# FROM PASCAL TO FORTH #

Leonard Morgenstern

Presented at Asilomar, November 1989. Slightly modified.

**Table of Contents**

## ABSTRACT#

Fundamental differences in viewpoint between Pascal and Forth are more important than certain obvious dissimilarities in syntax. Pascal is conventional, with a mode of thinking that derives from algebraic notation. Forth is "organic," growing out of the processor and operating system, and permits the stepwise development of complex programming features.

## INTRODUCTION#

The stimulus for writing this article was my recent purchase of Sedgwick's Algorithms with the aim of translating them from the authors Pascal into Forth. I quickly discovered that certain obvious and large-seeming differences between the two languages are mere syntactical variations whose effect is one of inconvenience, while less obvious features that derive from a divergence in viewpoint present more of a problem.

An example is array limits: Forth conventionally starts with 0, whereas Sedgwick usually uses 1. Translation is straightforward, although it may not always be easy to decide whether a 1 is the index of the first element of an array and can be changed to 0; or whether it is the quantity 1 that must not be altered. Less evident are profound dissimilarities between Forth and Pascal arrays, discussed in detail below. Similar comments apply to reverse Polish Notation, do loop limits, and Forths stacks.

Pascal, C, Fortran, and BASIC, are conventional languages one might say that they are all one language, ultimately derived from algebraic notation. C in particular was designed to conform to an abstract plan. Forth is genuinely different. It is organic, in the sense that it grows out of its environment (the microprocessor and operating system) in response to the practical and aesthetic needs of the programmer. Forth can be programmed so that it becomes a language specially written for your application. This fact, of which all Forth programmers are aware, even if subliminally, is a prime source of difficulty in establishing a Forth standard. Among major differences between Forth and Pascal are the following:

1. Traditional languages have three levels: microprocessor, language, and application. These distinctions disappear in Forth. Although it is convenient to refer to words written in assembler as primitive, or low level, there is no real difference between them and high level words, once they have been defined. A new word becomes part of Forth, the equal of all other words, and the user may ignore its supposed level.

2. Flexibility: Defining words can create new classes of words as needed. An example of how rather simple Forth can provide for actions difficult or impossible in Pascal will be given.

3. Structure: Pascals structures are rigid. Once incorporated in the language, they are violated only with great difficulty.

4. Abstractness: As noted, Pascal is designed to an abstract plan, whereas Forth's features are developed by the programmer to resolve practical necessities (although it can be made as abstract as one likes.)

## STRUCTURE IN FORTH AND PASCAL#

A good example of the philosophical differences between Forth and Pascal is programming structure, the canons of which were established by Wirth, Dykstra and others in response to the spaghetti code that resulted from the indiscriminate use of goto. The success of the scheme is its justification: individual programmers could now write segments independently and have them fit together on the first try. But, structure can be a hindrance. Even though it can be proved mathematically that a few simple structures can do anything, it sometimes requires more effort than it is worth merely to support a principle, when simple controlled violations are easily defined in Forth. The most familiar instance is breaking out of a DO loop with LEAVE.

Exception handling is an area where strict conformity to the canons of structure presents a formidable challenge, whereas unstructured methods are easy to apply. Error traps are, in effect, gotos (more accurately, go_back_tos) and are inherently unstructured. ABORT is the ultimate example, exiting the program and restarting Forth, but less drastic action can be achieved by various published schemes.

But Pascals avoidance of goto is not merely aesthetic. Pascals stack is invisible to the programmer, and to jump out of a structure may have unpredictable side effects. For this reason, many Pascals do not even have goto, and those that do often warn users (correctly) not to use it. Forths stacks are always open to view, and undesirable consequences can be anticipated and evaded.

## USING AND AVOIDING THE STACK#

In Pascal, every quantity has a name, and memory must be allocated for it; thus, a := b+c uses three names. Temporaries may also be required during the evaluation of complex expressions. In Forth, the parameter stack is commonly the best place to put quantitiesin the above example, if b and c are already there, + will add them; no variables being required; memory is saved and speed is improved. The return stack is available for temporaries. Beginners in Forth do not employ this approach enough, and overuse named variables; but experienced programmers may go the other way; superfluous ROT, SWAP, R> and >R commands are a signal that this has happened. Local variables, now provided by most Forths, alleviate the situation.

Therefore, when translating a complicated algorithm to Forth, do not be premature in using the stack. Use ancillary variables and locals liberally in the first draft. In later versions, eliminate the unnecessary ones.

## ARRAYS#

There is no standard array in Forth. This used to annoy me, until I read a comment by Charles Moore to the effect that it is so easy to write an array to suit your needs, that a standard is pointless. In Pascal, an array returns its contained quantity, and can be said to be VALUE-like, but the usual array in Forth returns an address and can be said to be VARIABLE-like. As a result, it is often possible to use primitives such as CMOVE to do fast shifting, instead of looping slowly with an index. (Some Pascals permit this.)

The idea of a Forth array is easily extended. DEFER-like arrays (execution vectors) are familiar; VALUE-like arrays and string arrays are easily written. The contents of any kind of array can be located in the dictionary, in a file (virtual memory), or in extended memory. Mixed arrays in Pascal (arrays of records) are easily duplicated In Forth, and can include DEFER-like elements, not possible in Pascal The principle will be illustrated below.

## MIXED ARRAYS (ARRAYS OF RECORDS)#

In Pascal, an array can have elements of any desired type, including mixed types (array of records), and any element or part of one can be accessed. The same can be done in Forth, with the added power that a mixed array can incorporate actions. As an example, verbs will be defined for an adventure game. The reader is invited to try to duplicate each feature in Pascal step by step.

1. Defining a Forth array. The following simple definer sets up an array composed of elements of any specified length. The array is variable-like, returning an address, and is situated in the dictionary. Range checking and multidimensionality can be added if desired.

```
: -BYTE-ARRAY ( n l -- ) ( i -- adr)
    CREATE DUP , *  ALLOT  DOES> DUP @ ROT * + 2+ ;
```

2. Defining offsets. In Pascal, it is possible to point to any desired part of an array element. For example, if the elements of an array a consist of an identification number and a string name, one can define the record so that a.number[2] and a.name[2] point to these quantities for the second individual. In Forth, one would use offsets.

```
: OFFSET ( n -- ) ( adr -- adr')  CREATE , DOES> @ + ;
```

3. Defining verbs. Each verb will have two fields, one for its action and another for its status, used to modify the action. A dot convention provides a Pascal-like syntax.

```
    20  4 -BYTE-ARRAY  VERB-ARRAY
    0 OFFSET .ACTION
    2 OFFSET .STATUS
```

Now, 1 VERB-ARRAY .STATUS puts the address of the status of verb number 1 on the stack, and the action of the verb can be executed by 1 VERB-ARRAY .ACTION PERFORM Note that a blank must appear between the name of the array and its offset.

4. Naming verbs: Verb numbers are clumsy; names should be used. In Forth, it is easy to define a new class, VERB, that associates a verb number with a name.

```
VARIABLE V#     V# OFF
:  @INCR  ( adr -- n ) DUP @ DUP 1+ ROT ! ;
:  VERB  CREATE V# @INCR ,  DOES> @ VERB-ARRAY ;

VERB GO     (GO becomes verb number 0)
VERB TAKE   (TAKE becomes verb number 1)
```

TAKE becomes the equivalent of 1 VERB-ARRAY, and we can write TAKE .ACTION PERFORM for 1 VERB-ARRAY .ACTION PERFORM.

## THE USEFULNESS OF MIXED ARRAYS: MODULES#

Forth makes it possible to incorporate some advanced programing features at a rather elementary level. An example is a module, which as used here, will refer to a program feature that forms a conceptual unit, but which may affect the program in many different places. Let us add a watchdog to our adventure game. The watchdog is moved from room to room by a random process. When it is in the same room as the adventurer, it stops him from taking anything. Programming this entails changing the action of two verbs GO, which moves the adventurer from room to room, must also move the dog and set or reset a bit in TAKE .STATUS. TAKE must examine its status and act appropriately.

Forth provides a smooth way to make these multiple changes without actually rewriting each definition every time. This is done by including in the module a new definition that extracts the old verb action, combines it with the new, and inserts the blend back into the .ACTION field of the verb. Note how brackets suppress compilation during extraction. F-PC has a word DEFERS that works along similar lines. (The word MOVE-DOG is not actually defined in the illustration.)

```
\ Module: Watchdog

: MOVE-DOG ...; \ Make random move, set/reset bit 0 of TAKE .STATUS

: NEW-GO  [ GO .ACTION @ , ]  MOVE-DOG ;
NEW-GO  GO .ACTION !    \ Add MOVE-DOG to GO and put it back

: NEW-TAKE TAKE .STATUS @ 1 AND     \ Insert old action of GO
     IF [ TAKE .ACTION @ , ]    \  into IF  THEN phrase
    ELSE ." You can't do that!" THEN ;
     NEW-TAKE  TAKE .ACTION !   \ Put it back
```

## FORTH DO LOOPS AND A WORD ON FACTORING#

Forths rule for DO loop limits may look capricious, but can be justified in two ways: 1) It facilitates factoring, that is, breaking a definition into fragments, for reasons that may range from indispensable (saving memory or improving read- ability) to mere whim; and 2) It follows the Forth convention of starting things with zero. Factoring in Pascal is limited to writing a new subroutine. In this familiar example,

```
: SPACES ( n -- ) 0 DO SPACE LOOP ;
```

factoring has detached the starting value and the loop limit from each other, a freedom not available in Pascal Clean syntax is the result, but the effect is like splitting an infinitive: frowned upon by purists, but useful, nevertheless.

Although this usage is fairly intuitive, making a formal rule of it becomes convoluted, especially when negative increments are employed. The Forth 83 stand- ard says that a loop terminates when the index crosses the barrier between the limit and limit-1, a formulation that translates easily into assembler on many microprocessors. A good mnemonic is that forward loops never attain the limit, while back- ward loops never pass it. A consequence is that a loop will execute 216 times if the starting value and limit are equal. To execute zero times in this case, use ?DO instead of DO.

BASIC-style loops (Pascal, too) can be converted to Forth-style by a simple transformation. The following works in the usual case, in which the starting value and the limt are both positive.

```
: BASIC-STYLE ( n1 n2 --  n1 n2)
            2DUP < IF SWAP THEN 1+ SWAP ;
```

The usage is: `3 5 BASIC-STYLE DO ................ LOOP`

An occasionally useful feature of Forth loops is that the increment can be changed during execution.

## RECURSION#

Forth supports recursive words. The overhead is the same as any other word. Recursion can be removed from any algorithm by employing a stack, always available in Forth. As a result, Forth programmers have little occasion to use recursion Whether to write a word in recursive or non-recursive fashion is often merely a matter of choosing to use the return stack instead the parameter stack, or vice versa. On the other hand, a stack in Pascal needs to be user-written, and entails a good deal of overhead. As a result, recursive methods are often preferred.

Nevertheless, there are routines that are natural for recursion, both in Forth and Pascal, for example, traversing a binary tree. It would be a real challenge to make this non-recursive!

```
DEFER DO-NODE    ( node -- )
DEFER RB         ( node -- node') \node > right branch
DEFER LB         ( node -- node') \node > left branch
DEFER NIL        ( -- nil )

: TRAVERSE  ( node -- )  RECURSIVE
    DUP NIL =   IF DROP
            ELSE DUP LB TRAVERSE  DUP DO-NODE
                RB TRAVERSE
            THEN ;
```

## APPENDIX: PRACTICAL HINTS CONVERTING AN ALGEBRAIC EXPRESSION TO RPN#

Converting an expression from algebraic to reverse Polish notation is mechanical. It can and has been programmed, but a pencil-and-paper (carbon based) algorithm will do just as well for ordinary purposes.

1. Find a binary operation, put brackets around it, and rearrange it, putting the operator at the end. Remove parentheses and replace them with brackets as appropriate. Continue until done.

```
(A  + 3 * (B+C) ) * SQRT(D+8)

( A  + [ 3  (B+C)  * ] ) * SQRT(D+8)

[ A [ 3  (B+C)  * ] +] * SQRT(D+8)

[ [ A [ 3  (B+C) * ] + ] SQRT(D+8) * ]

[ [ A [ 3  [ B C + ]  * ] +] SQRT(D+8) * ]

[ [ A [ 3  [ B C + ]  * ] +] SQRT [ D 8 + ] * ]
```

2. Reverse unary operations, in this case SQRT.

```
    [ [ A [ 3  [ B C + ]  * ] +] [ D 8 + ] SQRT *
```

3. Remove brackets.

```
     A   3    B  C +    *   +    D 8 +   SQRT *
```

4. Append @ to those symbols that represent variables.

## Insertion Sort#

The insertion sort will be used as an example of how to convert an algorithm from Pascal to Forth. The insertion sort is the way most people arrange a hand of cards. The already-sorted cards are on the left and the unsorted are on the right. Look at the first unsorted card (gray), determine where it is to go, then remove it temporarily. The cards in between are moved up 1 space, and the gray card is slipped into place.

Although the insertion sort is not theoretically one of the best, it is actually rather good in Forth for arrays that are not too large, and is surprisingly efficient for arrays that are almost in order. It is almost as simple to program as the bubble sort, which it resembles. The version given compiles to 182 bytes, including headers. The reasons are:

1. Because a Forth array returns an address, one can move elements in blocks by means of code primitives such as CMOVE> many Pascals need inefficient loops.

2. For large random arrays, binary search makes insertion approach O(nlogn).

3. If an array is almost in order, insertion sort with backwards linear search is O(n) very efficient indeed! Sedgwick recommends it for finishing up quicksort, as it is faster than the latter in the late stages, when the array has been chopped up into segments of a few elements each.

### Pascal Version of Insertion Sort (Sedgwick)#

```
Procedure insertion;
        var i,j,v: integer;
        begin
        for i:=2  to N  do
            begin
            v:=a[i];  j:=i;
            while a[j-1]>v  do
                begin a[j]:=a[j-1]; j:=j-1
                end
            a[j]:=v         /* Put value into place */
            end
        end  ;
```

This looks simple enough, but Sedgwick admits that it will not work as written, and, despite his praise of it, he does not give a working version! In Pascal, a counted loop cannot be used for the inner loop, because there is no way to break out. Note that the comparison operator is > which, in Pascal, permits comparisons more complex than mere numerical inequality.

In the Forth version, comparisons are made by a deferred word COMPARATOR which compares data at 2 addresses and leaves a sign, an integer only the sign of which is important, the magnitude being irrelevant. The result should be positive if the record at a2 is greater than (comes after) a1. A VALUE named RL record length is defined as an ancillary. (Note that this version does not confine itself to the artificially simple situation where the records are single- precision integers, but can sort records of any desired length.)

The outer loop starts with 1, the index of the second element. The current element is compared with the 0th element of the whole array. If less, then this is the insertion point; otherwise the sorted part of the array is scanned backwards until a greater-or-equal comparison is found, and unstructured exit from the loop occurs. This always will occur, and it corrects the unworkable part of Sedgwicks version. Finally, the item to be moved is compared with its left neighbor; if greater, then it is already in place, otherwise the element is moved to PAD for temporary storage, shift is performed by CMOVE>, and the item is put in place. The last comparison is not essential, but it will prevent unnecessary calculation of 0-length moves, which are frequent when the array is almost in order.

## INSERTION SORT WITH BACKWARDS LINEAR SEARCH#

Leonard Morgenstern

```
DEFER COMPARATOR ( a1 a2 -- s )
0 VALUE RL  \ Record length can be any size, not just 16-bits
:  ISORT    \ a0 n rl (adr, number of items, record length)
IS RL       \ a0 n
OVER RL + SWAP  \ a0 a0+rl n
RL * BOUNDS \ a0 lim1 lim2 (adr, limits for outer loop)
DO      \ a0 I OVER      \ a0 i1 a0
 COMPARATOR  \ a0 s       (adr, "sign")
 0<=      \ a0 (True signifies that trade will be with a0)
 IF      \ a0
  I OVER      \ a0 i1 a0 (Set up to trade with a0)
 ELSE        \ a0
  I 2DUP      \ a0 i1 a0 i1
  RL -        \ a0 i1 a0 i1-rl (Set up for inner loop)
  DO          \ a0 i1
   DUP I       \ a0 i1 i1 i
   COMPARATOR 0>   \ a0 i1 f (Compare  inner loop with target)
   IF           \ a0 i1 (TRUE indicates this is the place)
    I RL +      \ a0 i1 i+rl (Set up for unstructured LEAVE)
    LEAVE
   THEN
   RL NEGATE +LOOP \ End of inner loop.
 THEN          \ a0 i1 ax
 2DUP =       \ a0 i1 ax f (True=nothing to move)
 IF
  2DROP      \ a0 (Just clear stack)
 ELSE
  OVER PAD RL CMOVE \ a0 i1 ax (Move i1 to pad)
  2DUP -       \ a0 i1 ax d (d = number of bytes to move)
  OVER DUP RL +    \ a0 i1 ax d ax ax+rl (Getting ready)
  ROT CMOVE>    \ a0 i1 ax (Move)
  PAD SWAP RL CMOVE  \ a0 i1 (Insert in place)
  DROP         \ a0 (Clean up stack)
 THEN
 RL +LOOP    \ a0 (End outer loop)
DROP        \ Clean up stack & end.
;
```