the NOVICE.4TH and LIST.4TH code
http://www.forth.org/novice.html

Hugh Aguilar

September 7, 2010

Abstract

I will describe some of the main features of the novice.4th and list.4th programs, which are written in ANS-Forth. Forth has traditionally offered a Catch-22 — you can't write applications until you have basic tools available, but you can't implement these tools until you have acquired some experience writing applications. My library is intended to allow the novice to acquire this experience using by my code and not necessarily understanding how it works. The novice package is like sausage — you don't need to know how it is made to eat it. The reader of this article however, is assumed to be a fairly advanced Forth programmer who is deciding if he should recommend my novice package to novices or not.

1 novice.4th

There are a lot of useful functions and tools in the `novice.4th` file. For boosting speed, we have `MACRO:`, which is like colon except that it compiles inline. Also, there is `{`, which is used for defining local variables in colon words. This fixes the problem of `LOCALS|` defining its locals backwards. I think my `{` is compatible with the John Hopkins format, although I've never seen the John Hopkins code so I can't be sure. There are many other tools that are useful for application writing, but which won't be covered in this document.

1.1 :NAME and friends

By far, the most important function in the `novice.4th` file is this:

```
: :name ( str wid -- )      \ like colon except takes its name as a parameter
```

I tried to talk the Forth-200x committee into making `:NAME` standard, but they refused (they pretty much categorically refuse every suggestion made by an outsider). To a large extent, my original motivation for writing the novice package was to popularize `:NAME` so that it would eventually attain critical mass and get accepted into the standard. The Forth-200x committee's primary

(sole) concern is supporting legacy code, so they are a lot more interested in the popularity of a feature rather than its technical merits. This is also why ANS-Forth contains features that are incompatible with each other (>R etc., and local variables) — both features were popular, and the fact that the features don't work in conjunction was not an issue. This is also why there is so little innovation — a feature can't become popular unless it can first be written in standard Forth so everybody can use it, but if it can be written in standard Forth then there is no point in adding it to the standard — it is a Catch-22!

In addition to :NAME, I also have some similar definers trivially derived from :NAME:

```
: :2name ( prefix-str suffix-str wid -- )       \ used for suffixing or prefixing names

: :3name ( prefix-str mid-str suffix-str wid -- )

: :name! ( str wid -- )                         \ like :NAME but with a ! suffix

: :name@ ( str wid -- )                         \ like :NAME but with a @ suffix
```

1.2 DEFER and VECTOR

:NAME effectively obsoletes CREATE DOES> for writing defining words. The code generated by :NAME is *always* faster executing than the code generated by CREATE DOES>. Let us look at a simple example of both a CREATE DOES> function (DEFER) and a :NAME function (VECTOR), found in the novice.4th file:

```
: xxx
    true abort" *** uninitialized vector ***" ;

: defer ( -- )          \ stream: name
    create  ['] xxx  ,
    does>
        @ execute ;

: is ( xt -- )          \ stream: name
    state @ if
        postpone ['] 
        postpone >body
        !,
    else
        '  >body !  then ;
immediate

: <vector> { name | xt -- }
    here  ['] xxx ,  to xt
    c" is-" name  get-current :2name                    \ runtime: xt --
        xt lit, !, ;,
    name get-current :name                              \ runtime: --   \runtime runtime: i*x -- j*x
        state@, if,     xt lit,  postpone lit,  postpone @,  postpone execute,
        else,           xt lit,  @,  execute,                                       then,  ;,
        immediate
    ;

: vector ( -- )
    bl word hstr  dup >r  <vector>  r> dealloc ;
```

DEFER and IS should be familiar to every Forth programmer; these are used primarily for mutual-recursion, but also for any function whose definition may need to be changed at some later date without recompiling the original source-code. DEFER defines the "deferred" (a.k.a. "vectored") word. IS is then used to associate an xt ("execution token") of a word to that deferred word. My

definition of DEFER and IS is pretty typical. The reader will note that my IS is state-smart, which is considered to be a *Bad Thing* in Forth. In this case however, state-smartness is not a big deal because the possibility that somebody would tick IS is beyond unlikely.

The only new thing (in IS) is this function:

```
: !, ( -- )     \ runtime: val adr --
   postpone ! ;
```

I have a *lot* of functions like !, in the novice.4th file; they help to make colon-word generators much more readable — you don't have so many explicit POSTPONE invocations in your functions, which can become rather tedious.

DEFER is a CREATE DOES> definer (rather primitive) — let us now consider <VECTOR> that is a :NAME definer (more advanced). This definer generates two colon words. If the name given to <VECTOR> is XXX, the two colon words will be IS-XXX and XXX. The IS-XXX function is just a typical colon word; it has the xt compiled within it as a literal, and it stores this value into a dictionary variable. The XXX function is more complicated. It is an immediate colon word; it generates code that will fetch the xt value and execute it.

I generally always write a low-level definer with pointy brackets that takes the name as a cstring on the stack (such as <VECTOR> above). And I also have a high-leve definer without any brackets that pulls the name out of the input stream (such as VECTOR above). Note that HSTR converts a cstring into a new cstring stored in the heap. DEALLOC is like FREE but with more error-checking.

A pretty good argument can be made that it is a mistake for me to make my vectored word a state-smart immediate function, because some luckless soul will tick it and the result will be chaos. Maybe so! This has never happened to me, but it is possible and (by some accounts) likely. I just made my vectored words state-smart immediate functions for the sake of efficiency. Almost all of the applications that I write are *heavily* recursive. The four example programs that I provide (symtab.4th, LC53.4th, LowDraw.4th and slide-rule.4th) are all recursive at their crux. For me, efficiency in mutual-recursion is pretty important. If you are worried about the deferred word being state-smart however, it is easy to modify <VECTOR> to generate a typical colon word rather than an immediate word at some cost in speed (because it will be a function rather than inline-code) — this simplification will be left as an exercise for the reader.

1.3 down with CREATE DOES> definers!

There are *three* reasons why I don't like CREATE DOES> definers:

- CREATE DOES> definers are inherently inefficient. The compiler *can't* know if the data comma'd in after the CREATE is mutable or not, so it has to assume the worst-case scenario that the data is mutable (by an external function such as IS). The compiler can't compile the datum as a literal in the DOES> code, but rather it must compile a fetch of the data in the DOES> code. By comparison, I always know if my data is going to

be immutable (it usually is) or mutable (such as the example abovee). If it is immutable, I can compile a literal, which is much faster than a fetch from memory. Because of this, `:NAME` definers *always* generate more efficient code than `CREATE DOES>` definers — this is not a compiler issue, but is true of all ANS-Forth compilers. Some compilers (VFX) allow the programmer to specify with declarations that `CREATE` data is immutable, but these declarations are not ANS-Forth compliant — by comparison, my `:NAME` *is* ANS-Forth compliant, although it is much different than what most Forth programmers consider to be a "typical" definer.

- `CREATE DOES>` definers only allow for *one* action to be associated with the defined word. In `DEFER` above, this is the execution of the xt. Other actions must be written as external words using `>BODY` internally. In `DEFER` above, this is the `IS` "function" (not really a function in the theoretical sense, because it modifies global data). The `VECTOR` definer is a pretty simple example, as it has only two actions. The array definers (`1ARRAY` etc., and `ARY`) and the `WBUF` definer in `novice.4th` are better examples of how multiple actions can be associated with a data type.

- `CREATE DOES>` definers can be hard to read when the programmer manually accesses the data comma'd in after the `CREATE` (as done in "Starting Forth"). A common solution to this problem is to define a record externally to the `CREATE DOES>` definer. In the `DOES>` code, the base-adr is pushed onto the return stack and `R@` is then used throughout, followed by the field name, to access each field. This is pretty readable, but it also makes for bulky and inefficient code. My `:NAME` definers generate a lot more efficient code, and they are yet quite readable because the fields are given names (local variables) in the definer. In `VECTOR`, the `xt` local is such a field. Better examples of this readability are the array definers and `WBUF`, which have a lot more fields.

All of that meta-programming (with `POSTPONE`) can sometimes become complicated. A good technique for writing defining words is to first write the code as a single static instance. After the code is debugged, it can be rewritten as a defining word. For example, in `novice.4th` I have a definer called `WBUF` that defines ring-buffers. I also have a file `wbuf.4th` that contains a single static ring-buffer. This was a prototype of the more general `WBUF` definer. The novice can see the two-stage process used in developing a defining word.

2 list.4th

I programmed in Factor for a while. For various reasons I didn't like it, and I went back to Forth. One thing that did impress me about Factor however, was the way that it provides "sequences" as a fundamental data structure. Almost any program can use sequences for holding a wide variety of kinds of data. I decided that Forth needed something similar, which is why I wrote the `list.4th` package. Factor's sequences are actually stored as arrays internally, whereas I am using linked lists, but the general idea is the same. The Lisp language (that Factor is derived from) uses lists as their fundamental data structure, and

my aim was to put Forth in the same category. The `list.txt` file provides documentation for how this code is intended to be used. A brief overview will be provided here however.

2.1 the SEQ and DECOMMA data types

A list is defined like this:

```
list
    w field .line
constant seq

: init-seq ( str node -- node )
    init-list >r
    hstr    r@ .line !
    r> ;

: new-seq ( str -- node )
    seq alloc
    init-seq ;
```

Here, `SEQ` is a child data-type of `LIST`. There should always be an `INIT-xxx` and `NEW-xxx` function for the list. You should not do the initialization of the record in the `NEW-xxx` function. You need to have an `INIT-xxx` function because there might be a child of this data type, and its `INIT-xxx` function will need to call this `INIT-xxx` function. For example:

```
seq
    w field .head       \ a SEQ of the .LINE string split on comma deliminators
constant decomma

: init-decomma ( str node -- node )
    init-seq >r
    r@ .line @ count split  r@ .head !
    r> ;

: new-decomma ( str -- node )
    decomma alloc
    init-decomma ;
```

Here we have a child data-type called `DECOMMA` whose parent is `SEQ`. This data type will have a `.LINE` field that it inherits from `SEQ`, and will also have a `.HEAD` field of its own. In this case, `.HEAD` is a `SEQ` list. The `SPLIT` function splits a string on comma deliminators to form a `SEQ` list.

This style of pseudo-OOP I learned in Factor — the business of making the initialization a separate function from the creation was non-obvious (to me), but it is the crux of the system. My Forth code does have some weaknesses. The most glaring being that all of the field names are globally defined. A field such as `.HEAD` isn't intrinsically associated with the `DECOMMA` record. If another record also has a `.HEAD` field (pretty likely considering what a generic name `.HEAD` is), the result will be chaos. Still though, my `list.4th` is a step in the right direction. My slide-rule program (that generates gcode for a CNC milling machine and also PostScript for the faces of a slide-rule) was a crucible for my list package. That program came out pretty well IMHO, so I think the `list.4th` code has been proven to be capable of being the basis for a large application.

2.2 CLONE-LIST and CONCRETE-LIST

Lists are always created on the heap. We also have these functions that are used for making copies of lists:

```
: clone-list ( head -- new-head )

: concrete-list ( head -- new-head )
```

`CLONE-LIST` makes a copy of an existing list, and the new list is on the heap. `CONCRETE-LIST` is similar, except that the new list is in the dictionary. This is very useful for allowing the programmer to generate lists at compile-time rather than run-time. My `FREE` can distinguish between memory that was allocated on the heap or in the dictionary. If it is dictionary memory, then `FREE` does nothing. This allows the programmer to free up lists after they are no longer needed (to prevent memory leaks) without needing to know if those lists were originally generated at compile-time and put in the dictionary, or at run-time and put on the heap.

2.3 list traversal

Another worthy feature of Factor is quotations. These are very useful in certain circumstances (although they are somewhat overused in Factor what with `DIP` etc., imho). Quotations (and generators for the same reason) will never be introduced into Forth-200x because they require the ability to take a snapshot of the state of a function (especially the local variable stack), which is not possible in ANS-Forth. As I said before: anything that can't be written in ANS-Forth can't become popular, and therefore will never become "legacy code," which is the only criteria for gaining support in Forth-200x.

Although we can't have quotations in Forth, my `list.4th` package does have "touchers" (somewhat of an ugly name, but I couldn't think of anything else). These are functions that are called via `EXECUTE` for every node in a list.

```
: each ( i*x head 'toucher -- j*x )             \ toucher: i*x node -- j*x
```

Here is an example for the `DECOMMA` lists:

```
: <count-C> ( count node -- new-count )    \ increment count if the first field contains a capital C
    .head @ .line @ count  s" C"  search nip nip  if  1+  then ;

: count-C ( head -- count )
    0
    swap  ['] <count-C> each ;
```

The reader will note that the toucher can access data on the stack underneath the node (the `i*x` data). Our `<count-C>` toucher accesses the count value, which it increments. The only rule here is that the toucher can't remove or add data to the stack — and even this rule can be broken in certain cases (see `collect-C` in the `list.4th` file). The ability for the toucher to access data underneath the node is possible in Forth because Forth is untyped. This would not be possible in C/C++ because the toucher would have to have a prototype that exactly describes its input parameters (just the node) and would not allow touchers

to be written that accessed other input parameters. In C, untyped functions can be faked up by passing the underneath parameters inside of a struct, and declaring a void pointer to this struct as the parameter. This would be horribly ugly, and also inefficient — one of many reasons for avoiding C/C++ like the plague that it is.

Here is a slightly more complicated example:

```
: <purge-C> ( head node -- new-head )      \ remove any node in which the first field contains a capital C
    dup .head @ .line @ count  s" C"  search nip nip  if        remove  <kill-decomma>
    else                                                       drop                          then ;

: purge-C ( head -- new-head )
    dup  ['] <purge-C> each ;
```

Here we are filtering out the list. I use `REMOVE` to remove the pattern-matched node from the list, and `<KILL-DECOMMA>` to deallocate it so that it doesn't become a memory leak. I can remove *any* node from the list, including the head or the tail. This works because I have simple linked lists. Other implementations of linked lists (the grand-eloquently named "Forth Foundation Library" being an example) use a handle for improved efficiency. The handle is a record that contains pointers to the head and tail, so it is not necessary to sequentially search the linked list to find the tail node. The problem with using a handle however, is that you can't arbitrarily remove nodes from the list as I am doing in `purge-C`. If you remove the head or the tail node, your handle will no longer be up-to-date and the result will be chaos. My simple lists are somewhat less efficient (especially my `TAIL` function), but they are more robust. I don't think the use of a handle is a very good idea. If I do upgrade my list package in the future, I will upgrade it to a doubly-linked circular list, which has all of the same robustness as my current simple implementation.

I also have these words for traversing lists:

```
macro: each[                                  \ toucher: i*x node -- j*x
    begin  dup while
        dup .fore @ >r ;

macro: ]each
        r> repeat drop ;
```

These can be used like this:

```
: clone-node ( node -- new-node )             \ returns a list with only one node in it
    dup allocation >r
    r@ alloc  tuck                            \ -- new-node node new-node     \r: -- size
    r> cmove>  init-list ;

: clone-list ( head -- new-head )
    nil  swap each[  clone-node link  ]each ;
```

For the most part, I recommend the use of `EACH` rather than `EACH[` and `]EACH`. The reason is that `EACH[` and `]EACH` use the return stack internally, and this isn't obvious to the user. If the user writes code inside of `EACH[` and `]EACH` that access local variables in the function, the result will be chaos — and I'll get blamed. With `EACH` on the other hand, the user wouldn't expect his toucher to have access to the local variables in the parent function and wouldn't try to access them. The best solution would be to have quotations, in which case the

quotations could access local variables in the function where they are born — but that is not feasable in ANS-Forth.

In some cases, EACH[and]EACH are the best solution. In CLONE-LIST above, be best way to factor the code is to have a CLONE-NODE function. This means that our toucher has to include both CLONE-NODE and LINK. Rather than factor this out into a function of their own, it is easier and more readable to just write them inline inside of EACH[and]EACH.

Another advantage of EACH[and]EACH over EACH, is better efficiency. The user is welcome to use EACH[and]EACH for this reason, just be careful that you don't get into trouble trying to access local variables in your toucher code.

In addition to EACH, I also have traversers that are used for locating a pattern-matched node:

```
: find-node ( i*x head 'toucher -- j*x node|false )            \ toucher: i*x node -- j*x flag

: find-prior ( i*x head 'toucher -- j*x -1|node|false )        \ toucher: i*x node -- j*x flag
```

2.4 memory allocation words

All of the memory allocation words have been rewritten, with a few new ones thrown in:

```
: allocate ( n -- adr ior )                    \ the ior is false if successful

: concrete-alloc ( n -- adr )

macro: concrete-allocate ( n -- adr ior )      \ the ior is false if successful (it always is)

macro: <allocation> ( adr -- size )            \ signed \ negative means it is a concrete node
    w - @ ;

macro: allocation ( adr -- size )              \ unsigned
    <allocation> abs ;

: resize ( old-adr n -- new-adr ior )          \ the ior is false if successful

: free ( adr -- ior )                          \ the ior is false if successful
```

ALLOCATE and CONCRETE-ALLOCATE are used for allocating memory. ALLOCATE does this on the heap, and CONCRETE-ALLOCATE in the dictionary. They store the size of the allocated memory block in the cell just ahead of the memory block. ALLOCATE stores this as a positive number, and CONCRETE-ALLOCATE as a negative. ALLOCATION uses this information to return the size of the allocated memory block. This is necessary for CLONE-NODE and CONCRETE-NODE (used by CLONE-LIST and CONCRETE-LIST) so they know how big to make the new node and how much memory to CMOVE> over there to effect the copy. RESIZE also uses this size information similarly. FREE just checks to make sure that the memory block is in the heap (a positive size) rather than the dictionary (a negative size) so it doesn't try to free dictionary memory, which would crash the system.

I also tried to talk the Forth-200x committee into making CONCRETE-ALLOCATE, ALLOCATE, ALLOCATION, RESIZE and FREE part of the standard, but this idea was also quashed. Once again, I put these functions in my novice package in an effort to build up enough critical mass to be accepted into the standard.