A List Toolkit

Dr. Peter Knaggs

September 15, 2018

The ordered list is an essential construct in programming. Indeed it is fundamental to Forth in that the dictionary is a list of word definitions, the search order, accessed via the ungainly words `GET-CURRENT` and `SET-CURRENT` is a simple list of word lists. Many systems access this as a stack, providing words such as `+ORDER` and `-ORDER`.

While it is probably easer to implement a list in Forth than in many other languages of its generation. Many modern languages (e.g., Perl, PHP, JavaScript, Java, C#, etc.) include an ordered list as a first-class variable type. Indeed the ordered list is fundamental to the functional language paradigm (languages such as SML and Haskall). This paper is an attempt to provide a tool kit for manipulating a list of generic elements.

Used correctly an ordered list is able to represent many different data structures:

Stack	Add to the end of a list and remove from the end the list.
Queue	Add to the end of a list and remove from the start of the list.
Array	Index into a list
Sequence	
Set	Check for membership before adding element.

The philosophy of this tool kit is to provide a set of words to manipulate a list of nodes, each with a single cell element. While this means the list is capable of holding basic data types (character, integer) is is also able to hold a reference to another data structure such as a data node or object. This allows the list to be generic. The word names used in the proposed tool kit have been chosen because they do not exist in current systems.

1 Implementation

The words have been defined in a way that does not require any particular implementation, thus allowing the developer to use the most appropriate implementation method for there system. For example:

Array	When a list is constructed a *capacity* parameter is given which provides the implementer with a minimum list capacity. Thus it is possible to implement this word set with a simple array. Although not so useful. A cyclic array would be a better fit, allowing manipulation of both the start and and end of the list.

Linked List	A single (or doubly) linked list is the more traditional implementation technique used by most Forth programmers. If the list constructor ignores the *capacity* parameter, a linked list implementation is more than viable.
Array List	An array of *capacity* elements held in user memory. When an element is added to the list it will check if the list has space for the element. If not then it will allocate a new array of *capacity* $\times$ *ratio* elements, moving the exiting array into the new array, and returning the old array back into the user memory.
Bucket List	A linked list of buckets (an array of *capacity* elements) held in user memory. When an element is added to the list, it will check the list has space for the element. If not then it will allocation another bucket which is linked to the exiting list.

2 Indexing

When indexing into the list an index parameter (n) is used. This allows indexing into the list from the start of the list (when n is positive) or from the end of the list (when n is negative). Thus $n = -1$ will provide access to the last element of the list, while $n = 0$ will provide access to the first element in the list.

When the index is beyond the range of the list, an exception is thrown. For example, if a list has 10 elements, an index of 10 (or -11) will be beyond the range of the list. As the standard does not provide an "out of memory", "out of range" or "bad index" exception, the "result out of range" (-11) exception has been abused to indicate the index is out of range.

3 Iterator

Those languages that provide for a list also provide a mechanism to iterate over a list of elements in the general form:

for (*variable* **in** *list*) { *body* }

where the *body* of the loop is executed once for each element of the *list* and *variable* is given the current element of the list on each iteration.

A Forth variant of this idea is proposed, where the word `FOREACH` takes a list and creates an implementation dependent data structure *iter* that is used to control the iteration loop. The word `I` (from the iterator word set) places the current list element on the data stack, while the `NEXT` word (from iterator word set) ends the iteration loop.

Therefore we could implement a loop to sum the contents of a list of n as:

```
: SUM ( list -- n )
  0 SWAP FOREACH I + NEXT
;
```

If *iter* where to contain an *xt* of a word that determines the next element of the iteration, `NEXT` could simply call that *xt*. Replacing the iteration constructor word (`FOREACH`) with a word more appropriate to the iteration type that builds the corresponding *iter* (complete with relevant *xt*). This would provide the ability to iterate over any type of data structure, such as a word list, the characters in a string, the lines in a file, rows in a database and so on.

4 List toolkit

`CREATE-LIST` (*n* −− *list*) TOOLS

Create a new list capable of holding a minimum of *n* single cell elements, returning a list identifier *list*.

Comments:
This is a basic list constructor that builds a new anonymous list. n is the capacity *parameter. It also introduces the opaque type list. Depending on the implementation this could be an xt or a variable pointing to the first element in this list.*

Formally: $list = \langle\ \rangle$

`LIST:` (*n* "$\langle ccc \rangle$" −−) "list-colon" TOOLS

Crate a new list $\langle ccc \rangle$ capable of holding a minimum of *n* elements.

Implementation:
```
: LIST: ( n "<ccc>" -- ) CREATE-LIST CONSTANT ;
```

Comments:
This is the named list constructor.

Separating out the named list constructor and the anonymous list constructor allows the developer to combine the anonymous constructor with a system specific method of creating a definition, thus allowing developers to build a list factory.

`LIST+` (*x list* −−) "list plus" TOOLS

Add the element *x* to the end of the *list*.

If there is insufficient space in the list, the system may extend the list to allow the new element or throw a -11 (out of range) exception.

Comments:
This is frequently known as `push`, `add` *or* `append` *in other languages.*

Formally: $list' = list \frown x$

`LIST-` (*list* −− *x*) "list minus" TOOLS

Remove the last element the *list* placing on the stack (*x*).

If there are no elements in the list a -11 (out of range) exception is thrown.

Comments:
This is frequently known as `pop` *or* `remove` *in other languages.*

Formally: $list = list' \frown x$

`+LIST` (*x list* −−) "plus list" TOOLS

Add the element *x* to the start of the *list*.

If there is insufficient space in the list, the system may extend the list to allow the new element or throw a -11 (out of range) exception.

Comments:
This is frequently known as `insert` *or* `unshift` *in other languages.*

Formally: $list' = x \frown list$

`-LIST` (*list* −− *x*) "minus list" TOOLS

Remove the first element of the *list* placing it on the stack (*x*).

If there are no elements in the list a -11 (out of range) exception is thrown.

Comments:
This is frequently known as `shift` *or* `remove` *in other languages.*

Formally: $list = x \frown list'$

`CONCAT` ($list_1$ $list_2$ −−) TOOLS

Append the content of $list_1$ to $list_2$ such that $list_2$ contains all the elements in $list_2$ followed by all of the elements in $list_1$.

If there is insufficient space in $list_2$, the system may extend the list to allow for the new elements or throw a -11 (out of range) exception.

Comments:
This is frequently known as `concatenate`, `append` *or* `join` *in other languages.*

Formally: $list_2' = list_2 \frown list_1$

`>LIST` (*x n list* – –) "to-list" TOOLS

Insert the element *x* into the *list* at position *n*, relative to the start of the list. If *n* is negative, the position is relative to the end of the list. If *n* is beyond the end of the list a -11 (out of range) exception is thrown.

When *n* is 0, this is the same as `+LIST`. When *n* is -1, this is the same as `LIST+`.

Comments:
This is frequently known as `insert` *or* `insertAt` *in other languages.*

Formally: $list' = list_{0..(n-1)} \frown x \frown list_{n..\#list}$

`LIST>` (*n list* – – *x*) "list-from" TOOLS

Remove element *n* from the *list* returning the removed element (*x*). *n* is relative to the start of the list. If *n* is negative, the position is relative to the end of the list. If *n* is beyond the end of the list a -11 (out of range) exceptionis thrown.

When *n* is 0, this is the same as `-LIST`. When *n* is -1, this is the same as `LIST-`.

Comments:
This is frequently known as `remove` *or* `removeAt` *in other languages.*

Formally: $list' = list_{0..(n-1)} \frown x \frown list_{(n+1)..\#list}$

`/LIST` (*list* – – *u*) "slash list" TOOLS

Return the number of elements (*u*) in *list*.

Comments:
This is known as `count` *or* `length` *or simply* `#` *in other languages.*

Formally: $u = \#list$

`#LIST` (*x list* – – *u*) "number list" TOOLS

Return the number of times (*u*) the element *x* appears in the *list*. If *x* does not appear in *list*, *u* will be 0.

Comments:
This is known as `count` *in other languages.*

Formally: $u = \#\{\forall e \in list \,|\, e = x\}$

`?LIST` (*x n list* -- *u* | *-1*) "query list" TOOLS

Return the position (*u*) of the first occurrence of *x* in the *list*, starting the search at position *n*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. Return -1 if *x* is not found or if the start position (*n*) is beyond the range of the *list*. The first element in the list is at position 0 and the last element is at position -1. The result (*u*) is always given relative to the start of the list.

Rationale:

One can check for membership of a list by comparing the result to -1.

```
: in ( x list -- flag )
    0 SWAP ?LIST 0< 0=
;
```

Comments:

This is known as `indexOf` *in other languages.*

Formally: $(list_u = x \wedge x \notin list_{n..u-1}) \vee (u = -1 \wedge x \notin list)$

`LIST@` (*n list* -- *x*) "list fetch" TOOLS

Return the element (*x*) at position *n* of *list*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. The first element in the list is at position 0 and the last element is at position -1. If *n* is beyond the range of the *list* a -11 (out of range) exception is thrown.

Comments:

Other languages use the index operator `[]` *to index into a list.*

Formally: $x = list_n$

`LIST!` (*x n list* --) "list store" TOOLS

Set the element at position *n* of *list* to be the value *x*. *n* is relative to the start of the list if positive and relative to the end of the list if negative. The first element in the list is at position 0 and the last element is at position -1. If *n* is beyond the range of the *list* a -11 (out of range) exception is thrown.

Comments:

Other languages use the index operator `[]` *to index into a list.*

Formally: $x = list'_n$

`TRAVERSE-LIST` (*list xt(x * i x – x * j)* ––) TOOLS

The *xt* is executed once for each element in the *list*, with each element being presented to the *xt* on the top of the stack (*x*).

The *xt* may change the data stack during execution.

Rationale:

For example, it is possible to sum a list of *n* using the following:

```
: SUM ( list -- n )
    0 SWAP ['] + TRAVERSE-LIST
;
```

Comments:

This is known as `map` *or* `each` *in other languages.*

Formally: $\forall x \in list \bullet xt(x)$

`FOREACH` TOOLS

Interpretation:

The interpretation semantics are undefined.

Compilation: (C: –– *dest*)

Put the next location for a transfer of control (*dest*) onto the control flow stack. Append the run-time semantics given below to the current definition.

Run-time: (*list* ––) (R: –– *iter*)

Initialise an iteration over the *list*, placing the iteration control (*iter*) onto the return stack.

Rationale:

For example, it is possible to sum a list of *n* using the following:

```
: SUM ( list -- n )
    0 SWAP FOREACH I + NEXT
;
```

Comments:

The opaque type iter *will vary depending on the implementation, but will probably contain at least the list identifier and the current position within the list. This will alter the search order such that the iteration version of the* `I` *and* `NEXT` *words are found before the Core versions.*

`I` (–– *x*)(R: *iter* –– *iter*) TOOLS

Use the iteration control (*iter*) to return the current value (*x*) in the list iteration started by `FOREACH`.

Comments:

iter *is used to obtain the current value in the iteration, but is not altered.*

`NEXT` TOOLS

Interpretation:
: The interpretation semantics are undefined.

Compilation: (C: *dest* --)
: Append the run-time semantics given below to the current definition, resolve the backward reference *dest*.

Run-time: (R: *iter* --)
: Use the iteration control (*iter*) to determine the next element in the iteration. If there is another element in the iteration, update the iteration control and continute execution at the location specified by *dest*. If there are no more entries in the list, remove the iteration control (*iter*) from the return stack and continue execution with the next instruction.

Comments:
: *This should remove the iteration word list from the search order.*