 77 126 175
      +/.m
46 50 54 58 62 66 70
      +/+/m
406
      +/,m
406
```

The last two reductions show how to get the *total sum* of all elements in a matrix like `m`.

We want to write a short function to take the average of elements in a numeric *a* which may be a scalar, a vector or a more general array. It is fairly simple. We just divide the total sum of elements in *a* by the count of *a*, i.e. the number of elements in *a*; so we have the following function:

```
      {avg1:(+/,x)%^x}
avg1
      avg1 10
10
      avg1 v2
22.125
      avg1 m
14.5
```

Back to the beginning of the example `+/v10`, a numerical vector `vp` is called an *arithmetic progression* if the difference between any two consecutive elements in `vp` is a constant (called *step*) `d`. `v10` is a progression with `d=1`. In general, a progression `vp` of length `n` and step `d` starting at `b` is of the form `b+d*!n` (`[]IO=0`); for example,

```
      []IO<-0
      10+3*!15
10 13 16 19 22 25 28 31 34 37 40 43 46 49 52
      +/10+3*!15
465
```

For a progression, you really don't need the reduction operator to get its sum. The legend has it that when the great German mathematician Carl Friedrich Gauss was a young boy in a classroom, he figured out instantly that the sum of `1..100` is `5050` to the surprise of his teacher. His reasoning is as follows: you pair the first element with the last element and the sum is `101`, you then pair the next element with the next to last element the sum is still `101` and we have `50` such pairs; so the sum is `5050`. This argument also applies to the calculation of the sum of progressions. Hence, we write the following short function to calculate the sum of a progression `b+d*!n` where the `y` argument stands for step `d` and `x` stands for the two item list `(b;n)` of starting point `b` and length `n`:

```
      {sum_pv:[]IO<-0;(b;n)<-x;(b+b+y*n-1)*n%2}
sum_pv
      1 sum_pv (1;100)
5050
      3 sum_pv (10;15)
465
```

So what is the point? We now have computer and the reduction operator, why bother with the old man's trick? But wait a minute, suppose a progression is a trillion long, the reduction operation would take a trillion additions, it may even run out of memory; yet the function `sum_pv` only takes 6 arithmetic operations. The moral of the story is that

if we do have a formula (or more generally an analytic solution of mathematical nature), it pays to deploy that before embarking on a brute force computation. Of course, the great advance in scientific research provided by the power of modern computer and its assortment of software is that for irregular sequence or stuffs without analytic solutions one can still plough on using brute force computation.

It is not difficult to convince us the correctness of the formula for the sum of a progression. But for problems of more complicated nature, if we can find a formula, we need a formal method to prove its correctness. This is where the method of ***mathematical induction*** comes in. This method concerns about a statement $S(n)$ involving natural numbers $n \in N$. First, we prove the *base case*, i.e. $S(1)$ is true. Next, we assume $S(k)$ is true, and from there to prove that $S(k+1)$ is also true. We then conclude that $S(n)$ is true for all $n \in N$. This process is also called *proof by* ***induction***. The statement we like to try on is

$$\sum^{n}_{i=1} i^2 = (n*(n+1)*(2n+1))/6$$

that is the sum of squares from 1 up to n. Clearly, when $n=1$, $1^2 = (1*(1+1)*(2+1))/6$ holds. Now assume this is true of $n=k$, i.e. the following is true:

$$\sum^{k}_{i=1} i^2 = (k*(k+1)*(2k+1))/6$$

For $n=k+1$, we have

$$\sum^{k+1}_{i=1} i^2 = ((k*(k+1)*(2k+1))/6)+(k+1)^2 = (2k^3+3k^2+k)/6 + k^2+2k+1 = ((k+1)*(k+2)*(2(k+1)+1)/6$$

Hence, the formula is true for $n=k+1$, so it holds for all $n \in N$.

2.5 The scan operator

If the *sum* function (*plus reduction*) exemplifies the *reduction* operator / then the *partial sum* function `+\` exemplifies the ***scan*** operator denoted by the back-slash \. We show two examples first:

```
      v10
1 2 3 4 5 6 7 8 9 10
      +\v10
1 3 6 10 15 21 28 36 45 55
      *\v10
1 2 6 24 120 720 5040 40320 362880 3628800
```

We can see that `+\v10` is the *cumulative sums* of elements in `v10`, and `*\v10` is the *cumulative products* of elements in `v10`. More precisely, for a dyadic scalar function f, the result when derived function $f\backslash$ applied to a vector v with elements $v_1, v_2, \ldots, v_n$ is also a vector of the same length as that of v, and its k^{th} element is defined by

$$(f\backslash v)_k \quad \leftrightarrow \quad v_1 f v_2 \ldots f v_k \quad \leftrightarrow \quad f/v_1\ v_2\ \ldots\ v_k$$

for $k=1,\ldots,n$. We have more examples:

```
      w10
14 76 46 54 22 5 68 94 39 52
      _.\w10
14 14 14 14 14 5 5 5 5 5
      ~.\w10
14 76 76 76 76 76 94 94 94
      ^\1 1 1 0 1 0 0 0
```

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

We see that _.*a* gives the *cumulative minimum* of a vector *a* and ~.*a* gives the *cumulative maximum* of *a*. For a Boolean vector *b*, ^*b* turns all elements in it into `0` once a `0` appears in *b* (scanning from left to right); and &*b* turn all elements in it into `1` once a `1` appears in *b*. These last two examples could be quite useful in text processing. Suppose we have a line of text, i.e. a character string `txl`. We would like delete all the leading blanks in `txl` but not all blanks. We'll utilize the *or scan* after a comparison *not equal* to a blank to yield a Boolean vector for compress as shown by the expression and function below:

```
      (&\' '~=txl)/txl<-'   Hello, Mr.Smith.'
Hello, Mr.Smith.
      {del_leadblks:(&\' '~=x)/x}
del_leadblks
      del_leadblks '  This is a small world.'
This is a small world.
```

Next, suppose in a programming language *L* everything to the right of the character '`#`' in a line of code is part of a comment. We would like to write a function to eliminate comments in a line of code before parsing. We deploy the *and scan* to achieve this task:

```
      (^\'#'~=cln)/cln<-'  a:=a+1; # increment a'
  a:=a+1;
      {elim_comm:(^\y~=x)/x}
elim_comm
      '#' elim_comm '  area:= length * width; # compute area'
  area:= length * width;
```

We note that we designed the `elim_comm` function to be dyadic and having the left argument to representing the character signaling the beginning of a comment. This gives us the flexibility of choice. For example, if the designer of language *L* changes his mind to use '`$`' for starting a comment our function `elim_comm` still can be used to do the job simply by calling the function with '`$`' as its left argument.

This reminds us a question: how do we put *comment lines* in ELI? The answer is that in ELI a ***comment*** starts with a double slash '`//`' and ends at the end of that line. That is everything after a '`//`' is treated as part of a comment unless this is embedded in a character string as part of data. A comment line can be a line all by itself or mixed with ELI code, i.e. a comment can follow a piece of code on its left. One does not use comments that much when interacts with the ELI interpreter as we have done so far but commenting code is a good habit when writing ELI executable files (to be explained later) and in writing defined functions. You cannot write a comment inside a short function, but you can write a comment outside a short function in a file. We have several examples:

```
//2015-02-15 version 1
    a<-^/b            //b is Boolean, a is 1 if all elements in b are 1
      cc<-'this is a test // on a string'
      cc
this is a test // on a string
//
//a function to eliminate leading blanks in a string
    {del_leadblks:(&\' '~=x)/x}
```

The derived function of *scan* on a scalar function *f*, *f*\, extends to an array *a* in a similar fashion as that of *f*/ on an array *a*, i.e. it operates row-wise on each of its rows. And a companion operator named ***scan along the first axis*** *f*\. extends to arrays in a similar fashion for a scalar function *f* as that of *f*/. on arrays.

For a scalar function, the result of reduction f/v on a vector v always ends in a scalar, and the result of scan $f\backslash v$ is always is vector of the same length as that of v. A curious question: what happens if v is an *empty vector* (`#`v is `0`)?

Before we can answer this question we need to know how to enter a vector of length `0`. There are several ways:

```
      ss<-''       //this empty vector is of chacter type
      #ss
0
      v0<-!0       //you can also do with a reshape v0<-0#2
      #v0
0
```

Now we can try out:

```
      +/v0
0
      */v0
1
```

We see that first it would not end in error but the answer depends on f: `0` is the *identity element* of `+` while `1` is the *identity element* of `*`. Hence, f/v equals to the *identity element* of the dyadic scalar function f for an empty vector v.

2.6 The each operator

To introduce the ***each*** *operator* `"` let us recall the following: We are already familiar with the fact that for a monadic *scalar* function f applying to a vector $v = (v_1\ v_2\ \ldots v_n)$ then $f\ v$ is a vector

$$(w_1\ w_2\ \ldots w_n) \qquad \text{where } w_k = f\ v_k \text{ for k=1, 2, …, n.}$$

But what if f is a defined function? And this is where the *each* operator comes in. In short, if you replace $f\ v$ above by $f\ "v$ for a *defined function* f, the above property of $w_k = f\ v_k$ still holds. In fact, this is true even if v is a *list* $(v_1\ ;v_2;\ldots;v_n)$ instead of a vector. This *each* operator also applies to dyadic defined functions similar to the way of dyadic scalar functions. And for list operands, each operator applies to primitive dyadic or monadic functions as well as to derived functions such as `+/` or `~./`. Let us look at some examples:

```
      {f1: %x*x}
f1
      f1 2
0.25
      f1 3
0.1111111111
      f1 5
0.04
      f1"2 3 5
0.25 0.1111111111 0.04
      {f2:(x+y)%2}
f2
      3 f2 5
4
      9 f2 10
9.5
      _2 f2 8
3
      3 9 _2 f2" 5 10 8
```

```
4 9.5 3
      {avg1:(+/,x)%^x}
avg1
      L<-(3;10 30;7 9 21)
      avg1"L
<3
<20
<12.33333333
      +/"L
<3
<40
<37
      ~./"L
<3
<30
<21
      _.\"L
<3
<10 10
<7 7 7
      L%"3
<1
<3.333333333 10
<2.333333333 3 7
```

We can see that *each* is indeed a very powerful operator in ELI.

3. More Mathematical Functions

3.1 The power function and roots

We all know that multiplication can be regarded as addition raised to a higher level in the sense that adding a number *n* *t* times is the same as multiplying *n* by *t* for a positive integer *t*. In ELI terms, we have

`+/t#n` ↔ `n*t` for $t \in N_0$

Of course, for general multiplication `t` can be any real number; so multiplication is an extension of doing repeated addition. Now the question is what is the function which is multiplication raised to a higher level in the sense that it stands for repeated multiplication. Indeed, the *power* function represents repeated multiplication: x^2, *x* square is x raised to power 2 and $x^3 = x \star x \star x$ is *x* raised to power 3. The dyadic primitive ELI function corresponding to the *power* function is denoted the double character symbol '`*.`'.suggesting it is multiplication (`*`) raised to a higher level So x^2 is `x*.2` and x^3 is `x*.3` in ELI. We have the following identity:

x^n ↔ `*/n#x` ↔ `x*.n` for $n \in N_0$, $x \in R$

And this identity is extended to general $n \in R$ for $n \geq 0$, and this more general function is the ***power** function*. The process of raising `x` to `n`-th power is called *exponentiation* with `n` called the *exponent* and `x` the *base*. Please note that for `n=0` the left side of the identity above is the *times reduction* over the *empty vector*, thus it equals to the identity element of the multiplication function `*`, which is `1`. Hence, `x*.0` is `1`, i.e. *power zero* of any number `x` is `1`. Since `(x*.n)*( x*.-n)` is `1=(x*.0)`, we see that `x*.-n` is `1%x*.n` for $n \in N_0$. In general, a^{-b} with a negative exponent *-b* is the *reciprocal* of a^b for $b \in R$, $b \geq 0$.

Let us first set `[]IO←0`. We then have the following examples of the power function:

```
      2*!10
0 2 4 6 8 10 12 14 16 18
      2*.!10
1 2 4 8 16 32 64 128 256 512
      2*.-!10
1 0.5 0.25 0.125 0.0625 0.03125 0.015625 0.0078125 0.00390625 0.001953125
      (!10)*.3
0 1 8 27 64 125 216 343 512 729
      2 3 5 7 11*.11 7 5 3 2
2048 2187 3125 343 121
```

With exponentiation, we can write a number *n* which is n_0 multiplied by a 1 followed *k* 0s as

$n = n_0 \star 10\ldots0 = n_0 \star 10 \star . k$

Indeed, in ELI a floating point number such as *n* above can be written in ***scaled form*** as $n_0 e k$. For example,

```
      1.2800845e3
1280.0845
      1.2800845E3
1280.0845
      1.2800845e_3
0.0012800845
```

Note that '`e`' and '`E`' are interchangeable but this letter must follow a digit and be followed by a digit or _ and a digit with no space in between. The scaled form is quite suitable for writing huge or tiny numbers with great precision.

Suppose you have 500 dollars and you deposit it at a bank with annual interest rate of 3%. How much money would you have in your account at the end of first year, the second year, the third year … if you do not withdraw any money from your account at the bank? The amount you first deposited, $500 here, is called the principal *p* and the yearly interest rate *r*, of 3% here, means that interest payment is `p*3%100` or `p*0.03`. Together with the principal *p*, at the end of the first year there would be `p+p*0.03` or `p*1.03` in your account. And if you keep both the principal and the interest in your account there would be `(p*1.03)*1.03` and `((p*1.03)*1.03)*1.03` at the end of second year and third year respectively. To put in another form, there would be `p*1.03*.2` and `p*1.03*.3` in your account at the end of second and third year respectively. To see how your money will grow in ten years, we first reset `[]IO<-1`, then we have

```
      500*1.03*.!10
515 530.45 546.3635 562.754405 579.637037 597.026148 614.93693 633.38504 652.38659 671.95819
```

We would like to put this calculation into a short function `yr_amnt_r` for general use whose left parameter is the number of years for the money to grow and the right parameter is a list of two items, the first item is the amount of principal while the second item is the annual interest in percentage point. Here is the function and two function calls.

```
      {yr_amnt_r:(p;r)<-x;p*(1+r%100)*.y}
yr_amnt_r
      3 yr_amnt_r (500;3)
546.3635
      10 yr_amnt_r (1000;2.5)
1280.084544
```

We note that the first expression in this short function, `(p;r)<-x`, assigns two parts of the right argument to local variables denoting the principal amount `p` and the interest rate `r`. The first call of the function gives us the same result as the third element in the result list of `500*1.03*.!10` and the second function call above yields the amount of money after ten year for initial deposit of $1000 at annual interest rate of 2.5%.

We have the following identity for the right argument of the power function (exponent of the exponentiation):

$$a^{(b+c)} = a^b * a^c \quad \leftrightarrow \quad a*.(b+c) \quad \leftrightarrow \quad (a*.b)*(a*.c)$$

$$(a+b)^2 = a^2+2ab+b^2 \quad \leftrightarrow \quad (a+b)*.2 \quad \leftrightarrow \quad (a*.2)+(2*a*b)+b*.2$$

First we see that the left and the right arguments to the power function are clearly not symmetric since they behave differently. Second, we see that if we set *b* and *c* in the first line of identities above both to 0.5, we then get $a = a^b * a^c$ where a^b and a^c are equal, and that means they are the square roots of *a*! In other words, `a*.0.5` is the *square root* of *a*. By the same argument, the n^{th} root of *a* is *a* raised to the power of 1/*n*, i.e. `a*.%n` in ELI terms. We have the following examples for *square* roots, *cubic* roots and roots up to 10^{th} root:

```
      (!10)*.0.5
1 1.414213562 1.732050808 2 2.236067977 2.449489743 2.645751311 2.828427125 3 3.16227766
      1 10 100 1000 10000 100000 1000000*.%3
1 2.15443469 4.641588834 10 21.5443469 46.41588834 100
      1024*.%!10
1024 32 10.0793684 5.656854249 4 3.174802104 2.691800385 2.37841423 2.160119478 2
```

In particular, we see from the above that √2 is `1.414213562`; this of course is only an approximation to the real value of √2 similar to the situation of `1%3`. However, √2 is very different from `1%3` in a more fundamental sense. Namely, √2 is an ***irrational*** *number*, i.e. it cannot be expressed as a fractional number *n* over *m* with $n,m \in Z$, *n* and *m* have no common factor other than `1`. This fact was discovered by Greek mathematicians of the Pythagorean School more than two millenniums ago, and it was a great achievement in mathematics.

Pythagorean Theorem tells us that for a right triangle *C* with the lengths of the perpendicular sides *a* and *b*, and *c* the length of the side facing the right angle, we then have the following

$$a^2 + b^2 = c^2$$

When both *a* and *b* are equal to 1, we have *c* equal to √2, and this is how they encountered for the first time this irrational number. They adopted the prove by contradiction method to show the irrationality of √2 as follows: suppose √2= *m*%*n* with *n*,*m* two integers, *n* and *m* have no common factor other than 1. We then have

$$(\sqrt{2})\star n = m \quad \text{square both sides} \rightarrow \quad 2\star n^2 = m^2$$

This implies that *m* must an even number, say $m = 2\star m_0$ or $m^2 = 4\star m_0{}^2$. Substitute back to the equation above, we have $2\star n^2 = 4\star m_0{}^2$ or $n^2 = 2\star m_0{}^2$ and this implies that *n* too is an even number. This is a *contradiction* to our initial assumption that *n* and *m* have no common factor other than 1. Hence, √2 is *irrational*.

3.2 Euler's number *e* and the exponential function

The monadic counterpart to the power function *a*⋆.*b* is the ***exponential*** *function* ⋆.*b*, which is equal to ***e***⋆.*b*, where ***e*** is the famous ***Euler's number***. In other words, ⋆.*b*, the *exponential function* applied to *b* is a shorthand for ***e***⋆.*b* with the left argument to the power function fixed to ***e*** which approximately equals to `*2.718281828`:

```
      *.1
2.718281828
      e<-*.1
      e*._1 0 1 2 3
0.3678794412 1 2.718281828 7.389056099 20.08553692
      *._1 0 1 2 3
0.3678794412 1 2.718281828 7.389056099 20.08553692
```

Before discussing the Euler number *e* in more detail, we would like to state more precisely the concept of ***limit***. Suppose we have an infinite sequence of real numbers n_k, k=1, 2, 3, …, we say that n_k is approaching a limit *lm*, written as

$$\lim_{k\to\infty} n_k = lm$$

if given an arbitrary small number ∈, there exists a k such that $|lm-n_k|<\epsilon$. In other words, the sequence n_k can get closer and closer to *lm* because one can always pick a number from the sequence $\{n_k\}$ whose difference with *lm* in absolute value is less than the number ∈. For example, `%k`, i.e. `1%k`, has limit `0` as `k`→∞. Since all representable numbers in ELI or any digital computer are really rational numbers, √2 is the limit of a sequence of increasingly accurate approximation to this irrational number.

The Euler number *e* was first discovered by Jacob Bernoulli (it is named after the Swiss mathematician Leonhard Euler due to an identity discovered by Euler which we will touch on later) when he studied *continuous compound interest*. Suppose you have $1.00 and deposit it in a bank which offers an annual rate of 100%. At the end of a year you would have (1+1) = 2 dollars, 1 from the principal and 1 from the interest payment. Now suppose the bank decides to pay interest semi-annually: at the end of six months, you get (1+0.5) dollars and at the end of the year you get $(1+0.5)^2$ dollars. If the bank decides to pay interest quarterly, you would have $(1+0.25)^4$ dollars at the end of a year. That would be 2.25 and 2.4414 dollars respectively. Clearly, the more periods to divide a year to pay interest the more money you would have due to *compounding*. What happen if we keep increasing the number of equal periods for compounding interest payment?

Let us write a monadic short function `compr1` whose right argument `x` represents the number of equal periods in a year to pay interest and the result is the amount of money you would have by depositing one dollar at an annual rate of 100%:

```
      {compr1: (1+%x)*.x}
compr1
      compr1 2
2.25
      compr1 4
2.44140625
      compr1 12 //pay interest monthly
2.61303529
```

In order to see the overall trend of `compr1 n` when `n`→∞ we note that the function `compr1` can also take a vector argument to produce a vector result similar to the way of monadic scalar function. Now, we can try it out:

```
      compr1 2 4 12
2.25 2.44140625 2.61303529
      10*.!8
10 100 1000 10000 100000 1000000 10000000 100000000
      compr1 10*.!8
2.59374246 2.704813829 2.716923932 2.718145927 2.718268237 2.718280469 2.718281694 2.718281798
```

The **lim** $_{n\to\infty}$ `compr1 n` is the *Euler number* ***e*** which is truncated to `2.718281828` when displayed in ELI. In the IPO filing for Google in 2004, the company said it intended to raise $`2,718,281,828` which is *e* billion dollars rounded to the nearest dollar. The most accurate calculation of *e* has been carried out in 2010 with `1E12` decimal digits.

What if the annual interest rate for continuous compound is `r=%a` where `a` is a positive integer; for example if `a` is 2 then `r` is 50%. The growth then is the limit of `(1+(%a)*%n)*.n` which is `(1+%a*n)*.n` as `n`→∞, and this again can be rewritten as follows (remember that $c^{b*a} = (c^b)^a$):

`(1+%a*n)*.(a*n)*%a` ←→ `((1+%a*n)*.(a*n))*.%a`

Note that **lim** $_{n\to\infty}$`(1+%a*n)*.(a*n)` is the same as **lim** $_{n\to\infty}$`(1+%n)*.n`. Hence, when `n`→∞ the second expression above becomes e^r, `r= %a` for a positive integer `a`. Similarly, we can prove that if the rate of growth `r=b` is a positive integer then continuous compound results in e^r for total growth. And for a rate of growth `r= b` being a positive integer (say `b` is 3 then `r` is 300%), continuous compounding then leads to the growth which is the limit of `(1+b%n)*.n` as `n`→∞, and this expression can be rewritten as follows:

`(1+%(%b)*n)*.((%b)*n)*b` ←→ `((1+%n%b)*.(n%b))*.b`

Note again that **lim** $_{n\to\infty}$`(1+%n%b)*.(n%b)` is the same as **lim** $_{n\to\infty}$`(1+%n)*.n`. Hence, the second expression above is e^r when `n`→∞ for `r= b`, i.e. continuous compound at rate `r= b` results in e^r for total growth for a positive integer `b`. Combine these two results, we see that the above also holds for `r=b%a` a positive rational number. In general, we have

$$growth = e^{rate*time}$$

The *time* period here needs not to in whole number of years, it can be any time interval since compounding in growth is continuous.

The idea of using continuous compounding to represent the rate of growth also applies to decay. Suppose some material of weight *w* decay at an annual rate of 100%. After continuous decay at that rate what would be the weight of that material at the end of a year? Zero? Not really. We can calculate roughly at the end of 6 months it would it would weigh 0.5*w*, and at the end of the third quarter it would weigh 0.375*w*. In fact, we can write a function as before and apply it to a sequence of larger and larger numbers of dividing periods:

```
      {compr_1:(1-%x)*.x}
compr_1
      compr_1 1 100 10000 1000000 100000000
0 0.3660323413 0.3678610464 0.3678792572 0.3678794375
```

And we see it is approaching e^{-1}:

```
      *._1
0.3678794412
```

Hence, the following formula is for calculating decay

$$growth = e^{-rate*time}$$

3.3 The logarithm and natural logarithm functions and groups

We can regard the *division* function `%` as the *inverse* of the *multiplication* function `*` in the sense that if `a*b= c` then `c%b= a`. In this sense, the ***logarithm*** *function* *b*`%`.*a* is the inverse of the *power* function *b*`*`.*a*; namely, for positive numbers *a* and *b*, *b*`%`.*a*= *c* if *b*`*`.*c*= *a*, just as '`*`.' is the symbol for the power function '`%`.' is the symbols for the logarithm function written mathematically as $\boldsymbol{log}_b\, a$ with ***base*** *b*. We illustrate this function by the following examples:

```
      []IO<-0
      2*.!11
1 2 4 8 16 32 64 128 256 512 1024
      2%.1 2 4 8 16 32 64 128 256 512 1024
0 1 2 3 4 5 6 7 8 9 10
      10*.!8
1 10 100 1000 10000 100000 1000000 10000000
      10%.1 10 100 1000 10000 100000 1000000 10000000
0 1 2 3 4 5 6 7
      2 3 8*.5
32 243 32768
      2 3 8%.32 243 32768
5 5 5
      2 3 8%.24 27 30
4.584962501 3 1.635630199
      10%.17 190 2015 3.16227766
1.230448921 2.278753601 3.30427505 0.5
```

Like other arithmetic functions the logarithm function *b*`%`.*a* is a scalar function. So it operates on (conformable) arrays the same way as it operates on a pair of scalars and scalar extension applies if one argument is a scalar. Many times when a sequence of data is growing very fast like the right argument in the 4th expression above, it would be impossible to plot its growth on a chart. Instead, we can plot its logarithmic growth to indicate the scale of growth.

The *monadic* counterpart to the *logarithm* function is the ***natural logarithm*** function `%`.*a* which is the logarithm function *b*`%`.*a* with the left argument *b* fixed to be *e*, the Euler number; mathematically, it is written as $\boldsymbol{log}_e a$ or ***ln*** *a*.

```
      e<-*.1
      e
2.718281828
      10*!8
10 20 30 40 50 60 70 80
      e%.10*!8
2.302585093 2.995732274 3.401197382 3.688879454 3.912023005 4.094344562 4.248495242 4.382026635
      %.10*!8
2.302585093 2.995732274 3.401197382 3.688879454 3.912023005 4.094344562 4.248495242 4.382026635
      %.(50*800)
10.59663473
      (%.50)+%.800
10.59663473
      (%.0.05)+%.2000
4.605170186
      (%.0.05*2000)
4.605170186
      (%.0.05)
_2.995732274
      %.2000
7.60090246
```

Note the last several expressions above illustrate the general identity that

$$\boldsymbol{ln}\ (a*b) = \boldsymbol{ln}\ a + \boldsymbol{ln}\ b \qquad (1)$$

for two positive real numbers a and b because we have $e^{\boldsymbol{ln}\ x} = x$ for all $x>0$, hence raise e to the power of the left and right sides of the identity (1) we get $e^{a+b} = e^a * e^b$.

$$e^{\boldsymbol{ln}\ (a*b)} = a*b = e^{\boldsymbol{ln}\ a} * e^{\boldsymbol{ln}\ b} = e^{\boldsymbol{ln}\ a + \boldsymbol{ln}\ b} \quad \rightarrow \quad \boldsymbol{ln}\ (a*b) = \boldsymbol{ln}\ a + \boldsymbol{ln}\ b$$

A *map* $\boldsymbol{m}$ from a set X to a set Y is called a *one-to-one correspondence* between X and Y

if for every $x \in X$, $\quad \boldsymbol{m}(x) \in Y$
and for every $y \in Y$, there is a $x \in X$ such that $\boldsymbol{m}(x) = y$

We can see that such a one-to-one correspondence $\boldsymbol{m}$ is a *pairing* between the set X and the set Y. Two set S_1 and S_2 are called ***equivalent*** if there is a one-to-one correspondence between them. Obviously, *finite* sets having the same number of elements are equivalent to each other since we can set up a one-to-one correspondence between them. For two *infinite* sets being *equivalent* means that they essentially have the same number of elements as well. For example, the set N of natural numbers and the set $N_0 = N \cup \{0\}$ are equivalent since $\boldsymbol{m}(n) = n\text{-}1$ from N to N_0 is a one-to-one correspondence between the two sets. Let

$$N_{ev} = \{2, 4, 6, \ldots.\}$$

be the set of even numbers; then $\boldsymbol{m}(n) = 2n$ is a one-to-one correspondence between N and N_{ev}. Hence, N, N_0 and N_{ev} are all equivalent to each other. A set is called ***countable*** if it is equivalent to the set N of natural numbers. By laying out all fractional numbers in a matrix of rows where each row has the same denominator we can prove that the set R_0 of rational numbers is countable, i.e. we can devise a one-to-one correspondence between R_0 and N_0.

And using a *diagonal method* first devised by Georg Cantor, the founder of set theory, one can prove that the set R of real numbers is not countable. In other words, there are infinite sets which have more elements than a countable set. We are not going into details of these proofs.

A *binary operation* ⍟ on a set *S* is a *map* from a pair of elements (*x*, *y*) in *S* to an element *x*⍟*y* in *S* for all pairs of elements in *S*. Both *addition* + and *multiplication* × are binary operations on Z (and on *R*), but *division* is not a binary operation on *Z* or *R* because *x*%0 is not an element in *Z* or *R*. A ***group*** is a set *G* with a *binary operation* ⍟ on *G* such that it satisfies the following three conditions:

1) $x⍟(y⍟z) = (x⍟y)⍟z$ for all elements *x*, *y*, *z* in G;
2) there is an element *id* in *G* such that $id⍟x = x⍟id = x$ for each element *x* in *G*; and
3) for each element *x* in *G* there is an element *y* in *G* such that $x⍟y = y⍟x = id$.

Property 1) is called the *associative law*. The element *id* is called the *identity element* of *G*, and the element *y* in 3) is called the *inverse* of *x* in *G*. Note that it is not required for a group operation in G that $x⍟y = y⍟x$ for all *x, y* in *G*. One example of a group is *Z*, the set of all integers, under the binary operation of addition + where 0 is the identity element and the inverse of an *x* in *Z* is just -*x*. This also holds true if we replace *Z* by *R*. However, Z is not a group under the binary operation of multiplication × because many elements in *Z* do not have an inverse in *Z* under × where 1 is the identity element. The same is also true for *R* because the element 0 has no inverse under ×. If we define R^+ to the set of positive real numbers, namely

$$R^+ = \{x \mid x \in R, \text{x}>0\}$$

Then R^+ is a group under multiplication since we clearly have $x×(y×z) = (x×y)×z$ for all elements *x*, *y*, *z* in R^+ with 1 as the *identity element*, and for each $x \in R^+$, %*x* is the inverse of *x*.

An ***isomorphism*** from a group *G* to a group *H* is a one-to-one correspondence *m* between *G* and *H* such that

$$m(x⍟y) = m(x)⍟m(y) \quad \text{all } x, y \text{ in } G.$$

Note that the ⍟ in the left side of the above expression is the binary operation in *G* while the ⍟ in the right side of = above is the binary operation in *H*. Two groups G and H are *isomorphic* if there is an isomorphism from G to H. We see that isomorphic groups are basically the same group as far as group structure is of concern. In particular, for two finite groups to be isomorphic them must have the same number of elements. With all the discussion above, we can see now that the *natural logarithm* function ***ln***(*x*) is an *isomorphism* from the group R^+ under multiplication × operation to the group *R* under addition operation + since $x \to$ ***ln***(*x*) is a one-to-one correspondence between R^+ and *R* and we have ***ln*** $(a×b) =$ ***ln*** $a +$ ***ln*** *b* for all *a*, *b* in R^+.

3.4 Complex Numbers

The simple equation: $ax + b = 0$ has $x = -b\%a$ as its solution for real number coefficients *a* and *b* with *a*>0. The *quadratic equations* are those of the form

$$ax^2 + bx + c = 0 \quad \text{where } a, b \text{ and } c \in R \text{ with } a>0.$$

These equations have two solutions x_1 and x_2 (expressed in ELI notation) as follows:

$$x_1 = ((-b) + ((b×.2) + 4×a×c)×.0.5) \%2×a$$
$$x_2 = ((-b) + ((b×.2) - 4×a×c)×.0.5) \%2×a$$

The solutions to the general *cubic* (third degree) equation

$$ax^3 + bx^2 + cx + d = 0, \quad a>0$$

and the general *quartic* (fourth degree) equation

$$ax^4 + b\,x^3 + c\,x^2 + dx + e = 0, \qquad a>0$$

are also known by sixteenth century but their formulae are far more complicated. The Norwegian mathematician N. Abel showed in 1824 that there can be no formulae for solving general equations of degree higher than four. Of course, some particular quartic equations do have solutions in terms of the algebraic operations we studied so for. For example, for $ax^4 + e = 0$, we have $x = (-e\%a)*.0.25$. It was the French mathematician E. Galois (1811-1832) who introduced the concept of the *group* (as we described in the previous section) to study the properties of algebraic equations in order to decide which equations have algebraic solutions in terms of their coefficients.

In all our discussion about solutions to algebraic equations above, we actually neglected an important question: where do the solutions lie? In R, the set of real numbers, as the coefficients are? If we just consider the quadratic equation $x^2 + 1 = 0$; then $x = \sqrt{-1}$. We clearly have a problem here if we only search x in R since for any $x \in R$, $x^2 > 0$ or $x^2 = 0$. For this reason mathematicians developed the system C of complex numbers. Mathematically, a ***complex number*** consists of two parts: the real part a and the imaginary part b and written as $a+b\boldsymbol{i}$ where $a, b \in R$ and $\boldsymbol{i}^2 = -1$. If $b = 0$, then $a+b\boldsymbol{i}$ is just a real number a. We now have

$$B \subset N_0 \subset Z \subset R_0 \subset R \subset C$$

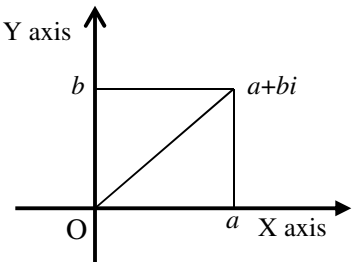
However, R is more than a set, it has operations such as addition `+` and multiplication `*`. We want to extend these operations to C consistent with R and the fact that $\boldsymbol{i}^2 = -1$.

Hence, for two complex numbers $a+b\boldsymbol{i}$ and $c+d\boldsymbol{i}$, we define

$$(a+b\boldsymbol{i}) + (c+d\boldsymbol{i}) = (a+c) + (b+d)\,\boldsymbol{i}$$

and

$$(a+b\boldsymbol{i}) * (c+d\boldsymbol{i}) = (ac-bd) + (ad+cb)\,\boldsymbol{i}$$



Complex Plane

The extension of the set R of real numbers to the set C of complex numbers is a continuation of the extension from natural numbers N to integers Z, and then to the set of rational numbers R_0. Each time it is in order to accommodate the solutions to operations from subtraction to division, and finally to taking the root of a negative number.

A complex $a+b\boldsymbol{i}$ is written in ELI as `ajb` where `a` is the real part and `b` is the imaginary part. If that complex number has no imaginary part, i.e. it is a real number, then we can write it as `aj0` or just `a`. On the other hand if $a=0$, then we must write it as `0jb`. Note that there must not be a space before or after `j`. For example,

```
      _1*.0.5
0j1
      _1j2 5.2j_2 0.01j5.8 20.1j0.5 + 2.3j3
1.3j5 7.5j1 2.31j8.8 22.4j3.5
      _1j2 5.2j_2 0.01j5.8 20.1j0.5 * 2.3j10
_22.3j_5.4 31.96j47.4 _57.977j13.44 41.23j202.15
      1 2 3 4+_1j2 5.2j_2 0.01j5.8 20.1j0.5
0j2 7.2j_2 3.01j5.8 24.1j0.5
      0j1 0j2 0j3 0j4+_1j2 5.2j_2 0.01j5.8 20.1j0.5
```

```
_1j3 5.2 0.01j8.8 20.1j4.5
      1 2 3 4*_1j2 5.2j_2 0.01j5.8 20.1j0.5
_1j2 10.4j_4 0.03j17.4 80.4j2
      0j1 0j2 0j3 0j4*_1j2 5.2j_2 0.01j5.8 20.1j0.5
_2j_1 4j10.4 _17.4j0.03 _2j80.4
```

We notice that in case that imaginary part is either a negative number or a fractional number less than 1, their first character '_' or '0' must follow 'j' immediately.

Since the division function can be defined as the inverse to the multiplication function and initially for positive integers the power function can be defined as repeated multiplication, they can also naturally be extended to complex numbers. For example,

```
      _22.3j_5.4 31.96j47.4 _57.977j13.44 41.23j202.15%2.3j10
_1j2 5.2j_2 0.01j5.8 20.1j0.5
      1.3j5 7.5j1 2.31j8.8 22.4j3.5 % 2 2j5 0j4 10j2
0.65j2.5 0.6896551724j_1.224137931 2.2j_0.5775 2.221153846j_0.09423076923
      1.3j5 7.5j1 2.31j8.8 22.4j3.5 *. 2
_23.31j13 55.25j15 _72.1039j40.656 489.51j156.8
      1.3j5 7.5j1 2.31j8.8 22.4j3.5 *. 0.5
1.798087392j1.390366236 2.744665096j0.1821715884 2.388319197j1.842299809 4.747198576j0.3686384658
```

We introduce a new monadic function + called ***conjugate***. For a complex number (a+b***i***) its *conjugate* equals to (a–b***i***). Hence, for real numbers, this function is just the identity function, i.e. it gives back the argument as its result. For true complex numbers, we have

```
      +0j1
0j_1
      +_1j2 10.4j_4 0.03j17.4 80.4j2
_1j_2 10.4j4 0.03j_17.4 80.4j_2
```

Recall that the *absolute value* of a real number a is a if a>0, –a if a<0 and 0 if a= 0. We can interpret this as the length from the point representing a in the real number line R to the point representing 0. We can regard a complex number as a point in a plane, the *complex plane*, with the real part as the horizontal x-coordinate and the imaginary part as the vertical y-coordinate. We can define the absolute value of a complex number a+b***i*** as the distance of that point from the origin at (0,0), i.e. it is equal to

$$\sqrt{(a^2+b^2)}$$

In ELI, see the following for the absolute value function's extension to complex numbers:

```
      |3j4
5
      |_3j4 3j_4 0j5 _5 10j_5
5 5 5 5 11.18033989
```

In a similar vein, the value of the *signum* function of a real number a (1 if a>0, –1 if a<0 and 0 if a= 0) can be interpreted as the unit vector from the origin 0 to the point a in R. And this definition of the *signum* function can be readily extended to complex numbers. For example,

```
      *3j4
0.6j0.8
      *_3j4 3j_4 0j5 _5 10j_5
_0.6j0.8 0.6j_0.8 0j1 _1 0.894427191j_0.4472135955
```

Not all primitive functions on *R* can be extended to *C*. For example, the *maximum* and *minimum* functions run into problems for complex numbers:

```
      2j4 _. 3j5
domain error
      2j4 _. 3j5
           ^
      2j4 ~. 3j5
domain error
      2j4 ~. 3j5
           ^
```

3.5 Trigonometric functions

Generally speaking, ***trigonometry*** is the study of *triangles*, or more precisely the relationship between the angles and lengths of sides of a triangle, and *trigonometric functions* are those involved in these computations. To simplify the situation, people usually concentrate the study of this relationship to an angle on a unit circle, i.e. a circle of radius 1, and the trigonometric functions compute the lengths of the sides facing or underlying that angle. And for this reason, the trigonometric functions are also called ***circle functions*** in APL and their representation involved the character symbol `○` for which the corresponding ASCII character in ELI is `@`. There is only one monadic function `@` in ELI but several dyadic `@` functions as we shall soon see.

Let us first try

```
      @1
3.141592654
      @!5
3.141592654 6.283185307 9.424777961 12.56637061 15.70796327
```

We recognize that `@1` is π (*pi*) and `@x` is just *x* times π. We know that for a circle *C* of radius *r*, the length of the circumference of *C* is $2\pi r$, the area of *C* inside its circumference is πr^2 and the volume of a sphere of radius *r* is $(4/3)\,\pi r^3$. Clearly, π is a very important mathematical constant and like the Euler's number ***e*** it is a *transcendental* (hence, an irrational) number, i.e. it is not a solution to a polynomial equation with complex number coefficients. How π was computed in the first place? Around 1400, the following formula for π was discovered:

$$\pi = (4/1) - (4/3) + (4/5) - (4/7) + \ldots$$

However, using this formula one has to calculate many terms to get a good approximation of *pi*. A better one which computes much faster (in terms of getting the number of digits quickly) is the ***Bailey-Borwein-Plouffe*** *formula*:

$$\pi = \sum_{k=0}^{\infty} [(1/16^k)((4/(8k+1)) - (2/(8k+4)) - (1/(8k+5)) - (1/(8k+6))]$$

We write a short function to compute the item in this summation (note that k is replaced by the implicit parameter `x`) and make some try run:

```
      {f:(%16*.x)*(4%1+8*x)+(_2%4+8*x)+(_1%5+8*x)-%6+8*x}
f
      f 0
3.133333333
      f 1
0.008089133089
      f"0 1
3.133333333 0.008089133089
```

```
      +/f"0 1
3.141422466
```

Note that we used the *each* operator `"` introduced in section 2.6: `f"0 1` is `f(0) f(1)`. We write another short function `BBN` to compute the sum

$$\sum_{k=0}^{n} [f(k)]$$

with parameter *x* replacing `n` and try out on some sample data:

```
//!x here is the vector 0 1 .. x-1
     {BBP:[]IO<-0;+/f"!x}
BBP
      BBP 2
3.141422466
      BBP 3
3.14158739
      BBP 5
3.141592645
      BBP 8
3.141592654
      @1
3.141592654
```

We see that when `n` reaches 8 the sum calculated from `BBP 8` is already indistinguishable from π as far as displayed precision of ELI is of concern.

In ELI *version* **0.3**, we implemented the system function called *printing precision*, `[]PP`, which controls the maximum number of digits to be displayed after the decimal point '.' of a floating point number. As we can see from the above display that in a *clear workspace*, the default value of `[]PP` is `10`. But this value of `[]PP` can be changed if one would like to see a *more precise* display (or a *shorter* display) of a fractional or a real number. In order to see a more accurate picture of how fast the *Bailey-Borwein-Plouffe formula* computes π we increase `[]PP` to `15` as follows:

```
      []PP
10
      []PP<-15
      BBP 8
3.14159265358897
      @1
3.14159265358979
      BBP 9
3.14159265358975
      BBP 10
3.14159265358979
```

We then realize that `BBP 8` and π differ in the last 3 digits, and only when BBP formula adds up 10 items it gives the same number to the 15 decimal place as that of π. As of late 2013, people have already computed π over `13.3` *trillion* (10^{13}) digits.

A quite different way to compute π using the so called Monte-Carlo method using ELI can be found in **§4.2** of W. Ching [2] and interested reader can take a look. It is based on the fact that the area of a quadrant of the unit circle equals to π/4; then one calculates this area by throwing a bunch of random dots into the unit square and counts the ratio of the total number of dots to the number of dots that falling within the quadrant of the unit circle. That method is more understandable intuitively but computationally less efficient than the Bailey-Borwein-Plouffe formula we used above. Of course, the whole point of using the Monte-Carlo method is to solve problems with no known formula or analytic solution.

We measure the size of angle *AOB* with its vertex at a circle of radius 1 by the length *x* of the arc from *A* to *B* on the circumference of the circle; this measure *x* of angle is in *radians*. So in ***radians***, the whole circle is of radius 2 π, an angle forming a straight line is of radian π and the angle forming by two perpendicular lines is of radian π /2. Another measure of angles is by *degrees*, the corresponding measures in terms of ***degrees*** of the above are 360º, 180º and 90º. One can easily write a short function to convert the measure of an angle from radians into degrees.

```
      {r2d:360*x%@2}
r2d
      r2d" @"(%4;%2;7%6;11%4)
<45
<90
<210
<495
```

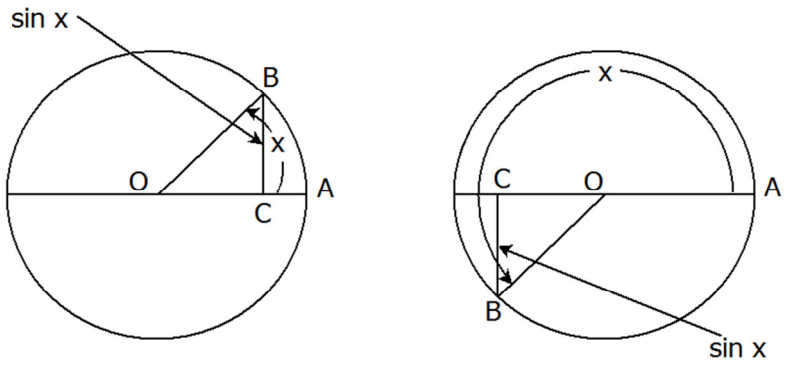
This means that for angles in radians π/4, π/2, 7π/6, 11π/4 the corresponding measure in degrees is 45^0, 90^0, 210^0, 495^0. We note that since `(%4;%2;7%6;11%4)` is a list, we need to apply the *each* operator `"` to `@` and this results in another list so as to require a second application of `"` to the function `r2d`. If what we feed `"` to a vector, the double application of *each* `"` would not be needed. Indeed, we have a primitive monadic function ***raze*** `,.` which turns a list with all numeric items into a numeric vector (for details of the *raze* function see section 2.1 of W. Ching [1]):

```
      ,.(%4;%2;7%6;11%4)
0.25 0.5 1.166666667 2.75
      r2d ,.(%4;%2;7%6;11%4)
14.32394488 28.64788976 66.8450761 157.5633937
      r2d @,.(%4;%2;7%6;11%4)
45 90 210 495
```

The main trigonometric functions are *sine* (***sin*** *x*), *cosine* (***cos*** *x*) and *tangent* (***tan*** *x*) functions and in ELI they are denoted by the following ***circle functions***:

`1@`***x***, `2@`***x***, `3@`***x***

respectively. If we draw a perpendicular line from point *B* on the unit circle towards the line *OA* and intersect it at point *C*, then the value of ***sin*** *x* is the length of *BC* and the value of ***cos*** *x* is the length of *OC* while the value of ***tan*** *x* is the ratio of the length of *BC* over that of *OC*.



Hence, we have the following examples:

```
     @0 0.5 1 1.5 2 0.25
0 1.570796327 3.141592654 4.71238898 6.283185307 0.7853981634
     1@@0 0.5 1 1.5 2 0.25
```

```
0 1 0 _1 0 0.7071067812
     2@@0 0.5 1 1.5 2 0.25
1 0 _1 0 1 0.7071067812
     3@@0 1 2 0.25
0 0 0 1 -1
```

Conversely, for a *right triangle OCB* knowing the angle *AOB* and the length of the side *BO* we can find the lengths of *BC* and *OC*. More generally, for a triangle *PQR* with the lengths of opposing sides being *p*, *q* and *r*, the *Law of Cosines* in trigonometry states that

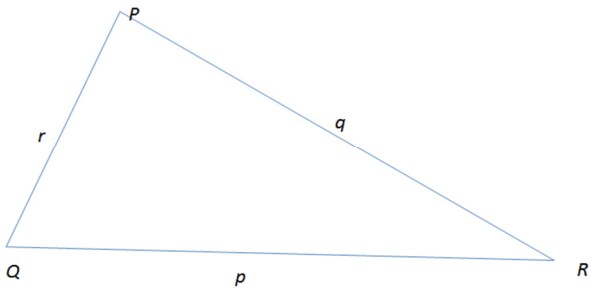
$$r^2 = p^2 + q^2 - 2pq\ \boldsymbol{cos}\ R$$

or in ELI short function form where right argument `x` represents angle *R*:

```
   {r:(p;q)<-y;((p*.2)+(q*.2)-2*p*q*2@x)*.0.5}
r
      (10;9) r @0.25
7.329446049
      (3;4) r @0.5
5
```

We see that for *R* being the right angle (the second case above *x*`=@0.5`) we have $r^2 = p^2 + q^2$. Hence, our old Pythagorean Theorem is a special case of the Law of Cosines (J. Durbin [3] §35).

Conversely, if we know the values of *p*, *q* and *r*, then we can find the angle R:



$$\boldsymbol{cos}\ R = (p^2 + q^2 - r^2)/2pq \quad \text{or} \quad R = \boldsymbol{arccos}\ (p^2 + q^2 - r^2)/2pq$$

where ***arccos*** *x* is the inverse function of ***cos*** *x*, and in ELI it is represented by the primitive *circle function* `_2@x`. Hence, we have function to calculate this angle R where right argument `x` now represents length *r*:

```
      {R:(p;q)<-y;_2@((p*.2)+(q*.2)-x*.2)%2*p*q}
R
      (5;5) R 3
0.609385308
```

We note that ***arcsin*** *x* is the inverse function of ***sin*** *x* and in ELI is represented by the circle function `_1@x`. Similarly, `_3@x` is the inverse function of `3@x`, the ***tan*** *x* function.

Finally, the amazing thing is that there is a relationship between π and the mathematical constant ***e*** vie complex numbers. This is the famous *Euler formula*:

$$\boldsymbol{e}^{ix} = \boldsymbol{cos}\ x + i\ \boldsymbol{sin}\ x$$

In particular, when *x*= π, ***cos*** *x* = `-1` and ***sin*** *x* = `0` so we have $\boldsymbol{e}^{i\pi} = -1$ or

```
     *.0j1*@1
_1
```

It certainly is too advanced for us to give a proof of this Euler formula here, but we can test this formula by writing a function for the right side and test it on a sample data points.

```
      {c_s:(2@x)+0j1*1@x}
c_s
      x
1 2 3j4 2.8 _10
      *.0j1*x
0.5403023059j0.8414709848 _0.4161468365j0.9092974268 _0.01813234507j0.002584703108
      _0.9422223407j0.3349881502 _0.8390715291j0.5440211109
      c_s x
0.5403023059j0.8414709848 _0.4161468365j0.9092974268 _0.01813234507j0.002584703108
      _0.9422223407j0.3349881502 _0.8390715291j0.5440211109
```

4. Coding with Arrays, Lists and Dictionaries

4.1 Accessing and changing array and list elements

We have already introduced *array* and *list* as the basic data structures in ELI in § 1.2, but we haven't touched on the topic of how to *access* elements in an array or a list, and how to *change* some elements in an array or a list. This is done by ***indexing*** and ***indexed assignment***, both depend on `[]IO`, the system variable *index origin* we introduced in § 1.4. Let us assume for the moment that we have `[]IO=1` and v is a vector of length of n, then $v[i]$ is the i-th element of v provided that $1 \le i \le n$; in case we have `[]IO=0` then $v[i]$ is the $(i+1)$-th element of v provided that $0 \le i \le n-1$. We note that `[]IO=1` is the default in ELI while in the programming language C it is like one always has `[]IO=0`. For example,

```
      w10
14 76 46 54 22 5 68 94 39 52
      w10[2]
76
      w10[5]
22
      w10[0]
index error
      w10[0]
         ^
      []IO<-0
      w10[2]
46
      w10[5]
5
      w10[10]
index error
      w10[10]
          ^
```

We see that when the index i is out of bound the system responds with an `index error`. We note that we can replace the single index i by a vector (even an array) $I = i_1, \ldots, i_m$ as long as each element in I satisfies the bound stated above; $v[I]$ then is a vector vI of length m whose k-th element of is $v[i_k]$. For example (still `[]IO=0`),

```
      w10[0 2 5 8]
14 46 5 39
      w10[9 7 7]
52 94 94
      w10[1+!5]
76 46 54 22 5
```

We see that I can have repeat elements and I can also be an integer expression as long as the resulting elements are all within the indexing bounds.

To change the values of selected elements in a vector v, we do an ***indexed assignment***:

$$v[I] \leftarrow w, \quad I = i_1, \ldots, i_m$$

where I is either a single index or a vector of indices (for simplicity we assume they are all distinct) as before all satisfy the bounds and w is either a singleton or a vector of length m; in case $m>1$ and w is a singleton then all elements $v[i_k]$ in v are replaced by w, otherwise each $v[i_k]$ is replaced by the corresponding element in w. For example (continue with `[]IO=0`),

```
      w10
14 76 46 54 22 5 68 94 39 52
      w10[I]<-w10[I<-0 2 5 8]*10
      w10
140 76 460 54 22 50 68 94 390 52
      w10[!3]<-0
      w10
0 0 0 54 22 50 68 94 390 52
      song<-'from sea to shining sea.'
      song
from sea to shining sea.
      song[5 12 20]<-'S'
      song
from Sea to Shining Sea.
```

Of course, in case any of these indices is out of bound we would get an `index error`.

Since a ***list*** is also a linear structure like a *vector*, it can be similarly indexed to get its elements and indexed assigned to change some of its elements as illustrated above for vectors. We can represent the set S_5 in **§ 1.3** in a *list* form as follows, and try to access some of its elements as well as to change some of them:

```
      S5<-(`Jobs;2011;('Apple Company';'Pixar'))
      S5
<`Jobs
<2011
<<Apple Company
 <Pixar
      S5[0 1]
<`Jobs
<2011
      S5[2]
<Apple Company
<Pixar
      S5[1]
2011
      S5[1]<-2011.10.05
      S5
<`Jobs
<2011.10.05
<<Apple Company
 <Pixar
```

We note that `S5[1]` is no longer a list but a scalar of integer type and it can be replaced by a scalar of type date because `S5` is a list which can be *non-homogeneous* while

```
      song[5 12 20]<-`S
domain error
      song[5 12 20]<-`S
                    ^
```

runs into problem because `song` is a vector of *characters*, i.e. a *string*, but `` `S `` is of *symbol* type.

For a general multi-dimensional array *a*, the indices to *a* are multi-dimensional. For simplicity, we assume that *a* is a *m* by *n* matrix, i.e. a two dimensional array; its indexing then is of the form

$$a[I;J], \qquad I= i_1, \ldots, i_m\,,\ J= j_1, \ldots, j_n$$

where indices in *I* select the *rows* of matrix *a* to be taken and those in *J* select the *columns* of *a* to be taken. Either *I* or *J*, or both *I* and *J* can be singletons. Moreover, either *I* or *J* in [*I*;*J*] can be missing, i.e. not present, in the indexing expression [*I*;*J*]; in this case, it means that all rows of *a* are chosen (when *I* is missing) or all columns of *a* are chosen (when *J* is missing). We illustrate these rules by following example of a character matrix (`[]IO=1`):

```
      alph<-3 8#'abcdefghijklmnopqrstuvwx'
      alph
abcdefgh
ijklmnop
qrstuvwx
      alph[3;4]
t
      alph[1 3;2+!4]
cdef
stuv
      alph[;2+!4]
cdef
klmn
stuv
      alph[1 3;]
abcdefgh
qrstuvwx
```

Indexed assignment to a matrix *a* follows the same general rule as that for a vector *v*. For the indexed assignment

$$a[I;J] \leftarrow w, \qquad I = i_1, \ldots, i_h \,,\; J = j_1, \ldots, j_k$$

to be valid both elements in *I* and *J* must be within bounds of the dimensions of *a*, and *w* is either a singleton or it has the same *shape* as that of *a*[*I*;*J*] (the *Cartesian product* of the index set *I* and the index set *J* with `!m` substituting for missing *I* and `!n` substituting for missing *J*). Let us continue our play with the array `alph`:

```
      alph[3;4]<-'y'
      alph
abcdefgh
ijklmnop
qrsyuvwx
      alph[1 3;2+!4]<-2 4#'CDEFSTUV'
      alph
abCDEFgh
ijklmnop
qrSTUVwx
      alph[;2+!4]<-3 4#'abcdefghijkl'
      alph
ababcdgh
ijefghop
qrijklwx
      alph[1 3;]<-'*'
      alph
********
ijefghop
********
```

Multi-dimensional indexing does not apply to lists since lists are linear.

4.2 Operations on arrays

We have already seen many primitive functions in ELI which take values of elements in one or two arrays to produce another array of the same shape; these are *scalar functions* and they carry out mathematical operations on the array elements. There are also many other primitive functions in ELI which take array(s) as argument(s) and transform them into new arrays possibly with new *shapes*. They are part of *mixed functions*. They either *rearrange* array elements or *take/drop* piece of an array. We start with the dyadic ***take*** *function* `l^.a`, where the left argument `l` is an integer vector of length `r` which equals to the rank of `a`. For the simple case of a vector `a`, `l` is just one integer which can be positive, negative or `0`. For example,

```
      w10<-14 76 46 54 22 5 68 94 39 52
      1^.w10
14
      #1^.w10
1
      w10[1]
14
      #w10[1]

      3^.w10
14 76 46
      _3^.w10
94 39 52
      12^.w10
14 76 46 54 22 5 68 94 39 52 0 0
      _12^.w10
0 0 14 76 46 54 22 5 68 94 39 52
      ch8<-'abcdefgh'
      5^.ch8
abcde
      _5^.ch8
defgh
      _9^.ch8
 abcdefgh
      12#w10
14 76 46 54 22 5 68 94 39 52 14 76
```

We see that `l^.a` takes the first `l` elements of `a` if `l>0`, and `l^.a` takes the last `l` elements of `a` if `l<0`, i.e. takes elements from `a` backwards. We also see that in the case of *overtake*, i.e. `(|l)>#a` resulting in more elements to be taken than there are, the additional elements are filled by the *typical element* of the type of `a` (the typical element of numeric type is `0` while the typical element of character type is a blank `' '`). We also notice the difference between reshape where when additional elements are required it reuses elements from `a` itself in ravel order but in overtake where additional elements are the typical element of the type of `a`. `0^.a` results in an empty vector. Take can be regarded as a whole piece indexing either from the beginning of an array or from the end of an array.

The dyadic primitive function ***drop*** `l!.a` is the opposite of *take* in the sense that what `l^.a` takes from `a` the `l!.a` drops from `a`. For example,

```
      1!.w10
76 46 54 22 5 68 94 39 52
      3!.w10
54 22 5 68 94 39 52
      _3!.w10
14 76 46 54 22 5 68
      5!.ch8
```

```
fgh
      _5!.ch8
abc
```

Again, *negative* drop means drop from the back; and clearly all *over drops* result in empty vectors.

All these apply to *lists* as well. For example,

```
      ll<-(3;33 4;0)
      2^.ll
<3
<33 4
      2!.ll
<0
```

We would not getting into the multi-dimensional case of *take* and *drop* here. Interested readers can consult **§** 1.8 in the ELI Primer [1].

The monadic function ***reverse*** `$a` transforms the array `a` to a reverse position. We illustrate this by the cases where `a` is a vector:

```
      $w10
52 39 94 68 5 22 54 46 76 14
      $ch8
hgfedcba
      ss<-`John `Jack `Joshua
      $ss
`Joshua `Jack `John
      $2015.04.01+!7
2015.04.08 2015.04.07 2015.04.06 2015.04.05 2015.04.04 2015.04.03 2015.04.02
```

The dyadic function ***rotate*** `l$a` rotate the array `a` to the *right* (`l>0`) or the *left* (`l<0`) by the amount `|l`. We again illustrate these by the cases where `a` is a vector:

```
      w10
14 76 46 54 22 5 68 94 39 52
      2$w10
46 54 22 5 68 94 39 52 14 76
      _2$w10
39 52 14 76 46 54 22 5 68 94
      5$ch8
fghabcde
      _5$ch8
defghabc
      2$ss
`Joshua `John `Jack
      _2$ss
`Jack `Joshua `John
      3$2015.04.01+!7
2015.04.05 2015.04.06 2015.04.07 2015.04.08 2015.04.02 2015.04.03 2015.04.04
      _3$2015.04.01+!7
2015.04.06 2015.04.07 2015.04.08 2015.04.02 2015.04.03 2015.04.04 2015.04.05
```

And we leave the multi-dimensional case for readers to look up in the Primer [1] where the concept of the ***axis*** of a *reverse* or a *rotate* gets into play with `$` indicating the axis is the last axis of an array while `$.` indicating the axis is the first axis of an array. We note here that the reverse function works on lists while the rotate does not:

```
      ll
<3
<33 4
<0
      $ll
<0
<33 4
<3
```

Other than the dyadic *reshape* function `#`, perhaps the most important dyadic primitive function in forming a new *array* (or *list*) in ELI is the ***catenate*** function *a,b* where *a* and *b* can be both scalars, or one scalar one *array*, or both *arrays* (with certain constrains on their shapes) or both lists. We note that the *monadic* counterpart of *catenate*, *,b*, is the ***ravel*** function we encountered earlier in §1.2. Let us first see some examples where *a* or *b* is either a scalar or a vector:

```
      100,200
100 200
      100,14 76 46 54 22 5 68 94 39 52
100 14 76 46 54 22 5 68 94 39 52
      14 76 46 54 22 5 68 94 39 52,200
14 76 46 54 22 5 68 94 39 52 200
      (2015.04.02+!6),2015.04.20+!2
2015.04.03 2015.04.04 2015.04.05 2015.04.06 2015.04.07 2015.04.08 2015.04.21 2015.04.22
      '*',ch8,'!!!'
*abcdefgh!!!
      `Adam `Bob,ss
`Adam `Bob `John `Jack `Joshua
      ll
<3
<33 4
<0
      ll,('USA';(`president;`Washington `Lincon))
<3
<33 4
<0
<USA
<<`president
 <`Washington `Lincon
      ll,10 20
<3
<33 4
<0
<10 20
```

We see that *a,b* simply *glue* two pieces of data together whether *a* and *b* are scalars, one scalar one vector, two vectors, one list one vector or two lists. In the last case, when *a* is a list while *b* is a vector, *a,b* just turns *b* to be the last item in the new list.

If one of the argument to the *catenate* function is an array while the other is a scalar, *a,b* always works. For example,

```
      m
1  2  3  4
5  6  7  8
9 10 11 12
      m,0
1  2  3  4 0
5  6  7  8 0
9 10 11 12 0
```

```
      0,m
0 1  2  3  4
0 5  6  7  8
0 9 10 11 12
      chm<-2 4#ch8
      chm
abcd
efgh
      chm,'*'
abcd*
efgh*
      chm,.'*'
abcd
efgh
****
      '+',.chm,.'*'
++++
abcd
efgh
****
```

We can clearly see the rule. We note that *a,.b* is a function companion to *a,b* called ***catenate along the first axis*** while *a,b* always *catenate along the last axis*. *a,b* (and *a,.b*) would glue two matrices a and b if they have the same heights (resp. the same width). For example,

```
      a
abcd
efgh
      b
123
456
      a,b
abcd123
efgh456
      c<-3 4#'opikvbnmxyz '
      c
opik
vbnm
xyz
      a,.c
abcd
efgh
opik
vbnm
xyz
```

If the widths (resp. heights) of two operands to *catenate* (resp. *catenate along first axis*) are not equal it would result in a *length error*:

```
     a,c
length error
      a,c
       ^
```

We refer more details of the rules of the catenate functions to § 1.8 in W. Ching [1]. We note that both array/scalar operands to the *catenate* functions must be of the same type, be they numeric, character or symbolic.

The two basic operations in set theory are that of *union* and *intersection* of two sets (§ 1.3). If we restrict our discussion of sets to be sets of a *particular data type* and use vectors (of that data type) to represent sets, then `a,b` can be the basis of implementing *set union* in ELI. The problem is that there can be elements belong to both sets so

`a,b` would contain duplicates but for a vector to represent a set each element can only appear once. To this end, we introduce the monadic primitive function ***unique*** `=` in ELI: for any array `a`, `=a` returns a vector consisting of all unique elements in `a`. For example,

```
      m
3 4 5
5 6 7
      =m
3 4 5 6 7
      =3 4 5 5 6 7
3 4 5 6 7
```

With this primitive function, we can write the following function for set union:

```
      {union: =x,y}
union
      `John `Fred `Peter union `Fred `Peter `Bob `Jack
`Fred `Peter `Bob `Jack `John
      'ABC' union 'abc'
abcABC
      3 6 9 union !7
1 2 3 4 5 6 7 9
      'ac' union `a `c
domain error
union[1]  z<-=x,y
               ^
```

The last expression runs into problem because the left operand in `x,y` is of symbol type while the right operand is of character type.

4.3 Set membership and linear locations of elements

We have shown that we can use a vector (of certain data type) to represent a set of elements of a particular data type. In set theory, the most important operation is to query whether an element *a* is a *member* of a set *S*, i.e. check whether $a \in S$ is true. Indeed, ELI has a corresponding dyadic primitive function ***belong to***, `a?s`, where `a` and `s` can be any scalar or array, for checking membership of `a` in the set consisting of elements in `s`. For example,

```
      2?'abc'
0
      2?!3
1
      2 8?!3
1 0
      cm<-3 4#'awy'
      cm
awya
wyaw
yawy
      cm?'abc'
1 0 0 1
0 0 1 0
0 1 0 0
      'abc'?cm
1 0 0
```

First, we observe that the result of the *member of* function `?` is Boolean, i.e. `0` or `1` representing ***false*** or ***true***. Second, if the two operands are of *different* data types then the result is *false*. Third, the *shape* of the result (such as `#cm?'abc'`) is the same as the shape of the left operand (`#cm`). For a left array operand *a*, the result is a *scalar extension* of the result of each element in *a* with respect to the right operand.

We have one more set operation to implement: that the intersection of two sets $A \cap B$; by definition (§ 1.3) this is the set consisting of all elements which belong to both *A* and *B*. For simplicity, we use two vectors to represent the two sets of concern (we can always apply *ravel* to an array to get a vector) so we can use the *compress* function. We have the following function:

```
      {intersect:(x?y)/x}
intersect
       'aby' intersect =,cm
ay
      (3*!3) intersect !10
3 6 9
      `sim `joe `mark intersect `ada `jane `joe `kim
`joe
      0j1 10 3.5 0.02 100 intersect 2 3.5 0j1 10
3.5 0j1 10
```

Note that the reason we need to apply `=,` to `cm` is to make the right side of the expression to represent a set where not only it needs to be a vector but also each element there can appear only once.

Suppose we have a group of passwords `psw` of unequal lengths and we would like to count the number of digits contained in each password. We'll do the following with a sample `psw`:

```
      psw<-('king88';'Yoo2x';'Le3sa56';'s967750')
      {cnt_dgs:+/x?'0123456789'}
cnt_dgs
      cnt_dgs" psw
<2
<1
<3
<6
      ,.cnt_dgs" psw
2 1 3 6
```

We see that the function `cnt_dgs` first asks which element in `x` is a digit and then adds up the `1`s using *plus reduction* to get a count. Here the result of the *member of* function is used immediately as the input to the plus reduction. Prof. Alan Perlis of Yale, the first recipient of the Turing Award coined the word ***dataflow programming*** to describe this style of programming. Hence, Ken Iverson in designing APL used single character of a special font to represent a high level primitive is not a mere imitation of mathematical notation for aesthetic beauty or an excessive desire for succinctness of code, it is to facilitate a *dataflow* style programming. Therefore, ELI, following the tradition of APL, encourages this *dataflow* style of programming for programming productivity as well as for clarity of code. The each operator `"` applies the function `cnt_dgs` to each element of the list `psw`; and `,.` is the monadic ***raze*** function which turns a list of elements of homogeneous type into a vector (see § 2.1 of [1]). We can further apply the `avg1` function of § 2.4 to the last expression above to get the average number of digits appearing in a password in `psw`:

```
      avg1 ,.cnt_dgs" psw
3
```

This is a further extension to the previous flow of data stitched together with a group of ELI operations to achieve a non-trivial computation.

There is a dyadic primitive function ***index of***, `v!a`, in ELI which bears certain similarity to the dyadic *member of* function `a?v` but with the roles of the left and right operands reversed; for a scalar `a`, instead of asking whether `a∈ v`, it asks *where* `a` is *located* inside vector `v`. The answer depends on `[]IO`. First if `a∈ v` is false, than `v!a` equals to

```
(#v)+1-[]IO=0
```

i.e. it equals to the length of `v` or `#v` plus `1` depends on whether `[]IO` is `0` or `1`, in other words, the *index* of the last element in `v` plus `1`. On the other hand, if `a∈ v` then `v!a` equals to the *index of* the first time `a` appears in `v`. For an array `a`, the result of `v!a` is a *scalar extension* on the right operand `a`, i.e. the *shape* of `v!a` is the *shape* of `a` and each element of $(v!a)_{ij}$ equals to $v!a_{ij}$. For example,

```
     `sam `fred `kate !`john
4
      []IO<-0
      `sam `fred `kate !`john
3
      'abcdecfgh'!'acw'
0 2 9
       []IO<-1
       'abcdecfgh'!'acw'
1 3 10
       'abcdecfgh'!2 4#'acwxfjh'
1  3 10 10
7 10  9  1
      v<-3.2 3.05 _4 1 12
      v!1 10 3.2
4 6 1
      m<-3 3#8 3.2 11 12 _4
      m
 8    3.2 11
12   _4    8
 3.2 11   12
      v!m
6 1 6
5 3 6
1 6 5
```

The *monadic* primitive function `?b` corresponding to the dyadic *member of* function `a?b` is called ***where***; for a Boolean vector `b`, `?b` gives the indices of elements in `b` which are equal to `1`. Hence, `?b` depends on `[]IO`. This function is quite useful for replacing a group of elements in a vector by a single new element or number of `(+/?b)` elements. For example,

```
      v<2
0 0 1 1 0
      ?v<2
3 4
      v[?v<2]<-0
      v
3.2 3.05 0 0 12
      (cv<-'abcdecfgh')?'cef'
0 0 1 0 1 1 1 0 0
      ?0 0 1 0 1 1 1 0 0
3 5 6 7
      cv[?(cv<-'abcdecfgh')?'cef']<-'x'
      cv
abxdxxxgh
      v[?v<2]<-100 200
      v
```

```
3.2 3.05 100 200 12
```

4.4 Outer product and inner product

There are two more *operators* in ELI, the ***outer product*** and the ***inner product***, which we shall introduce here. First, the ***outer product*** `.:` is a *monadic operator* (like all the operators we introduced so far but takes the argument on the right): given a dyadic primitive scalar function *f* it produces a *dyadic derived function* `.:`***f*** which operates on two vector arguments `A` and `B` as follows with (`A.:`*f*`B`) being a matrix:

$$(\texttt{A.:}f\texttt{ B})\texttt{[i;j]} \longleftrightarrow \texttt{A[i]}\,f\,\texttt{B[j]}$$

It is easy to see how this works by some simple examples:

```
      (!10) .:* !10
 1  2  3  4  5  6  7  8  9  10
 2  4  6  8 10 12 14 16 18  20
 3  6  9 12 15 18 21 24 27  30
 4  8 12 16 20 24 28 32 36  40
 5 10 15 20 25 30 35 40 45  50
 6 12 18 24 30 36 42 48 54  60
 7 14 21 28 35 42 49 56 63  70
 8 16 24 32 40 48 56 64 72  80
 9 18 27 36 45 54 63 72 81  90
10 20 30 40 50 60 70 80 90 100
      100 1000 10000 .:% 2 4 5 10
  50   25   20   10
 500  250  200  100
5000 2500 2000 1000
      2.3 1.7 4.5 9 .:~. 3 2 1.1 0.7 10
3   2.3 2.3 2.3 10
3   2   1.7 1.7 10
4.5 4.5 4.5 4.5 10
9   9   9   9   10
      2.3 1.7 4.5 9 .:>= 3 2 1.1 0.7 10
0 1 1 1 0
0 0 1 1 0
1 1 1 1 0
1 1 1 1 0
```

The first example is just our old multiplication table. The second example is a table showing when 3 piles of money divide by 4 groups of people in all combinations how much will be each person's share. The third example is the pairwise maximum of two numeric vectors, and the fourth example is the pairwise comparison (`>=`) of the same pair of numeric vectors. The arguments `A` and `B` to the derived function can be scalar or arrays and one can find the general rule in **§1.6** of [1] where more interesting examples of outer products can also be found.

The ***inner product*** *operator* `:` is a *dyadic* operator, it takes two dyadic primitive scalar functions ***f*** and ***g*** to produce a dyadic derived function ***f*****:*****g*** (*note* that in APL it is denoted by *f*.*g*); for vectors `V` and `W`, `V`*f*:*g*`W` is defined to be `f/V`*g*`W`. For example, let `unitpc` be the unit price of 5 merchandises and `qty` be a list of number of items bought for each merchandise, then the total payment is given by:

```
      unitpc<-2.5 11.2 7 13.05 1.95
      qty<-4 2 3 1 9
      unitpc +:* qty
84
```

`V` and `W` can be arrays, in that case, the last dimension of `V` must be equal to the first dimension of `W`. We refer to **§ 1.8** of [1] for details and additional examples of inner products.

4.5 Sort functions

Sorting is likely the most used operation in data processing. ELI provides two primitive functions, ***grade up*** `<v` and ***grade down*** `>v` for *sorting* numeric and character vectors. For a numeric vector `v`, `<v` produces a vector `u` of indices such that each `u[i]` is the *index* of `v[i]` in the new vector after we sort `v` into a vector of *non-decreasing* order; in other words, `v[<v]` is an *upward* sorted vector. Similarly, for the *grade down* function `>`, `v[>v]` results in a sorted vector of *non-increasing* order. For a character vector, the sorting order is the *lexicographic* order. For example,

```
      v
3.2 3.05 100 200 12
      <v
2 1 5 3 4
      v[<v]
3.05 3.2 12 100 200
      cc<-'pohycjkmacl'
      <cc
9 5 10 3 6 7 11 8 2 1 4
      cc[<cc]
acchjklmopy
```

Suppose `w` is a list of contributions (a pair of amount and name) to a club. We would like to produce a list of names in the order of amounts they contributed starting with the one with the most amount. We proceed as follows:

```
      w<-((3.2;`sam);(3.05;`jack);(100;`mary);(200;`joe);(12;`peter))
      v1<-^."w  //take the first item of each element in w
      v1
<3.2
<3.05
<100
<200
<12
<200
      i<-<>,.v1  //turn v1 into a vector and apply grade down function
      i
4 3 5 1 2
      w[i]
<<200
 <`joe
<<100
 <`mary
<<12
 <`peter
<<3.2
 <`sam
<<3.05
 <`jack
```

We are almost there; we just need to extract symbol names from the sorted list. To this end we code a short function to extract the last element in a list:

```
      {last:x[_1+[]IO+#x]}
last
      last (200;`joe)
```

```
`joe
      last"w[i]
<`joe
<`mary
<`peter
<`sam
<`jack
      ,.last"w[i]
`joe `mary `peter `sam `jack
      ,.last"w[>,.^."w]
`joe `mary `peter `sam `jack
```

Here we see the wisdom of providing `>v` as a set of indices instead of just the sorted vector because the indices can be further used to *rearrange* related items. The last expression above is a further example of ELI's *dataflow style programming* or what used to be called *one-liners* in APL which horrified people not well-versed in APL. Once one understands each of the operations in that line as we did going thru in its derivation, its clarity stands out, and compare this line with one or two pages of code in a verbose language it is far easy to maintain due to its succinctness and precision.

4.6 Dictionaries

In ELI, there is a special kind of *two items list* `(d;r)` called ***dictionary***: a *dictionary* `D` consists of a ***domain*** `d` which is a vector or a list, and a ***range*** `r` which is also a vector or a list of equal length as that of `d`, and a *one-to-one correspondence* set up between the two by the dyadic function ***map*** **:**, `d:r`. Once a dictionary `D` is set up, a pair of *system functions* will return its components: `key(D)` gives `d`, the domain, and `value(D)` gives `r` in list form (even if `r` is originally a vector), the range of `D`. The *domain* `d` must be a *simple* list of unique elements such as a vector of symbols, characters or numbers 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 yield the corresponding item in `r`, which can be a scalar, an array or a list in `r` as the result. For example:

```
      D1<-_1 0 1 2 3:`loeb `greg `carl `kevin `paul
      D1
_1| loeb
0 | greg
1 | carl
2 | kevin
3 | paul
      D1[_1]
`loeb
      D1[0 2]
<`greg
<`kevin
      D2<-'ABCDEF':(2 3;1.2 5;4.1 0;100 92;8 7.5;32 12)
      D2
A| 2 3
B| 1.2 5
C| 4.1 0
D| 100 92
E| 8 7.5
F| 32 12
      key(D1)
_1 0 1 2 3
      value(D1)
<`loeb
<`greg
<`carl
<`kevin
```

```
<`paul
      key(D2)
ABCDEF
      value(D2)
<2 3
<1.2 5
<4.1 0
<100 92
<8 7.5
<32 12
      D3<-`apple `orange `berry:(('R';0.5);('Y';0.45);3.1 4.2 2.5)
      D3
apple | -
orange| -
berry | 3.1 4.2 2.5
      value(D3)
<<R
 <0.5
<<Y
 <0.45
<3.1 4.2 2.5
```

If we regard a vector or a list of a length `n` as a *map* from `!n` to its elements, then a *dictionary* `D` becomes a natural extension to vector or list by replacing `!n` with `key(D)` as the underlying set of indices. *Dictionaries* are called ***hashes*** in the programming language **Perl**, and called ***maps*** in the popular language **Python**. Now let us recast the variable `w` of contributions in the previous section as a dictionary which maps individual names to the amount, and rework thru the excise to get the final result (*range* should be a list, but in this case it acts as a list):

```
      W<-`sam `jack `mary `joe `peter:3.2 3.05 100 200 12
      W
sam  | 3.2
jack | 3.05
mary | 100
joe  | 200
peter| 12
      value(W)
<3.2
<3.05
<100
<200
<12
      >,.value(W)
4 3 5 1 2
      key(W)
`sam `jack `mary `joe `peter
      (key(W))[>,.value(W)]
`joe `mary `peter `sam `jack
```

We see that by using dictionary the coding is much cleaner compared with that in the previous section. One just need to remember that `value(D)` always returns a list, so we need the *raze* function `,.` to turn it into a vector. Finally, we note that IBM APL2 (successor to the classical APL) has *general arrays* in lieu of *lists* but has no *dictionaries*.

5. Defined Functions, Script Files and Standard Library

5.1 Defined functions and control structures

We have briefly introduced *defined functions* in **§ 1.5** and learned how to write short functions, a special form of defined functions. We describe now how to write a defined function in general. One starts a defined function by typing the *two-character symbol* `@.` and ends the definition of a function with a matching `@.`. The first line of a function definition, i.e. the line starting with `@.`, is called the ***function head*** and it is of the form

`@.`[*res*`<-`]`[`*larg*`]` *fnam* `[`*rarg*`][;`*local_var_list*`]`

The notation [*a*] above means the item *a* is *optional*. So *res*`<-` is present in the function head only if the function named *fnam* returns a result named *res*, and *larg* is the variable name of the *left parameter* if there is one while *rarg* is the name of the *right parameter* if there is one. A function needs not return a *result* and it needs not take any argument (i.e. a *niladic* function); but if it has a left argument then it must have a right argument too. *local_var_list* is a list of local variables separated by `;`. We'll explain the meaning of local variables later. Let us try a simple example: a function which picks one element out of a vector (the right argument) according to the left argument:

```
      @.z<-la pick1 ra
        la<-la_.#ra
        z<-ra[la]
      @.
      7 pick1 4 5 10 11 15
15
      3 pick1 4 5 10 11 15
10
```

Once you type in the function head, ELI interpreter enters the ***edit mode***. You then enter the function body line by line; when you finish you enter a matching `@.` and the system jumps back to the original mode, i.e. the ***interpret mode*** signaling the end of a function definition. Once a function is properly defined, we can call the function by enter the function name, and supplying its argument(s) if it is not *niladic*. ELI would then take the values of the actual parameters, i.e. the ones we supplied in a call, to the formal parameters, i.e. the *larg* and *rarg* in the function definition and executing the code in the function body line by line until it reaches the end. If the function returns a result, i.e. *res*`<-` is present at the beginning of *function head* line, then somewhere in the function body, there must be an assignment to the result variable *res*; otherwise it would end up in a `value error`. Also, `->` is the branch symbol ELI inherits from APL which we'll not use much except that `->0` signals an *immediate exit* in a function execution and `->msg` will *abort execution* with an error message `msg`. If `k` is one of the local variables listed in the function head, any use of `k` will look for its value assigned in the function, otherwise a `value error` message will appear. Local variables as well as parameters of a defined function will disappear when defined function finishes its execution. See **§** 3.1 of [1] for rules on accessing non-local variables in a function execution.

All examples of ELI code so far are *one liners*, i.e. an ELI expression stitched together with primitive or defined functions, and their execution is strictly from *right to left*. We call these ***simple statements*** in ELI. A ***general statement*** in ELI is built out simple statements with ***control structures*** which alternate the execution order of the statements involved from a straight line by line execution. We *emphasize* here that general statement with control structures cannot be entered directly in ELI interpreter mode; they can only appear inside a defined function. We first introduce the *if-statement* which is of the form:

`if` `(`boolean-expression`)` `statement` `[`**`else`** `statement]`

where `statement` is one statement or several statements in multiple lines grouped together by a pair `{…}`of curry

brackets. The **else** part is optional (depends on what the code intends to do). We illustrate the use of if-statement with the following function which lists days in months with the year as a right parameter since in a *leap year* the month of February has 29 days instead of 28 and this is determined by whether the year can be evenly divided by 4:

```
[0]  z<-days_in_month y       //lines after @.
[1]  z<-`jan`feb`mar`apr`may`jun`jul`aug`sep`oct`nov`dec:12#_
[2]  z[`jan`mar`may`jul`aug`oct`dec]<-!31
[3]  z[`apr`jun`sep`nov]<-!30
[4]  if (0=4|y) z[`feb]<-!29
[5]  else z[`feb]<-!28
      @.z<-days_in_month y  //return from edit mode after @. In line [6] above
      )fns
days_in_month
     days_in_month 2015
jan| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
feb| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28
mar| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
apr| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
may| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
jun| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
jul| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
aug| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
sep| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
oct| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
nov| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
dec| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
     days_in_month 2016
jan| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
feb| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29
mar| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
apr| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
may| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
jun| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
jul| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
aug| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
sep| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
oct| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
nov| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30
dec| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31
```

We note that the first line in the function is setting up a dictionary whose range is a list of length 12 with empty elements. Actually, we could avoid the use of the if-statement in the function above by replacing it with

```
z[`feb]<-!28+(0=4|y)
```

The *while-statement* is of the form

```
while (boolean-expression) statement
```

where `statement` again may be one statement or several statements grouped together by a pair of `{}`. That statement is repeatedly executed as long as the `boolean-expression` inside `()` remains to be `1`, whose value presumably depends on variables in the `statement` as well as itself. We have the following example which prints out a bunch of numbers:

```
     @.z<-print;n;x  //go to edit mode to enter the function
      )fns           //after returning from edit mode
print
     []CR 'print'    //display the function definition
z<-print;n
```

```
z<-!n<-0
while (0.00000001<x<-%1+n<-(n+1)*.2)
z<-z,x
    print
0.5 0.2 0.03846153846 0.001477104874 2.18183405E_6
```

We note that `[]CR` is the ELI *system function* which display the *canonical representation* of a function. We remark that one must make sure that the `boolean-expression` in a while-statement will eventually become `0` unless one intentionally wants to have an *infinite loop*. Usually, there is a change of value either in the body of the while-statement or as in our case the change is in the `boolean-expression` itself (when `n` increases, `x` is getting smaller and smaller).

The *for-statement* is of the following form:

```
for (idxv:for-format) statement
```

where `for-format` is either a vector for `idxv` to iterate over in executing the `statement` or an indication of steps for `idxv` to go thru (we shall illustrate this second case of *for-statement* in the next section and we refer the *case-statement* to § 3.4 in [1]). Let us look at one example of the *for-loop*:

```
      @.z<-n draw gp;i
      )fns
draw
      []CR 'draw'
z<-n draw gp;i
z<-gp:(#gp)#_
for (i:gp) z[i]<-?.n
       100 draw `emma `john `jane `jack `karl
emma| 14
john| 76
jane| 46
jack| 54
karl| 22
      100 draw `emma `john `jane `jack `karl
emma| 5
john| 68
jane| 68
jack| 94
karl| 39
```

This function body's first line set up a dictionary with its domain equal to the right argument `gp` which is a group of people represented as a vector of symbol type and its range is a list of empty slots. The *for-statement* followed has `i` as `idxv` to iterate over `gp`. What it does is to randomly draw a number from `!n` to assign to corresponding slot `z[i]`. We notice that the second function call with the same arguments didn't produce identical results because each call to the *roll* function `?.` changes `[]RL`. We also notice that in the second call `john` and `jane` draw the same number `68`. This is due to the fact that each call of `?.` is independent of each other and hence different draws could get the same number just as when repeatedly flip a coin can end up in two heads. If we change the function to first get five numbers by calling `(#gp)?.n` and then spread the result to the slots then no two would get the same result. This of course depends on `n>=#gp`.

5.2 Recursion

Recursion refers to the fact of a function calling itself in the body of its function definition. The following series is called ***Fibonacci sequence*** (discovered by the Italian mathematician Fibonacci in 1202) where the next number is the sum of the previous two numbers:

0 1 1 2 3 5 8 13 21 34 55 89 144 233 . . .

If we denote the n-th Fibonacci number in this sequence by F_n then with seeds F_0 =0 and F_1 =1 we have

$$F_n = F_{n-1} + F_{n-2}$$

We write a *recursive* function `fib n` to generate this this sequence up to F_n. as follows:

```
[0]  z<-fib n
[1]  if (n<1)->'n must be greater than 0'
[2]  if (n=1) z<-0 1
[3]  else z<-z,+/_2^.z<-fib n-1
      fib 0
n must be greater than 0
      fib 1
0 1
      fib 15
0 1 1 2 3 5 8 13 21 34 55 89 144 233 377 610
```

The German mathematician and astronomer J. Kepler observed that the *ratio* of consecutive Fibonacci numbers converges to a limit, the ***golden ratio*** φ (if a>0, b>0 and $(a+b)/a = a/b$ then a/b is the *golden ratio* which equals to `(1+5*.0.5)%2=1.61803399`):

$$\lim_{n\to\infty} F_{n+1} \% F_n = \varphi$$

Now we turn to a less well-known but more difficult problem: given a non-negative integer n find integers a and b such that

$$(3+\sqrt{5})^n = a + b\sqrt{5}$$

write a function which takes n as input and (a,b) as output. Example input/output:

```
0 → 1, 0
1 → 3, 1
2 → 14, 6
3 → 72, 32
4 → 376, 168
5 → 1968, 880
6 → 10304, 4608
7 → 53952, 24128
8 → 282496, 126336
9 → 1479168, 661504
```

We first observe that

$$(3+\sqrt{5})^{n+1} = (a + b\sqrt{5})*(3+\sqrt{5}) = 3a+5b+(5a+3b)\sqrt{5}$$

In other words, if the pair $z_n=(a_n,b_n)$ is the solution to input n, then $a_{n+1}=3a_n+5b_n$ and $b_{n+1}=a_n+3b_n$ is a solution to n+1. Let `v` be the 2 element vector z_n and `m` be the matrix

```
      m<-2 2#3 5 1 3
      m
3 5
1 3
```

then the vector `m+.*v` is a solution for n+1. So we have the following recursive function to solve the problem:

```
@.z<-Pair n;a;b
  if (n=0) z<-0 1 0
  else z<-n,(2 2#3 5 1 3)+:*(Pair n-1)[2 3]
@.
      Pair 3
3 72 32
```

where the first element of the output is the input while the next two elements are the solution for that input. In case we want to output solutions up to `n`, we can do the following:

```
      Pair" 1||!5
<1 3 1
<2 14 6
<3 72 32
<4 376 168
<5 1968 880
```

where `1||v` turns a vector `v` into a list so the each operator `"` can be applied to each item of `v` (see ***partition*** function `||` in § 2.2 of [1]). We remark here that it is necessary to turn the argument into a list first because the function `Pair` cannot apply to a vector. We actually can write a non-recursive function to print out solutions to the above problem up to *n*. The function first lay out an *n* by 3 matrix, fill the first column with input numbers and calculate values in the other two columns row by row using the *step-format* of the *for-statement*:

```
     []CR 'Pairs'
z<-Pairs n;a;b
  z<-((n+1),3)#[]IO<-0
  z[;0]<-!n+1
  z[0;]<-0 1 0
  for (i:1;n) z[i;1 2]<-(2 2#3 5 1 3)+:*z[i-1;1 2]
      Pairs 9
0       1      0
1       3      1
2      14      6
3      72     32
4     376    168
5    1968    880
6   10304   4608
7   53952  24128
8  282496 126336
9 1479168 661504
```

Non-recursive functions usually run faster than their recursive counterparts and more amenable to parallelization.

5.3 Script files and output variables

One can enter a bunch of functions and variables into ELI non-interactively by loading a prepared text file. Such files are usually called ***script files*** and these are just ordinary text files of type *.txt or of type *.esf. An ELI script file can contain ELI *expressions* (*simple statements*), *defined functions* and *variables* (i.e. their values). A script file named file1.esf (or file1.txt) *properly* located (we'll explain this later) can be loaded into ELI or copied into an active ELI workspace by system commands

```
    )fload file1
```

or

```
    )fcopy file1
```

The difference between the two commands is that the first command will clear out the existing workspace and then load the content of `file1` while the second command will load the content of `file1` into an existing workspace. So in the case of `fcopy`, what already exists in the workspace will remain to be there provided there is no *name conflict* (in that case, the item will be replaced by the item of the same name from `file1`). Let us create a text file print.txt in `C:\csource\` directory (Windows version):

```
//2015-6-12 printing loop
[]IO<-0
w<-10?.100

@.print x
  for (i:0;_1+#x) {
    []<-'i= '
       []<-i
       []<-'x[i]= '
       []<-x[i]
  }
@.
```

We then click on item Options in the top bar of our ELI window, one item in the *draw down menu* is Workspace Path; click on that we see C:\Program Files (x86)\eli\ws\. This is the *default location* for ELI script files. We can either move our file there or change the path to `C:\csource\`. Next, type

```
      )fload print
      []IO<-0
      w<-10?.100
saved 2015.06.12 23:00:16 (gmt-5)
      []IO
0
      )vars
w
      )fns
print
      []CR 'print'
print x
  for (i:0;_1+#x) {
    []<-'i= '
       []<-i
       []<-', x[i]= '
       []<-x[i]
  }
      print !5
i= 0, x[i]= 0
i= 1, x[i]= 1
i= 2, x[i]= 2
i= 3, x[i]= 3
i= 4, x[i]= 4
      print w
i= 0, x[i]= 86
i= 1, x[i]= 24
i= 2, x[i]= 53
i= 3, x[i]= 45
i= 4, x[i]= 74
i= 5, x[i]= 90
i= 6, x[i]= 30
i= 7, x[i]= 29
i= 8, x[i]= 6
```

```
i= 9, x[i]= 56
```

We see that after we `)fload` the file, the ELI interpreter executes the executable statements in the file, and all the *variable*(s) assigned and *function*(s) defined in the file have been brought into the current active workspace. In our case, `[]IO` becomes `0` and we have variable `w` and function `print`. We can use these variables and functions and even save the whole thing into a new workspace by using the system commands `)wsid` and `)save` (§ 1.5).

In ELI *interpreter mode*, we observe that if a line of code is ending in an *assignment* (`i<-...`) then nothing will print out; on the other hand, the result of the last expression (such as a single variable) will be printed out. If we want to print out something in a defined function, we use the *system variables* ***quad output*** `[]` and ***bare output*** `[)` by assigning the expression we want to print to one of them as we did in the `print` function. The difference between `[]` and `[)` is that for `[]` the next output will be in a *new line* while for `[)` the next output will continue immediately after the previous output as we see in the function `print`.

We have wrote a text file as a *.txt script file to be loaded into ELI interpreter. There is another way to create script files and that is to *export* one from an active workspace by the *system command*

```
)out file1
```

where `file1` is the name of the file outputted to the *default location* of the ELI system we stated earlier. All the values of user defined variables and definitions of user defined functions in the workspace will be there (but no executed statements except that of `[]IO<-0` if `[]IO` has been changed). For function definitions, they are of the same form as in our hand prepared file `print`; for a variable it is of the form

```
&vnam typ rnk shape
,vnam
&
```

where ***typ rnk shape*** are the *type* ('`I`' for integer, '`E`' for floating point number, '`C`' for character, '`S`' for symbol etc. see § 4.2 of [1] for more details), *rank* and *shape* of variable `vnam`.and `,vnam` are raveled values of `vnam`. For example, if we have a `v<-%2` in our workspace, then `v` and the `w` in our preceding example are in file of the following form:

```
&v E 0
0.5
&
&w I 1 10
87 25 54 46 75 91 31 30 7 57
&
```

We can also write a variable in this form in a script file to be imported into ELI without an assignment.

5.4 The standard library

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

```
      )clear
      )fcopy standard
      alphabet<-'ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz'
      alphanum<-alphabet,'0123456789'
saved 2014.06.14 00:28:14 (gmt-5)
```

```
      )fns
all_equal   avg        cat        cat1       cutlowerl     cutlowerr     dat2wd     dat2wdc  ddif
det         diag       dt2d       dt2dat     dt2hms        dt2s          dt2sec     dt2ymd    eye
find        findj      findv      gmean      index         indexj        intersect  last     less
lgth        lower      median     mn_dev     mv_max        mv_min        mv_sum     nw_avg nw_max
nw_min      nw_sum     rand       randm      replac1s      replacac      replace    replic sec2ts
sort        stddev     substr     sym        take          take1         trace      tril     trim
triml       trimr      triu       union      upper         xor
```

will copy in the script file, then a group of pre-defined functions and two variables are available. One can study the file to see the functionalities and restrictions of these *pre-defined functions* by reading the comments and inspecting their definitions. We refer examples of their use to **§** 4.4 in [1]; some of these such as `intersect` and `union` we have already seen in previous sections. We note here that to use this library, before copying `standard.esf` into an existing workspace, one must make sure that no existing functions in the original workspace have the same name as those in `standard.esf`. Otherwise, a name change is required to prevent a loss of user functions.

6. Data and Probability

6.1 studying numeric data

For a sequence of real numbers *v*, the ***mean*** of *v* is the sum of *v* divided by the number of elements in *v*:

```
      {avg: (+/x)%^x}
avg
       avg _1 7
3
      v<-7 8 9 2 10 9 9 9 9 4 5 6 1 5 6 7 8 6 1 10
      avg v
6.55
```

The *mean* (i.e. *average*) of a set of data gives us a rough idea about the average size of each data point. But in some situations this could be quite misleading. Suppose we have a village of 500 people which has one billionaire, two millionaires and the rest all have 50000 dollars each. The average wealth of the village is then:

```
      avg 1000000000,(2#1000000),497#50000
2053700
```

and this average (over two millions) is certainly very high but does not truly reflect the wealth of most people in the village. To this end we introduce the ***median*** of a sequence of real numbers *v*, which is a number in *v* greater or equal to half of the numbers in *v* and less than or equal to half of numbers in *v*. In other word, the median of *v* is a number in *v* which sits in the middle of *v* when arranged according to size. How do we calculate the ***median*** of a sequence of numbers?

We mentioned at the end of last chapter that there is a script file `standard.esf` of *pre-defined* functions as part of the ELI distribution, and the function `median` is included there to calculate the median of a sequence:

```
     )fcopy standard
     []CR 'median'
z<-median x;w;m
z<-((0.5*w[m]+w[m+1]),w[m<-_1+[]IO+~.0.5*#x])[[]IO+2|#w<-x[<x]]
```

the basic idea is to sort the input vector `x` into a variable: `w<-x[<x]`and find the element `w[m]` in the middle of `w` if the input has odd number of elements (when `2|#w` is `1`) so `[]IO+2|#w` would choose the second element in the expression inside the outmost parenthesis on the left or take the average of the two elements sit in the middle of the sorted list `w` (`0.5*w[m]+w[m+1]`) when `2|#w` is `0`. When apply this function `median` piece by piece to our example of a village with three very rich people, we find the *median* as follows

```
      x <-1000000000,(2#1000000),497#50000
      w<-x[<x]
      w
50000 50000 50000 50000 ... 50000 1000000 1000000 1000000000
      #w
500
      m<-_1+[]IO+~.0.5*#x
      m
250
      w[m]
50000
      w[m+1]
50000
```

```
      median x
50000
```

We notice that there is also an `avg` function in `standard.esf` which differs from the one we just introduced earlier:

```
      []CR 'avg'
z<-avg x
z<-(+/x)%_1^.#x
      avg 5

      avg ,5
5
      x<-3 4#12?.100
      x
14 76 46 54
22  5 68 94
39 52 84  4
      avg x
47.5 47.25 44.75
```

The difference is that if the input is a scalar it gives out an *empty vector* as a result while our old `avg` will give the input as a result. On the other hand, our old `avg` can only take vectors as inputs while the new `avg` from `standard` library can take matrix inputs and gives the averages on each row as the result.

The ***variance*** of a sequence *v* is defined to be the average of the *squared difference* from the *mean* of *v*. The ***standard deviation*** (denoted by the symbol **σ**) of a sequence *v* is simply the *square root* of the *variance*. It measures how spread out the numbers in *v* are (from the average of *v*), it is the function `stddev` in the `standard` library:

```
      z<-stddev x
z<-((+/(x-avg x)*.2)%#,x)*.0.5
      x<-7 8 9 2 10 9 9 9 9 4 5 6 1 5 6 7 8 6 1 10
      stddev x
2.765411362
```

For a sequence of numbers *v*, the ***mode*** of *v* is the number in *v* which appears most often in *v*. For example, if *v* = 2 3 2 1 3 5 8 3, then the *mode* of *v* is 3. How to find the *mode* of a numeric sequence *x*? We first apply the monadic function *unique* = to it to get all unique elements *u* of *x* and then form the outer product *u.*:=*x*. For example,

```
     x<-7 8 9 2 10 9 9 9 9 4 5 6 1 5 6 7 8 6 1 10
     u<-=x
     u
7 8 9 2 10 4 5 6 1
      u.:=x
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0
0 0 1 0 0 1 1 1 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0
      +/u.:=x
2 2 5 1 2 1 2 3 2
```

The resulting outer product is a *m* by *n* Boolean matrix, where *m* is the length of *u*, *n* is the length of *x* and each row's 1s indicating places in *x* equal to that element in *u*; to sum up each row is just counting the frequency of occurrence of each element in *u*. We store this into a variable *f* and take its maximum and ask where the maximum occurs in *f* and finally use this position of figure out the corresponding element in *u* which is the mode of *x*:

```
      ~./f<-+/u.:=x
5
      ?f=~./f
3
      u[?f=~./f]
9
```

To combine the computations we've done so far and to store it into a code segment which we can reuse later, we write a *short function*; we name this monadic function `mode`:

```
      {mode: u[?f=~./f<-+/(u<-=x).:=x]}
mode
      mode x
9
```

Given a sequence of numbers such as *x* above, we would like to construct a *frequency table* with unique elements of *x* in ascending order in one column and the corresponding frequencies in another column. We have the following short function:

```
      {fqtb: f<-+/(u<-=x).:=x;a<-u[i<-<u];b<-f[i];([] Number<-a;Frequency<-b)}
fqtb
      fqtb x
Number Frequency
----------------
1       2
2       1
4       1
5       2
6       3
7       2
8       2
9       5
10      2
```

`a` is the sorted list of `u` in ascending order while `b` is the corresponding list of frequencies; the last statement in the function definition is a way to produce a *table* in ELI which is explained in **§ 5.2** in [1], the ELI Primer.

6.2 basic combinatorics

Let S_n be a finite set of n elements. A ***r-permutation*** $P_{r,n}$ is an ordered r-*tuple* $(a_1, a_2,\ldots, a_r)$ of r distinct elements from S_n (to make it simple, we can assume that S_n is a set of numbers, then a *r-permutation* of S_n is just a vector *v* of distinct elements from S_n of length r), r≤n, for a positive integers r and n. Giving a r, r≤n, how many *r-permutations* $P_{r,n}$ are there for S_n? For the first element a_1 of $P_{r,n}$, we have n items to choose from S_n, for the second element a_2 of $P_{r,n}$, we have n-1 items to choose from what is left in S_n, and for the last element a_r we have (n-r)+1 choices. Hence, we have

$$P(n;r) = n*(n-1)*\ldots*((n-r)+1) \qquad (1)$$

distinct choices to form a $P_{r,n}$. In other word, the formula (1) above gives the number of distinct r-permutations of S_n. In particular, we have

$$P(n;n) = n*(n-1)*\ldots*1 \quad (2)$$

We note that (1) can rewritten as

$$P(n;r) = P(n;n)\%P(n-r;n-r) \quad (3)$$

there $P(n;k)$ are k-permutations of S_n, k=r, n, n-r.

Mathematically, the function $n \rightarrow P(n;n)$ is call the *factorial* of n (usually denoted by n!). In ELI, this *factorial* function is denoted by a slightly twisted (monadic) symbol `|.n` (remember `!n` is *iota* of n in ELI):

```
      |.0
1
      |.3
6
      |.10
3628800
      (|.10)%|.7
720
      {pnr0:(|.y)%|.y-x}
Pnr0
       10 pnr0 3
720
```

So we have the *factorial* function `|.n` and the short function `n pnr0 r` to calculate $P(n;r)$. However, there is a problem for the function `pnr0` when the arguments are large because the factorial function grows very quickly. To overcome this difficulty we write a new function which uses formula (1) instead of (3):

```
      {pnr:*/(-x)^.!y}
pnr
      100 pnr 3
970200
```

A r-***subset*** of a finite set is a subset $\{a_1, a_2,\ldots, a_r\}$ of r distinct elements of S_n. The difference between a r-*subset* and a r-*permutation* of S_n is that for a r-subset the order of elements doesn't matter. *The difference between a set S of numbers $\{a_1, a_2,\ldots, a_n\}$ and a vector $v=(a_1, a_2,\ldots, a_n)$ is that in S the *order* of elements doesn't matter, and we can regard v as a particular *representation* of S but different from S conceptually. How many r-subsets of S_n are there with $r \leq n$? We know there are r-permutations $P(n;r)$ of S_n, and we know there are r-permutations $P(r;r)$ of S_r. Hence, the answer is $P(n;r)\%P(r;r)$. Let us put in a short function:

```
       {bin:(|.y)%(|.y-x)*|.x}
bin
       10 bin 3
120
```

Actually, ELI has a dyadic primitive function ***binomial*** `x|.y` to do exactly that with r on the left-side:

```
      3|.10
120
      7|.10
120
      4|.11
330
```

This function is called *binomial* because it is the binomial coefficient $\binom{n}{k}$ of the expansion of the polynomial $(x+a)^n$.:

$$(x+a)^n = \sum_{k=0}^{n} \binom{n}{k} x^k a^{n-k}$$

The difference of an r-permutation and an r-subset (a combination) can be illustrated by the following problem: there are two ways to form a committee of three people out of a group of 100 people; in case i) everyone in the committee are equal, in case ii) there is a chairman, a treasury and a secretary of the committee. In case (ii) then answer is $P(100;3)$ and as we see above the answer is `970200`. In the second case, the answer is

```
     3|.100
161700
```

A r-***sample*** of a set S_n of n elements is an ordered r-tuple $(a_1, a_2,\ldots, a_r)$ of r *not necessarily distinct* elements from S_n. How many r-*sample*s of S_n are there? A similar argument as that for the number $P(n;r)$ applies except that at each stage we have *n* choices instead of a decreasing number of choices. Hence, the number of r-*sample*s of S_n is n^r. We note that there is no requirement of r≤n as in r-permutation case since a_i and a_j can be the same.

Let S_n be {0,1}, i.e. n=2, and then a r-*sample* of S_n is just a bit-string of length r. Recall that the *roll* function `?.n` which randomly picks an integer from `!n` in **§ 1.4**. So, if we set `[]IO=0`, then `?.!2` will randomly pick a bit and an r-*sample* of S_n can be generated and counting the number of its *on*-bits as follows:

```
      []IO←0
      ?.32#2
0 0 0 1 0 1 0 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 0
      +/0 0 0 1 0 1 0 0 1 1 0 1 0 1 1 1 0 0 1 1 1 1 1 1 1 0 0 0 1 1 1 0
18
```

We can designate 1 as the *head* and 0 as the *tail* of a coin and regard an r-*sample* of S_n as the result of tossing a coin r-times. We know that the total number of heads turned up after you throw a coin 1000 times is close to 500. What is the probability of this total number being exactly 500? We can look at a thousand coin-tossing as a set of 1000 items and we are counting the number of ways to pick 500 items out of it, and then divide it by the size of the space of all outcomes which is about 2.5%:

```
      (500|.1000)%2*.1000
0.02522501818
```

Let *A* be a set $\{a_1, a_2,\ldots, a_n\}$ of n elements representing n different ways, and let *B* be a set $\{b_1, b_2,\ldots, b_m\}$ of m elements. The *Cartesian* ***product*** *A•B (A* ***cross*** *B)* is defined to be to be the set of all (a_i, b_j) ways with a_i in *A* and b_j in *B*. The ***multiplication principle*** then says *A* ***cross*** *B* has n*m elements. This concept of product *A•B* of two finite sets can easily be extended to product $A_1 \bullet A_2 \bullet\ldots\bullet A_k$ of multiple finite sets $A_1, A_2,\ldots,A_k$ of sizes $n_1, n_2,\ldots,n_k$. The *extended* ***multiplication principle*** then says the size of the product $A_1 \bullet A_2 \bullet\ldots\bullet A_k$ is $n_1 * n_2 *\ldots* n_k$.

Recall what we stated in **§** 1.1 that in ELI, as in many other programming languages, a *variable name* or a *function name* must start with a letter, then followed by a digit or a letter or the character '_' but that the *under-bar* character cannot be the last character of a *name*. How many possible (variable/function) *name*s of 3 characters $c_1c_2c_3$ are there in ELI? Let *A* be the set of letters, upper case and lower case together; *A* has 52 elements. Let *B* be the set of letters and digits; *B* has 62 elements. Finally, let $C = B \cup \{\text{'\_'}\}$. The set of all legal 3-character names in ELI is the set

$$N_3 = \{\, c_1c_2c_3 \mid c_1 \in A,\ c_2 \in C,\ c_3 \in B\}$$

This is the product $A \bullet C \bullet B$ of A, C and B. Hence, N_3 has 52*63*62= 203112 elements by the multiplication principle.

Let A be a set $\{a_1, a_2, \ldots, a_n\}$ of n elements representing n different ways, and let B be a set $\{b_1, b_2, \ldots, b_m\}$ of m elements; assume $A \cap B = \{\}$, . i.e. A and B are disjoint. The *sum* $A+B$ (A ***or*** B) is defined to be to be the set of all c_k ways with either $c_k = a_i$ in A or $c_k = b_j$ in B. The ***addition principle*** then says (A ***or*** B) has n+m elements. This concept of sum $A+B$ of two disjoint finite sets can easily be extended to sum $A_1 + A_2 + \ldots + A_k$ of multiple *mutually disjoint* finite sets $A_1, A_2, \ldots, A_k$ of sizes $n_1, n_2, \ldots, n_k$, $A_j \cap A_i = \{\}$ if $j \neq i$. The *extended* ***addition principle*** then says the size of the *sum* is $n_1 + n_2 + \ldots + n_k$.

Let N be the set of all legal names of a variable/function in ELI up to 3 characters. Then N is the *sum* of N_1 (the set of names with one character only), N_2 (the set of names with two characters) and N_3. The sizes of N_1, N_2 and N_3 are 52, 52*62 and 203112 respectively. The three sets N_1, N_2 and N_3 are clearly mutual disjoint. Hence, the size of their sum N is 52+ 3224+203112=206388.

Let U be the set of totality of all possible ways under consideration and it is of size n; and let A be a subset of U of size m. The set $\sim A$ (***not*** A) is defined to be the set of those elements in U which are not in A. The ***complement principle*** then says that the size of $\sim A$ is n-m.

How many possible variable names in ELI up to 3 characters are there which do not contain a '_' character? The set U in this case is the set N we studied above, its size is 206388. The set A is this case is the set of names in N with a character '_'. Since '_' cannot appear as the first or the last character of a name, A is a subset of N_3 and it is of size 52*1*62= 3224 because the middle character is fixed with one choice only. Hence, the answer is the size of $\sim A$ which is 206388-3224=203164.

6.3 elementary probability theory

Probability theory has its roots in attempts to analyze games of chance such as in gambling (a *game of chance* is a game whose outcome is strongly influenced by a randomizing device, upon which contestants may bet on money) by mathematicians from 16th century on. But its precise description was first given by the French mathematician Pierre-Simon Laplace in 1795. Today, techniques based on probability theory are at the core of Big Data analysis and Artificial Intelligence.

An ***experiment*** is an occurrence whose outcome is subject to chance such as flipping a coin or rolling a die. It has a ***sample space***: the set S of all possible outcomes. For example, the sample space of flipping a coin is {0, 1} where 0 represents a tail while 1 represents a head; and D={1,2,3,4,5,6} (the numbers on a die's faces) in the case of throwing a die. If our experiment is tossing a coin 1000 times, then the sample space B is the set of all Boolean strings of length 1000. A sample space needs not to be discrete. For example, our experiment can be picking a point from the *unit interval* [0,1], i.e. set of real numbers x with $0 \leq x \leq 1$, and the sample space is [0,1]. A sample space must have some "*size*". For a discrete finite sample space, this is just the number of elements in S. For example, the size of B is 2^{1000}. For a non-discrete sample space the size is some kind of *measure*. For example, the size of [0,1] is 1, i.e. the *length* of the interval.

An ***event*** E is one possible outcome of the experiment. For example, in the case of rolling a die, one event, E_1, is that the face number of die showing up is an even number. In the case of a point in a unit interval, one event E_2 is that $x \leq 0.5$. One can look at an event E as a particular *subset* of the sample space S, i.e. $E \subset S$. The ***probability*** of an event E is the *ratio* of its "*size*" to that of the sample space S which we denote by $\boldsymbol{p}(E)$. In the cases E_1 and E_2 above, we have both $\boldsymbol{p}(E_i) = 0.5$, i=1,2.

Let A and B are two events of a sample space S, then the probability of the event $A \cap B$, the *intersection* of A and B may or may not be the following (see section below) in contrast to the Cartesian product $A \bullet B$ case in the preceding section

$$\boldsymbol{p}(A \cap B) = \boldsymbol{p}(A) * \boldsymbol{p}(B) \tag{4}$$

On the other hand, for two *disjoint* events A and B (i.e. $A \cap B = \{\}$) of a sample space S, then the probability of the event $A \cup B$, the *union* of A and B is

$$\boldsymbol{p}(A \cup B) = \boldsymbol{p}(A) + \boldsymbol{p}(B) \tag{5}$$

And finally, for an event A of a sample space S, let S-A or A' be the event consisting of all occurrences in S not in A, i.e. the *complement* of A, we have its probability as follows:

$$\boldsymbol{p}(A') = 1 - \boldsymbol{p}(A) \tag{6}$$

We can see that formulae (4)-(6) of combining probabilities of events are just a reformulation of the multiplication* principle, the addition principle and the complement principle in basic combinatorics in the previous section. Indeed, elementary probability theory is a set-theoretic extension of intuitive ideas developed in basic combinatorics.

What is the probability of the event D that when you throw two dice and the sum of the resulting face numbers is exactly 10? If you get a 1, 2 or 3 in your first die, you would not end in a sum of 10 no matter what number turns up in your second die. Let us denote the event that the first throw resulting in number 4 by A_4, and let A_5 and A_6 be the events associated with the face numbers 5 and 6. Similarly, we define B_4, B_5 and B_6 to be the events that the number turns in the second die is 4, 5 and 6 respectively. The event $A_4 \cap B_6$ is part of D, i.e. you get a 4 and a 6 in your two throws. We assume the dice is not rigged so each of the six numbers has an equal chance of turning up. Hence, p(C)=1/6 for C to be any of the A_i or B_j here and $\boldsymbol{p}(A_4 \cap B_6)$=1/36. Therefore,

$$\boldsymbol{p}(D) = \boldsymbol{p}((A_4 \cap B_6) \cup (A_5 \cap B_5) \cup (A_6 \cap B_4)) = \boldsymbol{p}(A_4 \cap B_6) + \boldsymbol{p}(A_4 \cap B_6) + \boldsymbol{p}(A_4 \cap B_6) = 3*(1/36) = 1/12$$

Note that each A_i is disjoint from A_j for $i \neq j$ (you cannot turn up two numbers in the same die. Hence the components in the union above are mutually disjoint, so formula (5) applies.

In a class of 24 students, what is the probability of two students having the same birthday? First let us make a simplifying assumption that there is no one born on February 29 of a leap year. So, there are 365 possible birthdays for each student and the size of our sample space is 365^{24}. Let us denote A to be the event that no two students have the same birthday, and then ask what is $\boldsymbol{p}(A)$? For that to happen, the first student can choose any day to be his/her birthday, for the second student he/she has 356-1 days to choose from, the probability p_2 of the event of two students have different birthdays is p_2=(365-1)%365, the probability p_3 of 3 students have distinct birthdays is p_2*(365-2)/365, and the probability p_{24} of all 24 students have distinct birthdays is $\boldsymbol{p}(A) = p_{23}*(365-23)\%365$; and the probability of two students in a class of 24 having the same birthday is $\boldsymbol{p}(A') = 1-\boldsymbol{p}(A)$. Put the detail calculation in ELI as follows:

```
      24^.$!365
365 364 363 362 361 360 359 358 357 356 355 354 353 352 351 350 349 348 347 346 345 344 343 342
      (24^.$!365)%365
1 0.99726027 0.994520548 0.99178082 0.989041096 0.98630137 0.9835616438 0.9808219178 0.9780821918
      0.9753424658 0.9726027397 0.9698630137 0.9671232877 0.9643835616 0.9616438356 0.9589041096
      0.9561643836 0.9534246575 0.9506849315 0.9479452055 0.9452054795 0.9424657534 0.9397260274
      0.9369863014
      */(24^.$!365)%365
0.4616557421
      1-*/(24^.$!365)%365
0.5383442579
```

The last expression is by formula (6) for $\boldsymbol{p}(A')$. We can put this calculation into a short function with the number of students, or people in a party as the right parameter x:

```
      {p_samebdat:1-*/(x^.$!365)%365}
p_samebdat
      p_samebdat 24
0.5383442579
      p_samebdat 50
0.9703735796
```

We see that while for a group of 24 people the probability of two persons having the same birthday is a little more than half, for a group of 50 people, you are very likely to encounter people in the party having the same birthday.

We can now put our discussion so far in a more formal setting: A ***probability space*** is a mathematical triplet $(S, F, \boldsymbol{p})$ where

1. S is a *sample space* of all possible outcomes of an experiment.
2. F is a collection of subsets of S called *events* each of which containing zero or more outcomes.
3. an assignment of *probabilities* to the events, i.e. a real-valued *function* $\boldsymbol{p}$ from F to [0,1].

For a finite (discrete) set S, F usually is just the collection of all subsets of S and the $\boldsymbol{p}$ is a ratio of the *size* of E in F relative to the size S as in many examples we have studied. In more general case, the "*size*" in question is replaced by something called a *measure* on S, and F is a σ*–algebra* (a collection F of subsets of S with the following three properties: i) the empty set $\{\}$ and S are in F, ii) if E_1 and E_2 are in F then $E_1 \cap E_2$ and $E_1 \cup E_2$ are in F too, iii) all *event*s in F are *measurable*). An example is S= [0,1] and F be the set of all (Lebesgue-)*measurable* subsets E of [0,1], and $\boldsymbol{p}$ of E is just the measure of E.

A ***random variable*** v on sample space S (of a probability space $(S, F, \boldsymbol{p})$) is a *real valued function* on S whose value is subject to valuations due to *chance*. For example, the top face value f when rolling a die is a random variable on the sample space {1, 2, 3, 4, 5, 6}. In general, S needs not be a finite set. For example,

$$f(\mathrm{y}) = \boldsymbol{p}(\{\mathrm{x}|\ \mathrm{x}^2 \leq \mathrm{y},\ \mathrm{x}[0,1]\})$$

is a *random variable* on the unit interval [0,1].

The ***mathematical expectation*** E of a random variable v, or just ***expectation*** of v or the ***expected value*** of v, on a probability space $(S, F, \boldsymbol{p})$ is the "*average*" value of v on S. For a discrete set S, this is defined by

$$\mathrm{E}(v) = \sum_{x \in S} v(\mathrm{x}) * \boldsymbol{p}(\mathrm{x}) \qquad (7)$$

For example, the *expected value* of the top face (of a die) function f is

```
    +/(!6)%6
3.5
```

because each $\boldsymbol{p}(\mathrm{x})$ is 1/6. This says that on average, you can expect a top face value of 3.5 after many rolls. For a random variable v on non-discrete probability space $(S, F, \boldsymbol{p})$ the expectation of v is calculated as an *integral* over S against a *measure* μ representing $\boldsymbol{p}$ with the summation $\sum_{x \in S}$ replaced by an integral sign over S and $\boldsymbol{p}(\mathrm{x})$ replaced by $\mathrm{x}dx$. We are not getting into details here since we have not explored the subject of calculus in this tutorial. Overall, we see that the expect value of a random variable on a sample space S as a "*weighted*" probability, or *non-uniformly distributed* probability on S.

The concept of *expectation* actually preceded that of probability as it relates to gambling. An example from the Dutch mathematician C. Huygens in the use of his concept of expectation is the following game: if two dice are rolled. If a seven (sum of two top faces) is thrown, play X gets amount a; if a ten is thrown player Y gets amount b; and in any other cases, the amount a is divided evenly between the two players. What are amounts players X and Y *expected* to win? We can regard X and Y as two random variables on sample space S of two dice's face values {(x,y)} which is of size 36. The size of the event A of a seven is twice that of {(1, 6), (2, 5), (3, 4)}, i.e. 6, and event B is {(4,6), (6,4), (5,5)} of size 3, and the event C of outcomes neither in A or B is of size 36-(6+3)= 27. Hence, the expected values for X and Y are

$$E(X)= (6a+27*0.5*a)/36 = 13a/36 \quad (8)$$
$$E(Y)= (3b+27*0.5*a)/36 = b/12+9a/24 \quad (9)$$

If $a=b$, then E(Y) is $11a/24$, player Y has expected advantage over player X as `0.4583` vs `0.3611`. In (8) and (9), we use the fact that if $\{E_1, E_2, \ldots, E_n\}$ is ***partition*** of S into events, i.e. E_k are *mutually disjoint* and their *union* is S, and that on each E_k the value of the random variable v is a_k, k=1,…,n, then formula (7) becomes

$$E(v)= a_1*\boldsymbol{p}(E_1)+ a_2*\boldsymbol{p}(E_2)+\ldots+ a_n*\boldsymbol{p}(E_n) \quad (10)$$

6.4 conditional probability

For a probability space $(S, F, \boldsymbol{p})$ and two events A and B in S, we define the ***conditional probability*** of A given B, denoted by $\boldsymbol{p}(A|B)$, to be a measure of the probability of an event A under the assumption that another event B has occurred, or the likelihood that an experiment is in event A given that another event B has already happened. Let the event B' be *complement* of the event B in S. Clearly, B and B' are mutually exclusive and any experiment is either in B or in B'. Hence,

$$\boldsymbol{p}(A) = \boldsymbol{p}(A|B)\,\boldsymbol{p}(B) + \boldsymbol{p}(A|B')\,\boldsymbol{p}(B') \quad (1)$$

If we replace A above by the event $A \cap B$, i.e. the collection of experiments each of which is both in A and B, we have

$$\boldsymbol{p}(A \cap B) = \boldsymbol{p}(A|B)\,\boldsymbol{p}(B) \quad (2)$$

since $\boldsymbol{p}(A \cap B\,|B)= \boldsymbol{p}(A|B)$ and $\boldsymbol{p}(A \cap B\,|B')=0$. If $\boldsymbol{p}(B)> 0$, (2) can be rewritten as

$$\boldsymbol{p}(A|B) = \boldsymbol{p}(A \cap B)/\boldsymbol{p}(B) \quad (3)$$

And if we switch the role of A in (2) with that of B, we have

$$\boldsymbol{p}(A \cap B) = \boldsymbol{p}(B|A)\,\boldsymbol{p}(A) \quad (4)$$

Let us have an example. Suppose the probability of a person in a small town in upper state New York who has visited New York City (event B) is 70%, that who has visited Boston (event A) is 55% and that who has visited both cities (event $A \cap B$) is 45%. If a randomly selected person in that town has already visited New York City, what is the likelihood that this person has visited Boston as well? According to (3), we have $\boldsymbol{p}(A|B) = 0.45/0.7= 0.642857$.

Two events A and B in S are ***statistically independent*** if both

$$\boldsymbol{p}(A|B) = \boldsymbol{p}(A) \qquad \text{and} \qquad \boldsymbol{p}(B|A) = \boldsymbol{p}(B) \quad (5)$$

are true. In that case, we then have

$$\boldsymbol{p}(A \cap B) = \boldsymbol{p}(A)\,\boldsymbol{p}(B) \quad (6)$$

This is a formalization of the important concept in probability that events A and B do not *influence* each other, or the old idea that in tossing a coin several times, the coin never remembers: let A be the event that the first toss of a coin results in a head and B be the event that the second toss of the same coin results in a tail. Then, clearly both sides of (6) above equal to ¼. On the other hand, suppose our sample space is that of rolling a pair of dice; A is the event that the first die turns up a 2 and B is the event that the sum of resulting tops of the two dice is 3. We have $\boldsymbol{p}(A)$=1/6, B ={(1,2), (2,1)} and $\boldsymbol{p}(B)$=1/18 and $A \cap B$ ={(2,1)}, so the left side of (6) above is 1/36 while the right side is 1/108. We note that $\boldsymbol{p}(A|B)$, the likelihood of A occurs given that B already has occurred (so only 1 or 2 can turn up for the first die) is ½, i.e. (2) above holds. Hence these two events are not statistically independent, in other word, event A is influenced by event B.

When we combine the formulae (2) and (4) above, we get the **Bayes' theorem**:

$$\boldsymbol{p}(A|B) = \boldsymbol{p}(B|A)\,\boldsymbol{p}(A) \,\%\, \boldsymbol{p}(B) \tag{7}$$

$$\boldsymbol{p}(B|A) = \boldsymbol{p}(A|B)\,\boldsymbol{p}(B) \,\%\, \boldsymbol{p}(A) \tag{7a}$$

Bayes' theorem is an application of conditional probability formula to successive events in order to find the likelihood of a particular cause having brought about a particular result. It is an important tool used in *statistical inference*. The theorem was first formulated by Rev. Thomas Bayes and further developed by S. Laplace.

A simple example of an application of this theorem is the following: Suppose our sample space in the *population* in U.S., 0.2% of the people in U.S. are 65 years old (event A) and 1% of the population has cancer (event B); we also know that the probability of a person with cancer who happens to be 65 years old is 0.5% ($\boldsymbol{p}(B|A)$). What is the chance of a 65 year old person in U.S. having cancer, i.e. ($\boldsymbol{p}(A|B)$)? According to (7), we have

```
      (0.005*0.01)%0.002
0.025
```

i.e. 2.5%, a bit more than doubling the chance of having cancer in U.S. general population.

Many times, we do not know directly the denominator in formula (7) directly but do have other pieces of information to get to it. Suppose B_1, B_2 and B_3 are mutually disjoint and their union is the sample space S, in other word, $\{B_1, B_2, B_3\}$ is a *partition* of S. With the same argument as that in deriving formula (1) with B_1 replacing B etc. we have the following:

$$\boldsymbol{p}(A) = \boldsymbol{p}(A|B_1)\,\boldsymbol{p}(B_1) + \boldsymbol{p}(A|B_2)\,\boldsymbol{p}(B_2) + \boldsymbol{p}(A|B_3)\,\boldsymbol{p}(B_3) \tag{8}$$

Hence, we can use the right-hand side of (8) to replace $\boldsymbol{p}(A)$ in (7a) if these information are available.

Suppose we find in a study that 30%, 50% and 20% of families in a nation (B) are of low-income, middle-income and affluent respectively (B_1, B_2, B_3). Further, we call a kid *tall* (A) if he/she is of height at least 6 feet. We also know respectively that 4%, 5% and 3% of kids from low-income, middle-income and affluent families are tall. What is the probability of a *tall* kid being from affluent family? Substituting B_3 for B in (7a), we have

$$\boldsymbol{p}(B_3|A) = \boldsymbol{p}(B_3|A)\,\boldsymbol{p}(B_3) \,\%\, \boldsymbol{p}(A) \tag{9}$$

Combining (9) with (8) we have the following computation for $\boldsymbol{p}(B_3|A)$:

```
      pBs<-0.3 0.5 0.2
      pA_Bs<-0.04 0.05 0.03
      (pA_Bs)[3]*pBs[3]
0.006
      (pA_Bs)[3]*pBs[3]%+/pA_Bs*pBs
0.1395348837
```

This example leads us to the following general form of the **Bayes' theorem**:

Let $\{B_1, B_2, \ldots, B_n\}$ be a partition of the sample space S of an experiment. If for i = 1,2,…,n, $\boldsymbol{p}(B_i)>0$, then for any event A of S with $\boldsymbol{p}(A)>0$

$$\boldsymbol{p}(B_k|A) = \boldsymbol{p}(B_k|A)\,\boldsymbol{p}(B_k) \% (\boldsymbol{p}(A|B_1)\,\boldsymbol{p}(B_1) + \boldsymbol{p}(A|B_2)\,\boldsymbol{p}(B_2) + \ldots + \boldsymbol{p}(A|B_n)\,\boldsymbol{p}(B_n)) \tag{10}$$

In statistical applications of (10), $B_1, B_2, \ldots, B_n$ are called *hypotheses*, $\boldsymbol{p}(B_i|A)$ is called *prior* probability of B_i, and the conditional probability $\boldsymbol{p}(B_i|A)$ is called *posterior* probability of B, after the event occurrence of A.

References

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

[2] W.-M. Ching, Introduction to Programming with Arrays using ELI, http://fastarray.appspot.com/, 2014.

[3] John R. Durbin, Mathematics, Its Spirit and Evolution, Allyn and Bacon, Boston, 1973.