AN INTRODUCTION TO ARRAY PROGRAMMING IN KLONG

Nils M Holm

Contents

Preface

Array programming languages are increasingly popular, but a lot of criticism focuses on their syntax. The author would like to argue that the terse syntax of array languages is a feature rather than a relict from less civilized days. For example, imagine you wanted to find the maximum distance between two subsequent values in a vector of ascending numbers:

```
[0 6 13 22 25 33 41 42 43 49 49]
```

The Klong program that achieves this goal looks like this:

```
|/--:'
```

and is pronounced “Max-Over Negate Minus-Each-Pair”. Reading from the right to the left: Minus-Each-Pair computes the negative distance between each pair of numbers in the vector. Negate will negate these distances, making them positive. Max-Over finds the maximum of the distances.

Some people might prefer a more familiar syntax, like

```
over(max; negate(eachpair(minus; x)))
```

But the author would like to suggest that

```
Max-Over Negate Minus-Each-Pair
```

is actually more comprehensible.

Once you get used to that notation, common operations can be replaced with the obvious characters (yes, Max and Or are the same operator; see page 35):

```
| Over - - Each-Pair
```

Then, after learning a few new operator symbols:

```
|/ - -:'
```

Finally the white space can be removed, because it does not carry any information anyway:

```
|/--:'
```

After getting familiar with the notation, it becomes beautiful and

very comprehensible. Once the initial barrier is passed, more verbose representations of algorithms suddenly may start to look long-winded and clumsy.

It has often been suggested that creating an array language with more familiar syntax might be a great achievement. In fact, such a language would not be hard to implement, but it would not offer any advantages over existing array languages, because it is not just the semantics, it is the *syntax* of array languages that makes them interesting.

Go back and have a look at the example using familiar syntax. Then look at the Klong example again. Read it loud! Which one looks more comprehensible?

Apropos reading loud: It is essential to learn the names of all Klong operators when learning the language! The program

```
|/--:'
```

just looks like a bunch of characters, but knowing that they are pronounced Max (`|`), Over (`/`), Negate (`-x`), Minus (`a-b`), and Each-Pair (`:'`), it already starts to make sense.

Once array programming is mastered, it offers an elegant and intuitive way to solve problems. However, it requires to readjust existing code reading and writing habits a bit. Klong programs are *terse*. If you miss a character, you are lost, so Klong programs must be read very carefully.

Even writing programs in array languages is a different process. You will have to recognize arrays in the problems you solve in order to make the best use of the language.

This is what this text is about: understanding Klong programs and solving problems by array manipulation.

Welcome to the world of array programming! Enjoy the tour!

Nils M Holm, January 2018

Using This Text

There are lots of examples in this text, so it is a good idea to install a Klong interpreter on your system. Its source code is available at **t3x.org** and it should work without modification on most Unix systems, Plan 9, and Cygwin.

In example dialogs, user input is indented and answers printed by the interpreter are outdented:

```
      1+2
3
```

Multiple *expressions* may be given in a single line, separated by semicolons, but only the last one will print a result:

```
      a::5;b::7;a+b
12
```

In general, Klong programs, like the above, and function names and operators, like **.rn** or **+**, print in a bold monospace font. Names of Klong operators and adverbs, like Plus or Each-Pair, will be capitalized. Sometimes operators will have some symbolic arguments attached in order to indicate whether they are monadic (single-argument) or dyadic (two-argument) verbs. For instance, **-a** is called Negate and **a-b** is called Minus.

Variables, like x, gcd, or cos will be rendered in italics.

Mathematical equations, like $(\sum L)/n$, are typeset in italics and are not valid Klong expressions, even though they often translate pretty directly to programs, like **(+/L)%n**.

The book is intended to be read front-to-back. Many concepts are based on things that are explained earlier in the text, so skipping ahead may hurt your progress.

Although this is not strictly necessary, it is *highly* recommended to try the examples. Also, do not move ahead before you have really understood the programs. Klong is a bit like mathematics: if you just “get the idea” without paying attention to the details, you might get lost later.

Basic Vocabulary

Arrays

The fundamental data type in array languages is, naturally, the *array*. Arrays come in multiple flavors, the simplest form being the *vector*. Here is a vector containing five integers:

```
      [1 2 3 4 5]
```

Klong vectors are collections of data objects enclosed by square brackets. For now, the objects contained in vectors will be integer numbers. Klong offers the usual *operators* that also can be found in programming languages of other paradigms, like *Plus* (**+**) and *Times* (*****):

```
      1+2
3
      3*4
12
```

However, Klong operators can also be applied to vectors instead of *scalar* or *atomic objects*. Basically an "atomic" object is any object that is not a vector. For instance, adding a scalar to a vector will add the value of the scalar to all elements of the vector:

```
      2+[3 4 5]
[5 6 7]
      [3 4 5]+10
[13 14 15]
```

Adding two vectors will add their elements pairwise:

```
      [1 2 3]+[4 5 6]
[5 7 9]
```

Other operations work on entire vectors, like finding the number of elements contained in them using the *Size* (**#**) operator:

```
      #[1 2 3]
3
      #[]
0
```

The last example takes the size of the *empty vector*, `[]`.

`[]` is sometimes also pronounced "nil" ("not in list"), which indicates that `[]` is a list rather than a vector. In fact Klong does not make any difference between a *list* and a vector. For instance, elements can be *joined* to either side of a list or vector using the *Join* (`,`) operator:

```
      [1 2 3],4
[1 2 3 4]
      0,[1 2 3]
[0 1 2 3]
      [1 2],[3 4]
[1 2 3 4]
```

Sometimes it makes sense to view scalar data types as single-element vectors, but this is not entirely true in Klong, as we will discover later. However, joining two scalars will still yield a vector:

```
      5,7
[5 7]
```

The Join operator is particularly useful for creating lists or vectors that contain "dynamic" elements, i.e. elements whose values are not constant:

```
      (3*a^2),(4*b),c
```

The resulting vector will contain the values of the expressions $3a^2$, $4b$, and c which, of course, depend on the values of the variables a, b, and c. If you tried to include an expression, like `3*a^2`, in a bracket-delimited array, something entirely different would happen:

```
      [3*a^2]
[3 :* :a :^ 2]
```

This is because the brackets delimit an *array literal*, which is a notation for array *constants*, i.e. arrays that do not contain any expressions, but only other data objects, like numbers, strings, and (like in the above case) symbols. We will get back to this later (page 40).

Of course, you can not only *add* objects to vectors, you can also extract or remove elements from them. The *Take* (`#`) operator

fetches the first (or last) n elements from a vector, depending on the sign of its first operand:

```
      2#[1 2 3 4 5]
[1 2]
      (-2)#[1 2 3 4 5]
[4 5]
```

Similarly, the *Drop* (_) operator removes the given number of elements from the head or tail of a list:

```
      2_[1 2 3 4 5]
[3 4 5]
      (-2)_[1 2 3 4 5]
[1 2 3]
```

Note that *all* Klong operators associate to the *right*, so

```
      -2_[1 2 3 4 5]
```

would actually be equal to

```
      -(2_[1 2 3 4 5])
```

This is also a valid Klong program, but it first drops two elements from the list and then negates the remaining elements, giving

```
[-3 -4 -5]
```

The sentence `-3` is in fact the program "Negate 3" and not an integer number with a value of negative 3. Therefore, and because operators associate to the right:

```
      -3+4
-7
```

Most Klong operators are pretty relaxed about the arguments they accept. For example, dropping 100 elements from a 5-element list is perfectly acceptable, yielding an empty list:

```
      100_[1 2 3 4 5]
[]
```

and taking 100 elements from the same 5-element list just returns a list of 100 elements, with the values of the elements "cycling" through the original list:

```
      100#[1 2 3 4 5]
[1 2 3 4 5 1 2 3 4 5 1 2 3 4 5 ···]
```

When Take reaches the end of the input vector, it just wraps around and takes the next element from its beginning again.

The Take and Drop operators can be combined to extract elements from any position inside of a vector. For instance:

```
      4#3_[1 2 3 4 5 6 7 8 9 10]
[4 5 6 7]
```

Combinations like **a#b_c** are called *idioms*. The idiom **a#b_c** (pronounced "a Take b Drop c") extracts a elements starting at position b from vector c. Learning array programming is very much about learning idioms. It is a bit like learning to read. At the beginning you read individual letters, then words, and then you begin to recognize the patterns of sentences while you are reading. It is the same in Klong, so there is no need to become frustrated when more complex programs are not immediately intelligible in the beginning. This is a perfectly normal point on the learning curve.

Higher-Dimensional Arrays

Unlike some other array languages, Klong distinguishes between scalars and vectors of a single element. For example, the integer object **1** is a scalar or an *atom*, while the object **[1]** is a vector of one element. The *Atom* (**@**) operator returns truth for atomic (non-vector) objects and falsity for vectors:

```
      @1
1
      @[]
1
      @[1]
0
```

All operators returning *truth values* deliver 1 when the condition checked by them applies and 0 otherwise. I.e. 1 and 0 are the canonical truth values in Klong. Other objects also have truth values, which will be discussed later (page 34).

The size of a vector is computed by the Size (#) operator:

```
      #[1 2 3 4 5]
5
```

Note that the **#** symbol is used for both the Size and the Take operator. To distinguish between the operators, Take is often written **a#b** (a-Take-b) while Size is written **#a** (Size-a). So the operators are distinguished by their *arity*, i.e. the number of their *arguments* or *operands*.

An operator taking a single argument, like **#a**, is called a monadic operator or *monad*, and a two-argument operator, like **a#b**, is called a dyadic operators or *dyad*.

What has been referred to as a "vector" so far is, more precisely speaking, a one-dimensional *array*, or *1-array*. The *shape* of a vector v is another vector containing the length of v; it is returned by the *Shape* (^) operator:

```
      ^[1 2 3 4 5]
[5]
```

The shape of any scalar object is zero:

```
      ^123
0
      ^[]
0
```

A vector of vectors of equal length is called a two-dimensional array, 2-array, or *matrix*. For example,

```
      [[1 2 3 4]
       [5 6 7 8]
       [9 0 1 2]]
```

is a vector containing three vectors of four elements, or a 3×4 matrix. Its shape is, unsurprisingly:

```
      ^[[1 2 3 4]
        [5 6 7 8]
        [9 0 1 2]]
[3 4]
```

The number of elements of the shape of an array indicates the

number of *dimensions* of that array; it is also called the *rank* of the array. For instance, the rank of a matrix, like the above, is

```
      #[3 4]
2
```

The rank of a vector is `#[n]` → `1` for any value of n, and the rank of a scalar a is `#^a` → `#0` → `0`. The notation $a \rightarrow b$ is not part of the Klong language; it is used in this text to denote that the expression a has the value b. Note that the Size operator, when applied to a number, returns the *magnitude* or *absolute value* of that number, e.g.: `#-1` → `1` or `#0` → `0`. Therefore the idiom `#^a` even works when a is a scalar.

Every matrix is a vector (of vectors), but not every vector of vectors is a matrix. For instance:

```
      ^[[1 2 3]
        [4 5 6]
        [7 8]  ]
[3]
```

Because the last element contained in the above vector has a different size than the other elements, the object is just a vector (or list) of three elements, but not a matrix. Only a vector of equally-sized vectors is a 2-array.

The concept of array rank extends past two dimensions. A matrix of equally sized vectors is a 3-array or 3-dimensional matrix:

```
      ^[[[111 112 113 114]
         [121 122 123 124]
         [131 132 133 134]]
        [[211 212 213 214]
         [221 222 223 224]
         [231 232 233 234]]]
[2 3 4]
```

For a 2-matrix to be a 3-matrix, *all elements* of the 2-matrix must be vectors of the same size or, in other words, all elements of the outermost vector must have the same shape. The *Each* (`'`) adverb, which will be explained later (page 22), can be used to extract the shape of each element of a vector:

```
      ⍴'[[0 0] [0 0] [0 0]]
[2 2 2]
      ⍴'[[[0 0 0] [0 0 0]]
         [[0 0 0] [0 0 0]]]
[[2 3] [2 3]]
      ⍴'[[[0 0] [0 0 0] [0 0]]
[2 3 2]
```

So the first two of the above objects are higher-dimensional arrays, but the last one is not, because not all elements of its shape are the same. The first of the above examples shows a 3×2 matrix, the second one a $2 \times 2 \times 3$ matrix, and the last one a vector. What shape does the following object have? What shapes do its elements have?

```
      [[[0 0 0]
        [0 0 0]
        [0 0 0]]
       [[0 0 0 0]
        [0 0 0 0]]]
```

Take a moment to think about it!

The object itself is a vector (shape `[2]`) and its elements are a 3×3 matrix and a 2×4 matrix.

An *n-array* (an array of rank n) of any rank and shape can be created from existing data objects using the Reshape (`:⍴`) operator. The Reshape operator is mostly a generalization of the Take operator that also accepts a shape instead of a count. With a left integer argument, it is almost like Take

```
      5:⍴[1 2 3]
[1 2 3 1 2]
```

but when $a = 0$ in `a:⍴b`, it returns b (of rank 0) instead of `[]`:

```
      0:⍴1
1
```

When the shape parameter a is a vector of integers, Reshape returns an array of the requested shape and fills it with elements taken from b:

```
      [3 4]:^[1 2 3]
[[1 2 3 1]
 [2 3 1 2]
 [3 1 2 3]]
```

Note that the resulting object may have a different shape than the one requested in the shape parameter a, if b is an array. For instance:

```
      0:^[1 2 3]
[1 2 3]
```

(requested shape 0, returned shape `[3]`) or

```
      [2 2]:^[[1 2 3]]
[[[1 2 3]
  [1 2 3]]
 [[1 2 3]
  [1 2 3]]]
```

(Requested: 2×2, returned: $2 \times 2 \times 3$.) This is because Reshape fetches elements only from b, but not from elements *contained* in b, i.e., the first element of `[[1 2 3]]` (the vector `[1 2 3]`) is fetched over and over again instead of cycling through the elements 1, 2, and 3. Adding the same vector over and over again adds an extra dimension to the resulting array.

Operations on Arrays

Earlier in this text, the Plus (**+**) operator was used to add scalars to vectors and vectors to each other. In fact the concept is much more general though, and extends to arrays of any rank. For example, the program

```
      1+[3 3 3]:^!27
```

first creates a 3-array (a 3-dimensional matrix) of the shape $3 \times 3 \times 3$ and then adds 1 to *every* element of it. The Enumerate operator `!a` is just shorthand for a list of integers from 0 to $a - 1$:

```
      [0 1 ··· a-1]
```

and the idiom `1+!n`, which creates an ascending list of integers from 1 to n, is very common in Klong programs.

When applying Plus to two arrays of different rank, it still adds their elements pairwise as long as they are compatible. For instance:

```
      [1 2 3]+[[4 5 6]
               [4 5 6]
               [4 5 6]]
[[5 6 7]
 [6 7 8]
 [7 8 9]]
```

The 1 of the left argument is added to the first row of the right-hand matrix argument, 2 is added to the second row, and 3 to the third one.

Note that this works only if the arguments of the operator have *compatible shapes*. The shapes of two arrays are compatible, if they have equal sizes up to the rank of the lower-ranking of the two arrays. For example, arrays of the shapes

```
[2 3]
[2 3 4 5]
```

are compatible, because their shapes share the prefix `[2 3]` *and* the length of the prefix, 2, is the rank of the lower-ranking one of the arrays. Here are some more examples of adding compatible arrays:

```
      [1 2 3]+[3]:^0
[1 2 3]

      [1 2 3]+[3 4]:^0
[[1 1 1 1]
 [2 2 2 2]
 [3 3 3 3]]

      [[1 2] [3 4] [5 6]]+[3 2 4]:^0
[[[1 1 1 1] [2 2 2 2]]
 [[3 3 3 3] [4 4 4 4]]
 [[5 5 5 5] [6 6 6 6]]]
```

An object of shape 0 is compatible with any other shape, as has been shown earlier. It will just be added to all elements of the other argument.

In particular, a non-atomic operand of an operator like Plus does not have to be a proper array as long as the other operand is an atom. It can very well be a nested list (a vector or vectors of varying length), as the following example illustrates:

```
      1+[2 [3] [4 5] [6 [7]] 8]
[3 [4] [5 6] [7 [8]] 9]
```

BTW: do not worry, if some operations on arrays of different shape look confusing at the moment. In most Klong programs, these operations are much more obvious than the more complicated examples above.

An operator that accepts an array and a scalar and combines the scalar with every single element in the array is called an *atomic operator*. Pretty much all *arithmetic operators*, like *Minus* (`-`), *Times* (`*`), *Divide* (`%`), *Negate* (`-`), *Min* (`&`), *Max* (`|`), and *Not* (`~`) are atomic. An atomic operator is called "atomic", because it applies to each atom contained in an array instead of the array itself. For instance, the Negate operator `-a` is applied to every atom in a list or array when applied to a list or array:

```
      -[3 3]:^1
[[-1 -1 -1]
 [-1 -1 -1]
 [-1 -1 -1]]
```

but the (non-atomic) *Reverse* operator `|a` affects its array argument in its entirety:

```
      |[1 2 3 4 5]
[5 4 3 2 1]
```

Array Transformations

Reversing a vector reverses the order of its elements. Because a matrix is a row-major vector of vectors, reversing it changes the order of its rows:

```
      |[3 3]:^1+!9
[[7 8 9]
 [4 5 6]
 [1 2 3]]
```

Remember: the idiom **1+!n** creates a vector of the numbers 1 through n.

The *Rotate* operator **a:+b** rotates the elements of the vector b to the right or left by **#a** (Magnitude-a) elements, depending on the sign of a:

```
      2:+[1 2 3 4 5]
[4 5 1 2 3]
      (-2):+[1 2 3 4 5]
[3 4 5 1 2]
```

When applied to a matrix, it rotates its rows up or down:

```
      1:+[3 3]:^1+!9
[[7 8 9]
 [1 2 3]
 [4 5 6]]
```

The columns of a matrix can be rotated using the idiom **:+:** (*Rotate-Each-Left*), e.g.:

```
      1:+:\[3 3]:^1+!9
[[3 1 2]
 [6 4 5]
 [9 7 8]]
```

The idiom uses the Each-Left (**:**) adverb, which will be discussed later (page 38).

The *Transpose* operator **+a** transposes a matrix:

```
      +[[1 2]
        [3 4]
        [5 6]]
[[1 3 5]
 [2 4 6]]
```

There are more array operations, but those will be explained by means of example programs in the remainder of this text.

Strings

A *string* can be thought of as a *vector* of *characters*. A character is a data object that represents a single letter of the ASCII

alphabet. A character literal is introduced by the sequence **`0c`**. Here are some characters:

```
0ca 0c[ 0c, 0c  0c0
```

Non-printing characters cannot be represented using **`0c`** notation, except for the blank, which is second-to-last in the above example. All characters can be created using the *Char* (**`:#`**) operator, for instance: **`:#10`** (newline), **`:#8`** (backspace), or **`:#27`** (escape).

All array operations can be applied to strings, as well. E.g.:

```
      "abc","def"
"abcdef"

      |"abc"
"cba"

      2:+"abcdef"
"efabcd"

      [3 3]:^"abcdefghi"
["abc" "def" "ghi"]
```

Note, though, that there are some subtle differences between strings and vectors of characters in Klong. First, the end of a vector of characters is the empty list:

```
      3_[0ca 0cb 0cc]
[]
```

and the end of a string is the empty string:

```
      3_"abc"
""
```

Second, a vector of characters is not structurally equal to a string, so they do not *match* (see page 56), even if they contain the same characters:

```
      "abc"~"abc"
1
      [0ca 0cb 0cc]~[0ca 0cb 0cc]
1
      "abc"~[0ca 0cb 0cc]
0
```

A vector of characters can be converted to a string using the following idiom:

```
      "",,/[0ca 0cb 0cc]
"abc"
```

The idiom uses the *Over* (`/`) adverb, which will be explained in the next section (page 24). The idiom acts as an identity operation when applied to a string so, when in doubt, applying it to a string will not do any harm:

```
      "",,/"abc"
"abc"
```

Joining the empty string catches the case where the string has only a single character, because `,/[`x`]` $\rightarrow x$ for any object x and `"",`$c \rightarrow$ `"`c`"` for any character c.

Adverbs

Operators (and functions) are also referred to as *verbs* in array programming and data objects as *nouns*. *Adverbs* modify verbs in order to do something slightly different. For example, the *List* verb `,a` puts its argument in a singleton list:

```
      ,!3
[[0 1 2]]
```

but when combined with the *Each* adverb `f'a` (f-Each-a), it turns *each element* of its argument into a list:

```
      ,'!3
[[0] [1] [2]]
```

The f in the notation `f'a` denotes a function, which in this case includes operators. For any *argument* a, `f'a` applies f to each element of a. Another example: the Size operator returns the size of a vector:

```
      #[[1] [2 3] [4 5 6]]
3
```

but `#'a` (Size-Each a) returns the size (or magnitude in case the element is a number) of each element of a:

```
      #'[[1] [2 3] [4 5 6]]
[1 2 3]
```

The Each adverb only makes sense when used to modify non-atomic verbs, because atomic verbs always behave like their modified counterparts when applied to arrays. For instance, there is no difference between Negate (`-`) and Negate-Each (`-'`) or Not (`~`) and Not-Each (`~'`), etc:

```
      -[1 2 3]
[-1 -2 -3]
      -'[1 2 3]
[-1 -2 -3]
```

in particular, there is no difference between these two expressions:

```
      -5
-5
      -'5
-5
```

When the argument of a verb modified with Each is a scalar, the verb is simply being applied to the scalar.

Like operators, adverbs have an *arity*: there are monadic adverbs and dyadic adverbs. The Each adverb is monadic, it takes a single argument to its right. The verb that is modified by it does not count as an argument.

In the case of Each, there is a dyadic variant called *Each-2*, which uses the same symbol as Each and is distinguished from its cousin by its context. A verb modified by Each appears in front of an argument a:

```
f'a
```

while a verb modified by Each-2 appears *between* its two arguments, a and b:

```
a f'b
```

Note that the arity of the verb f is a different thing. In the case of Each, f happens to be monadic and in the case of Each-2, f happens to be dyadic, but this is merely coincidence.

When applied to two vectors, Each-2 combines their elements pairwise using f:

```
      [1 2 3],'[4 5 6]
[[1 4] [2 5] [3 6]]
      (!4):+'4#,!4
[[0 1 2 3]
 [3 0 1 2]
 [2 3 0 1]
 [1 2 3 0]]
```

In case the second example looks a bit tough: **`4#,!4`** takes four elements from the singleton list containing the vector **`[0 1 2 3]`**, giving a matrix containing the same row four times. **`(!4):+'`** then rotates each element by a different amount.

Again, Klong is relaxed about argument shapes. When the two arguments of Each-2 have different sizes, the excess elements of the longer vector will be ignored:

```
      [1 2 3],'[4 5 6 7 8]
[[1 4] [2 5] [3 6]]
```

In particular, this means that

`a f'b` $=$ **`a f b`**

if either a or b is a scalar, i.e. the Each-2 adverb will be ignored in these cases.

The *Over* adverb **`f/a`** (f-Over-a), which has been mentioned earlier in this text, is a monadic adverb modifying a dyadic verb. The verb that is being modified is *folded over* the vector a. For instance, *Plus-Over* will sum up the elements of a vector:

```
      +/[1 2 3 4 5]
15
```

More, generally, **`f/a`** applies its verb f to the first two elements of a and then to the result and the third element, etc. Generally,

$$f/a = (((a_1\ f\ a_2)\ f\ a_3)\ \ldots)\ f\ a_n$$

where a indicates an n-element vector.

For example, `+/[1 2 3 4 5]` equals

```
(((1+2)+3)+4)+5
```

When a verb modified by Over is applied to an atom, the atom will be returned unchanged. This can be undesired when, for instance, summing up a vector of unknown size, because the result would be non-numeric for the empty vector:

```
      +/[]
[]
```

In these cases the dyadic variant *Over-Neutral* can be used, which supplies a *neutral element* to the operation:

```
      0+/[]
0
```

Formally, `a f/b` is equal to `f/a,b`.

There are lots of other adverbs in Klong. They will be explained by means of example programs in the remainder of this text.

Solving Problems

Idioms

When solving problems in Klong, there are three initial questions to ask:

- is there an operator that can do this?
- is there an operator-adverb combination that can do this?
- is there an idiom that can do this?

For instance: Transpose a matrix? Use the Transpose (**+**) operator. Concatenate the elements of a list? Use the *concatenate* idiom *Join-Over* (**,/**). Let us have a look at a more complex example, like the factorial function, which is defined here (slightly simplified by ignoring the case $0!$) as follows:

$$n! = \begin{cases} 1 & \text{if } n = 1 \\ n(n-1)! & \text{otherwise} \end{cases}$$

A different way to spell this out would be: “the product over the range of integers from 1 to n”:

$$n! = \prod_{i=1}^{n} i$$

which translates pretty directly to a Klong program:

- the integers from 1 to n is the idiom **1+!n**
- the product over a list of numbers is ***/** (Times-Over)

So the program that computes the factorial of a number $n \geq 1$ would be

```
*/1+!n
```

BTW, Klong has unbounded integers rather than integer arithmetic modulo some constant, so the above program can compute really big factorials:

```
   */1+!100
933262154439441526816992388562667004907159682643
816214685929638952175999932299156089414639761565
```

```
18286253697920827223758251185210916864000000000000000000000000
```

We have now created a new idiom by concatenating two existing ones. Combining idioms, nouns, and operators can solve quite a few problems. Let us have a look at another one. The average of a list L of numbers is

$$\mu = \frac{\sum L}{|L|}$$

i.e. the sum of all elements of L divided by the magnitude, or number of elements, of L. We already know that the magnitude of L is `#L` and the sum over its elements is `+/L`. The divide operator of Klong is called *Divide* (`%`).

So we need to sum up L and then divide by its magnitude:

```
+/L%#L
```

There is a little complication though, because all programs so far evaluated strictly from the right to the left, which might cause some trouble here. While the above idiom works fine, it will have the intermediate step

```
L%#L
```

which divides each element of L by `#L`, which may cause some loss of precision, because it does a lot of real number divisions, which are inherently imprecise.

So what we *should* do is to sum up the elements first and *then* divide by `#L`. Like in mathematics, this can be done by using parentheses:

```
(+/L)%#L
```

Another thing that is unsatisfactory about this solution is that is contains the list L twice, so when applying this idiom to a literal list, the list would have to be typed in twice. We can avoid this, though, by using a function.

Functions

A (single-argument) idiom is turned into a *function* by replacing each occurrence of its argument by the letter x and enclosing the

idiom in curly braces. So, for instance, the idiom

```
(+/L)%#L
```

becomes the function

```
{(+/x)%#x}
```

The letter x denotes the *independent variable* or *argument* of the function. In case you wonder how Klong knows that x is a function variable: the first variable of a function is always called x.

A function of a single variable is called a *monadic function* or *monad*, where the term "monad" is more general, because it also applies to monadic operators. Like an operator, a function is a *verb*, so it can be applied to arguments:

```
{(+/x)%#x}(1+!10)
5.5
```

Whenever an expression E in parentheses immediately follows a function, the function is *applied* to that expression: first the value of E is computed and the result is substituted for each x in the idiom of the function. Then the resulting idiom is evaluated and its value is returned by the function.

Like nouns and operations, function applications have values that can be used in expressions. For instance, we could compute the variance of a data set (represented by a list L) using the formula

$$var(L) = \frac{\sum_{i=1}^{n}(L_i)^2}{|L|} - (\mu_L)^2$$

In words: the sum of squares of L over `#L` minus the squared mean of L. This should translate pretty directly to Klong. Take a moment to do it yourself before you read ahead!

```
((+/L*L)%#L)-{(+/x)%#x}(L)^2
```

This program is a composition of the "mean" function from above and a new expression taken from the variance formula. However, this program is not quite readable, mostly because the "mean" function was inserted literally into the variance formula.

More complex functions are typically *bound* to variables, which

can then be used in the place of the function itself. The program

```
      mu::{(+/x)%#x}
:monad
```

binds the "mean" function to the variable *mu* using the *Define* (`::`) operator. The variable *mu* can then be used in the place of the literal function inside of the variance function, *var*:

```
      var::{((+/x*x)%#x)-mu(x)^2}
:monad
```

The *Power* (`^`) operator is used in the function to compute the square of the mean, because `mu(x)*mu(x)` would compute the mean twice. Using `x*x` versus `x^2` earlier in the function is merely a matter of taste; there is no advantage to either of them.

Note that the value of a definition is the value to which the variable is bound. Hence a chain of Define operators can be used to bind multiple variables to the same value or even different values:

```
      a::b::c::0
```

would bind all the variables a, b, and c to the value 0, and

```
      c::1+b::1+a::0
```

would set $a = 0$, $b = 1$, and $c = 2$.

A variable bound by Define outside of a function is called a *global variable*. After its definition it can be used anywhere in a program (and even before its definition in some cases; we will explore this possibility later).

After defining the functions shown above, the name *var* can be used to compute the variance of a list of numbers:

```
      var([3 1 4 1 5 9 2 7])
7.25
```

BTW, the definition of *mu* developed in this section is exactly the definition from the Klong statistics module.

We can now extend the list of questions to ask when solving a problem in Klong by another instance:

- is there a function that can do this?

Index