A one line of ELI to count the number of heads in a sequence of coin tossing

By Wai-Mee Ching, 2016-5-16

ELI is an interactive programming language system with its core derived from the classical array programming language APL invented by Ken Iverson. Thus, ELI has 80 monadic (i.e. having a right argument only) and dyadic (i.e. having a left and a right argument) primitive (i.e. provided by the system) functions together with 6 operators (they apply to primitive functions), each one of them is represented symbolically by one or two ASCII characters. For example, $a < b$ is the less than function while $a <. b$ is the encode function, and $a <- b$ is the assign function.

A line of ELI code consists of one or several expressions separated by a ‘;’ and separate expressions are executed from left to right as in other programming languages. However, each expression is evaluated from right to left with all functions having equal precedence while an argument to a function inside a pair of parenthesis is evaluated first before the execution of the function involved. For example, we have

\[
\begin{align*}
3 \times 2 + 4 & = 18 \\
(3 \times 2) + 4 & = 10
\end{align*}
\]

The line you enter is the line with indentation and the system responds with a line without indentation. The result of one function can feed as an argument to the next function or be assigned to a variable for further use.

Now, let us consider a problem: throw a coin 1000 times and count the number of heads turned up in this experiment. We use a 0 to represent a tail and a 1 to represent a head while a throw is simulated by a random number generator which produces a 0 or 1 with equal chance. ELI has a system variable called \[\text{IO}\] which is set to 1. The monadic primitive function !\[n\] generates \[n\] consecutive integers starting from \[\text{IO}\] and \[\text{IO}\] can be changed to 0 if we like. We see that

\[
\begin{align*}
!10 & = 1 2 3 4 5 6 7 8 9 10 \\
[\text{IO}] & < 0 \\
!10 & = 0 1 2 3 4 5 6 7 8 9
\end{align*}
\]

ELI has another primitive function \(?.n\) which randomly picks a number (with equal probability) from \(!n\) each time it is executed. We see that

\[
\begin{align*}
?.10 & = 1 \\
?.10 & = 7 \\
?.2 & = 0
\end{align*}
\]

Now one prominent feature of APL/ELI is that for a class of primitive functions called scalar functions (includes all arithmetic functions) which operates on an array is the same as it operates on each element of the array. In particular, when an array is a vector (i.e. a 1-dimensional array), we have for the inverse function \(%/a\) the following

\[
\begin{align*}
\%2 & = 5 8 10 3 \\
0.5 & = 0.2 0.125 0.1 0.3333333333 \\
2 \times !10 & = 0 2 4 6 8 10 12 14 16 18
\end{align*}
\]
It happens that the roll function \( \, \) is also a scalar function:

\[
\begin{array}{ccc}
? & 3 & 8 & 10 \\
1 & 4 & 8 \\
? & 2 & 2 & 2 \\
0 & 0 & 1
\end{array}
\]

Now for a vector \( v \), we can sum it up by applying the reduction operator \( / \) to the addition function \( + \) to its right and the reduction operator can also apply to other suitable primitive functions such as multiplication:

\[
\begin{array}{c}
+/1 4 8 \\
1 3 \\
+/0 0 1 \\
1 \\
+/1 4 8 \\
32 \\
+/0 0 1 \\
0
\end{array}
\]

We are almost there. The question now is how to generates a thousand 2’s to the right of the roll function \( ? \). before we apply the plus reduction to the result of roll.

Each array \( a \) in ELI has a shape \( s \) which is a vector consisting of lengths of each dimension of \( a \). For a matrix \( a \) its shape is a vector of two elements indicating the height and the width of \( a \). For a vector \( v \) the shape of \( v \) is simply its length. For an array \( a \), applying the primitive function shape \( \# \) to it will yield its shape. In case \( a \) is a 3 by 4 matrix we have:

\[
\begin{array}{cccc}
a & 0 & 1 & 2 & 3 \\
& 4 & 5 & 6 & 7 \\
8 & 9 & 10 & 11 \\
3 & 4 \\
& 8 & 9 & 10 & 11 & 10 & 15 \\
6
\end{array}
\]

There is a dyadic primitive function reshape \( \# a \) which results in an array \( b \) of shape \( s \) using elements from \( a \) (Here we notice another unique feature of ELI inherited from APL that many primitive function symbols can represent a dyadic or monadic function depending on whether it has a left argument):

\[
\begin{array}{cccc}
3 & 4 & 10 \\
0 & 1 & 2 & 3 \\
& 4 & 5 & 6 & 7 \\
8 & 9 & 0 & 1 \\
5 & 8 & 9 & 10 & 11 & 10 & 15 \\
8 & 9 & 10 & 11 & 10 & 11 & 10 & 15 \\
10 & 2 \\
2 & 2 & 2 & 2 & 2 & 2 & 2 & 2
\end{array}
\]

We observe that if there are not enough elements in \( a \) to form the required \( b \), reshape will reuse the elements in \( a \) and if there are more elements in \( a \) than what is required then a tail section of \( a \) will be dropped. In any case the one line ELI code to solve our problem is as follows:

\[
+/?.1000\#2
\]

502
We would like to save this piece of code into a user defined function. A defined function $f$ in ELI has a name (in contrast to a primitive function being denoted by a symbol), a right argument or a left-right argument pair (or no argument at all). For a simple defined function, we can write it in short-function form which is a line of code preceded by the function name and enclosed in a pair of curly brackets where $x$ is assumed to be the right argument, $y$ is the left argument if present, and $z$ or value of the last expression is the return result of the function $f$. In our case, we have

\[
\{\text{cnt\_hds: } []\text{IO}<0; +/?.x#2} \}
\]

where the right argument $x$ is the number of independent throws of a coin. The second line above is ELI’s response to a successful definition of a short function named `cnt_hds`. Note that it is important to set `[]\text{IO}` to 0 first because the system default value for `[]\text{IO}` is 1. Now, we can call the function with different arguments:

\[
\begin{align*}
\text{cnt\_hds} & \text{ 1000} \\
501 & \text{ cnt\_hds} \text{ 2000} \\
987 & \text{ cnt\_hds} \text{ 8000} \\
3981 & \text{ cnt\_hds} \text{ 8000} \\
0.501 & \text{ 501}\%\text{1000} \\
0.981 & \text{ 981}\%\text{2000} \\
0.4905 & \text{ 0.4905} \\
0.497625 & \text{ 3981}\%\text{8000} \\
\end{align*}
\]

We notice that the first call above yields a slight different value than what we have earlier (501 vs 502). This is due to the fact that two different sequences of throws are two different experiments. We also observe that the number of total heads divided by the number of throws is fairly close to 0.5 indicating that the coin throw is fair.

We used this one line ELI code to illustrate ELI’s primitive-based array-oriented programming, and its dataflow style of coding, i.e. the output of one operation is used as an input to the next operation. The resulting succinctness naturally leads to code clarity and programming productivity. To learn more about ELI and to download the free software please visit [http://fastarray.appspot.com](http://fastarray.appspot.com) or [http://www.sable.mcgill.ca/~hanfeng.c/eli/](http://www.sable.mcgill.ca/~hanfeng.c/eli/).