<table>
<thead>
<tr>
<th>Page</th>
<th>Section</th>
</tr>
</thead>
<tbody>
<tr>
<td>95</td>
<td>Historical Perspective</td>
</tr>
<tr>
<td></td>
<td>Publisher's Column</td>
</tr>
<tr>
<td>96</td>
<td>Balanced Tree</td>
</tr>
<tr>
<td>108</td>
<td>Letters</td>
</tr>
<tr>
<td>109</td>
<td>The Execution Variable and Array</td>
</tr>
<tr>
<td>111, 118, 119</td>
<td>Meetings</td>
</tr>
<tr>
<td>112</td>
<td>Project Benchmark</td>
</tr>
<tr>
<td>113</td>
<td>IPS — A German FORTH-Dialect</td>
</tr>
<tr>
<td>116</td>
<td>The CASE, SEL, and COND Structures</td>
</tr>
</tbody>
</table>
HISTORICAL PERSPECTIVE

FORTH was created by Mr. Charles H. Moore in 1969 at the National Radio Astronomy Observatory, Charlottesville, VA. It was created out of dissatisfaction with available programming tools, especially for observatory automation.

Mr. Moore and several associates formed FORTH, Inc. in 1973 for the purpose of licensing and support of the FORTH Operating System and Programming Language, and to supply application programming to meet customers' unique requirements.

The Forth Interest Group is centered in Northern California, although our membership of 2,000 is worldwide. It was formed in 1978 by FORTH programmers to encourage use of the language by the interchange of ideas through seminars and publications.
BALANCED TREE
DELETION IN FASL

Douglas H. Currie, Jr.
Nashua, NH

Abstract

FASL (Functional Automation Systems Language) is a derivative of FORTH containing significant modifications. This paper discusses one of these, the FASL tree, an implementation of the AVL (height balanced) tree. FASL trees are a data type of the language, and are used in the implementation of the dictionary. An algorithm for deletion in FASL trees is presented, as well as a FASL program to implement the algorithm.

Key Words and Phrases

deletion, height-balanced trees, binary trees, search trees, FORTH.

CR Categories

3.7, 4.10, 4.20, 4.34, 5.25, 5.31

Introduction to Height-Balanced Trees

The use of balanced trees has become almost commonplace in database management, and is seeing limited use in symbol tables. Many systems would benefit from the use of balanced trees, but their designers could not afford the time to develop the algorithms. A case in point is the extensive use of hashing in "high-speed" microcomputer assemblers. Hashing techniques have significantly improved the performance of many assemblers, but analysis of these routines shows a best case performance on the order of several milliseconds (due to the inefficiency of division, or pseudo-random number generation on microprocessors). FASL trees, on the other hand, have a guaranteed worst case performance of far less than a millisecond even in fairly large (over five hundred node) trees.

In FUNCTIONAL* systems, FASL trees are used in a line editor, data storage directories, FACT (a truth table compiler), message routing tables, microcomputer assemblers, as well as the FASL dictionary. A general purpose microassembler uses a balanced tree (fields) of balanced trees (contents) to describe the target micro-instruction. The use of multiple trees allows identical keys in different contexts (e.g., label names and macro names).

The height-balanced tree was first proposed by two Russian mathematicians, G. M. Adel'son-Vel'skiy and E. M. Landis in 1962 (hence AVL tree). The idea is to maintain a binary tree so that the height of the subtrees at any node differ by at most one. The technique incurs a penalty of only two extra bits per node (FASL uses an 8-bit byte), and makes it possible to search for, insert, or delete a node with a worst case of $O(\log N)$ operations (where $N$ is the number of nodes).

Introduction to FASL Trees

Algorithms for search and insertion in AVL trees are presented by Knuth (The Art of Computer Programming, Vol. 3, Section 6.2.3); these two algorithms were implemented in machine code and (along with Indirect Threaded Code) became the basis for FASL. The deletion algorithm was not implemented at this time for two primary reasons: Knuth didn't give it, FASL didn't "need" it. Deletions occur much more rarely than insertions or searches; FASL lived for over a year with no delete operation.

*Functional Automation Gould Inc.
3 Graham Drive
Nashua, NH 03060
For example, when a file was deleted from a FASL directory, the entire directory was reconstructed without the "deleted" node. The time penalty incurred was not significant because directories are small (for FASL trees), and had to be copied anyway to be sent to the disk. (FASL lives in a message environment. The disk is in another Cyblok*).

After an overview of FASL trees and their use, the remainder of this paper will deal with the development of a FASL tree deletion program in FASL. For an introduction to binary search trees, see Knuth (The Art of Computer Programming, Vol. 3).

FASL trees are composed of a number of sixteen byte nodes (see Figure 1). The tree is identified with the address of its head node. From the head node we may find the root node, and thus the entire tree. The head node contains a pointer to its root node, a pointer to its available nodes list, and an integer which is the tree's height.

All nodes other than the head node contain an eight byte key, a left link, a right link, a one byte balance factor, and three uncommitted bytes. The key is used to access the node. Given a key, the search routine compares it to the key at the root node. If it is less, the search continues with the node identified (pointed to) by the left link. If it is greater, the search continues with the node identified by the right link. The search terminates when it matches the key (success), or reaches a null link (failure). The null link is represented by zero. The balance factor is the height of the right subtree minus the height of the left subtree. The insertion routine always leaves the tree balanced, i.e., the balance factor is always minus one, zero, or plus one.

The insertion routine obtains new nodes from the free nodes list. This list is simply a number of nodes linked with their right links. A null right link indicates the end of the free nodes list. When the insertion routine needs a free node, it obtains its address from the free nodes list pointer in the head node, and replaces it with the right link of that node. If the free nodes list pointer is null, then the tree is full.

The technique used by the insertion routine to maintain tree balance is essentially the same as for deletion. Basically, four cases arise in insertion when the tree must be rebalanced: single or double rotation, left or right. The discussion is postponed until the section on deletion.

*Cyblok is a registered trademark of Functional Automation/Gould Inc.
To get a feeling for the efficiency of FASL trees, consider a dictionary of five hundred nodes. If this dictionary was stored as a linked list, a worst case access time of five hundred compares would be incurred, with an average access time of two hundred fifty compares. Stored as a FASL tree, this dictionary has a worst case access time of nine compares, an average of eight. The numbers become even more convincing as the dictionary grows in size.

FASL Tree Operations

FASL provides operations for creating trees, inserting and searching for nodes, and accessing the uncommitted data in a node. For example, the FASL text

```
100 TREE SYMBOLS
```

creates a tree named SYMBOLS with two hundred fifty-six available nodes (the radix is hexadecimal). Assuming there is a string of text in an area named PAD which is to be used as a key to access the tree,

```
PAD SYMBOLS LEAF
```

inserts a node in the tree SYMBOLS with this key. LEAF leaves a boolean flag on the stack to indicate success or failure, and if successful leaves the address of the new node on the stack under the boolean.

Usually, new nodes are initialized with some data. The following FASL text will insert a node with the key in PAD (as above), and initialize its uncommitted bytes with constants:

```
12 3456 PAD SYMBOLS LEAF
  IF F#!
  ELSE DROP2
  FI
```

Later, the data may be retrieved onto the stack as follows:

```
PAD SYMBOLS FIND
  IF F#@
  ELSE FAIL FAIL FI
```

If the string in PAD is the same as was used in the preceding example to insert the node, then the data retrieved will be 12 3456. If another string is in PAD, then the data retrieved will be 00 0000, unless a node has been inserted with this string as a key, in which case the data associated with this node will be retrieved.

From the example, it should be clear how to use the FASL trees for a symbol table for an assembler. Text is read to PAD until a delimiter, and then inserted in the tree. In the case of labels, the node would be initialized with the current pseudoPC, and a flag byte to indicate "label." If the inserted text was a macro name, the node might be initialized with a pointer to the macro text and a flag byte to indicate "macro." Alternatively, separate trees may be created so that identical keys may be used as macro and label names. Later, when a label or macro is used, it may be looked up in the tree to find its corresponding values.

The TREE operation allocates space for the tree in the FASL Global Area (where code for colon-words is placed). Another operation, TREEINIT, is provided to initialize trees in space that the FASL user has allocated (e.g., in FUNCTIONAL Cybloks there is a minimum of 256K bytes of "Public Memory" which is accessed through "Windows," and is not part of the FASL Global Area). The TREEINIT operation is often used in the Local Area (space allocated on the Return Stack) or in Public Memory.
The Deletion Algorithm for FASL Trees

A deletion algorithm for binary trees, and the steps required to adapt this algorithm to balanced trees are provided by Knuth (The Art of Computer Programming, Vol. 3, Sections 6.2.2 and 6.2.3). The details of the balanced tree deletion algorithm are presented here, but first a review of binary tree deletion.

Deleting a node from a binary tree may be decomposed into four cases (see Figure 2). Call this node "X". In the first two cases one of the links of X is null, the other link is a "don't care" (i.e., a pointer or null). In both cases the other link simply replaces the link pointing to X. In case three the right son of X has a null left link. In this case the left link of X replaces the left link of its right son, and the right link of X replaces the link pointing to X. In case four the symmetric successor of X must be found. This is done by following left links starting with the right son of X until a null link is encountered. The left link of the father of the symmetric successor is replaced by the right link of the symmetric successor. The left and right links of the symmetric successor are replaced by the respective links of X, and the link which points to X is replaced by a pointer to the symmetric successor.

In all cases the essential left-to-right order of the nodes is preserved. The deleted node is inserted in the free nodes list, and the algorithm terminates.

All that is required (!) to adapt this algorithm to balanced trees is to insure that the balance is maintained after the deletion. An important observation is that the effect of deletion on the binary tree is to reduce the length of a single path through the tree by one.

This path begins at the head, and ends in cases one and two with the node which replaced X (i.e., the node which is pointed to by the link which used to point to X). In cases three and four the path ends with the node which used to be the right son of the symmetric successor of X. (Note that the ending node may actually be null.)

The path may be represented as a list of pairs

\[(N.0, f.0) (N.1, f.1) \ldots (N.i, f.i)\]

where each N.j is a node address, and each f.j is a direction (-1 left, +1 right). N.0 is the head node, f.0 is the +1 (since the "right link" of the head node points to the root). The pair (N.i, f.i) is the end node minus one, and identifies the end node of the path (which, again, may be null). Rebalancing may be required at each node in the path, starting with node (N.i, f.i), working backwards. This is in contrast to insertion where rebalancing is required for, at most, one node.
Adapting the deletion algorithm for binary trees to balanced trees requires that as the tree is searched for the node to be deleted (and for its symmetric successor in cases three and four), a list of pairs describing the path is created. Once the node is deleted, nodes are rebalanced back along the path until a termination condition is reached.

The path is constructed on an auxiliary stack. The operations "Push(x,y)" to push a pair, "Pop(x,y)" to pop a pair, and "Top(x,y)" to read the top pair without popping are used, as well as the capability of saving and restoring the path stack pointer.

Using the notation "Link(-1, M)" for left link of node M, "Link(1, M)" for right link of node M, "Bal(M)" for the balance factor of node M, and "Key(M)" for the key of node M, the following is a detailed algorithm for deleting the node with key K in a balanced tree.

1. Initialize local path stack. Push(HEAD, +1). Set X to Link(+1, HEAD).

2. If K is less than Key(X), go to (3) moving left. If K is greater than Key(X), go to (4) moving right. Otherwise go to (5), key is found.

3. If Link(-1, X) is 0, go to (11), key is not in tree. Otherwise Push(X, -1), set X to Link(-1, X), and go to (2), keep searching.

4. If Link(1, X) is 0, go to (11) key is not in tree. Otherwise Push(X, 1), set X to Link(1, X), and go to (2), keep searching.

5. There are four cases:

   5a) Link(1, X) = 0; Top(N.k, f.k). Set Link(f.k, N.k) to Link(-1, X). Go to (7) to rebalance.

   5b) Link(-1, X) = 0; Top(N.k, f.k). Set Link(f.k, N.k) to Link(1, X). Go to (7) to rebalance.

   5c) Link(-1, Link(1, X)) = 0; Top(N.k, f.k). Set Link(-1, Link(1, X)) to Link(-1, X). Set Link(f.k, N.k) to Link(1, X). Set Bal(Link(1, X)) to Bal(X). Go to (7) to rebalance.

   5d) Otherwise; Push(X, 1), set Z to Link(1, X). Save path stack pointer in PSP. Go to (6) to find symmetric successor.

6. Push(Z, -1). Set Z to Link(-1, Z). Repeat this step until Link(-1, Z) = 0. Finally, Top(N.k, f.k). Set Link(-1, N.k) to Link(1, Z). Set Link(-1, Z) to Link(-1, X). Set Link(1, Z) to Link(1, X). Now swap PSP and the path stack pointer. Pop(N.k, f.k), Top(N.k, f.k), Push(Z, 1), substituting the symmetric successor for the deleted node on the path stack. Swap PSP and the path stack pointer again to restore. Set Link(f.k, N.k) to Z. Set Bal(Z) to Bal(X). Go to (7) to rebalance.
(7) Insert X into the free nodes list.

The algorithm proceeds as follows beginning with the last pair of the path:

(8) Pop(N.k , f.k).
    If N.k = HEAD, set Height(HEAD) to Height(HEAD)-1 decreasing the height of the tree, and go to (11) terminating the algorithm. Otherwise go to (9).

(9) There are three cases based on the balance factor:

(9a) Bal(N.k) = 0 ; Set Bal(N.k) to -f.k, and go to (11) terminating the algorithm.

(9b) Bal(N.k) = f.k ; Set Bal(N.k) to 0, and go to (8) taking one more step back along the path.

(9c) Bal(N.k) = -f.k ; Rebalancing is required, go to (10).

(10) There are again three cases.
    (Referring to Figures 3, 4, and 5, A is N.k, α is the subtree containing the path the algorithm has been following, B is the node pointed to by the opposite link from the link which points to α, Link(-f.k , N.k)):

(10a) Bal(A) = Bal(B) (Figure 3);
    Set Bal(A) and Bal(B) to 0. (single rotation)
    - Set Link(-f.k , A) to Link(f.k , B).
    - Set Link(f.k , B) to A.
    - Top(N.k , f.k), set Link(f.k , N.k) to B.
    Go to (8) taking one more step back along the path.

(10b) Bal(A) = -Bal(B) (Figure 4); If Bal(X) = Bal(A), then set Bal(A) to -Bal(X) and Bal(B) to 0. Otherwise set Bal(A) to 0 and Bal(B) to -Bal(X).
    Set Bal(X) to 0. (double rotation)
    - Set Link(-f.k , A) to Link(f.k , X).
    - Set Link(f.k , X) to A.
    - Set Link(-f.k , B) to Link(-f.k , X).
    - Set Link(-f.k , X) to B.
    Top(N.k , f.k), set Link(f.k , N.k) to X.
    Go to (8) taking one more step back along the path.
(10c) \( \text{Bal}(B) = 0 \) (Figure 5);
Set \( \text{Bal}(B) \) to \( -\text{Bal}(A) \).
(same rotation)
Set \( \text{Link}(\neg f.k , A) \) to \( \text{Link}(f.k , B) \).
Set \( \text{Link}(f.k , B) \) to \( A \).
Top(\(N.k , f.k)\), set \( \text{Link}(f.k , N.k) \) to \( B \).
Go to (11) terminating the algorithm.

(11) Deallocate path stack. Done!

---

**Implementing the Algorithm in FASL**

A FASL program to implement the balanced tree deletion algorithm is relatively straightforward (see the listing below). Some preliminary colon-words are defined to access the links, and to access a Local Stack. RCRUMB and LCRUMB are defined (in commemoration of Hansel and Gretel) for adding pairs to the path stack; then colon words for the three cases encountered in rebalancing are defined.

The main colon-word, DROPLEAF, takes stringname and treename parameters just like LEAF and FIND, but leaves no return values since it is always successful. The PROC... ENDPROC pair allocate and deallocate a Local Data Area for the path stack and associated variables. For the most part, DROPLEAF follows the...
deletion algorithm presented. Nested IF statements are used to evaluate the case constructs. The string compare in the first (search) WHILE loop tests for less-than directly, and examines FASL Registers (WO, W1) to resolve the trichotomy. (This is an efficiency measure, and has to do with the fact that there is not guaranteed to be a string delimiter in the node's key.)

Empirical tests show that DROPLEAF runs in the 50 to 100 millisecond range for trees with about 500 nodes. For comparison, LEAF runs in the 0.1 to 1 millisecond range on the same trees. The large difference between these runtimes results from the fact that LEAF is highly optimized machine code, only requires one rotation maximum, and does not require a path stack. As previously mentioned, DROPLEAF is used very infrequently, and there has been no incentive to implement it in machine code.

```forth
( HEIGHT BALANCED )
( TREE DELETE )
( 17Mar80 )

( LOCAL DATA AREA )
( OFFSET )
( ------ )

2 saved path stack pointer
4 path stack pointer
6 address of link to node to be deleted
8 start of path stack
.
.
30 end of path stack + 1
)

( 1.1 )
: LLNK! 2 + @ ;
: LLNKR 4 + @ ;
( 2,0 )
: LLNK! 2 + ! ;
: LLNK! 4 + ! ;

( 1.0 )
: PUSH 4 'D @ 2 + 'D + ! ;
( 0.1 )
: Pop OFFFE 4 'D + ! 'D @ ! ;
( 1.1 )
: CRUNS DUP PUSH OFFFE PUSH ;
: LCRUNS DUP PUSH SUCCEED PUSH :

( 3.2 )
: SINGLROT OVER2 LTZ!
IF DUP LLNK! OVER2 LLNK!
SWAP OVER LLNK!
ELSE DUP LLNK! OVER2 LLNK!
SWAP OVER LLNK!
FI ;

: ROTCASE1 FAIL OVER C! FAIL OVER2 C1
SINGLROT SWAPDROP FAIL SWAP ;

: ROTCASE2 FAIL OVER C! OR OVER C1
SINGLROT ;

: ROTCASE3 OVER C! OR OVER C1
SINGLROT SWAP DROP OVER C!

: ROTCASE4 OVER C! OR OVER C1
SINGLROT SWAP DROP OVER C!

: ROTCASE5 OVER C! OR OVER C1
SINGLROT SWAP DROP OVER C!

ELSE OVER C! 1 DO DUP DUP OVER SWAPDROP SWAPDROP SWAPDROP DROPH2.

: MOVESZ D U 6 'D : 8 ;

( 2,0 )
( <name> <tname> )
: DROPLEAF
30 PROC 8 'D 4 'D !
SWAP OVER CRUNS 4 MOVELS WHILE DUP
IF OVER C! OR - SLIT! DUP IF OVER 10 W + W 1 - ELSE W! 0 - C! FI ELSE FAIL FAIL FI
CONTINUE
IF LCRUNS 2 ELSE CRUNS 4 FI MOVELS WHILE SWAP DROP SWAPDROP DROPH DROPLINK
IF DUP LLNK!
IF DUP LLNK! DUP LLNK!
IF 4 'D @ 2 'D 1 CRUNS DROPLS
REPEAT LCRUNS SWAPDROP DUP LLNK! DROPLS DUP LLNK! ZEROP
UNTIL OVER2 LLNK! OVER LLNK!
DROPLS OVER2 LLNK! OVER2 LLNK! OVER LLNK! OVER LLNK! SWAPDROP
DUP 2 'D 0 :
ELSE OVER LLNK! OVER LLNK!
CRUNS FI
OVER C! OVER C1
ELSE DUP LLNK! FI
ELSE DUP LLNK! FI
6 'D @ 1
OVER OA # OVER LLNK! OVER OA + 1
REPEAT
POP POP OVER2 OVER SWAP
IF DUP C! DUP IF OVER2 + SWAP AND
IF OVER 3 + OVER + C! DUP C! IF OVER2 OFF AND OVER C! -
IF ROTCASE2 ELSE ROTCASE1 FI ELSE ROTCASE3 FI
POP POP DUPSU SWAP DUPSU DUPSU - 3 + 1
ELSE FAIL SWAP C! DROP FAIL FI
ELSE DROPLS C! SUCCEED FI ELSE 2 + 1 SUCCEED FI
UNTIL ELSE DROPLS FI DROP ENDPROC
```

Page 103
FASL Credits

FASL arose in response to a need within FUNCTIONAL for a simple and efficient interpreter for system software development. An early FASL Manual (1977) was written with contributions from Eric Frey, Michel Julien, Roland Silver, and Ron Lebel. The idea of implementing the dictionary as a height balanced (AVL) tree came a year later, and with it the FASL TREE data type.

FASL was also made possible by the unselfishness of G. M. Adel'son-Vel'skiy and E. M. Landis, Donald E. Knuth, and Charles Moore.

The author has recently learned of two language processors which use AVL Trees for symbol tables, but not as a data type of the language: a MUMPS system (Dave Bridger for Tandem), and the IBM FORTRAN H Compiler. The current status of these language systems is not known by the author.

Special thanks to Kit Andrews for typing the manuscript on Functional's Wang Word Processor, and patiently illustrating the final versions of the Figures.

Assembler Listings for Search and Insertion

The following pages contain excerpts from the FASL listings pertaining to tree search and insertion for the 6800. Referring to these listings:

(1) The names used in the comments correspond to those used in Knuth's Algorithm 6.2.3A.

(2) The routines use variables HEAD and AVAIL to identify the tree and free nodes list on each invocation; the key should be in the eight byte area K.

(3) The variable VTV may be initialized to point to the default subroutine DEFNOT which causes a "failure" return on an insertion attempt to a full tree, or to a user supplied subroutine which allocates a new free nodes list (with at least one node) by placing the address of the list in AVAIL.

(4) Trees are initialized by placing a starting address in HEAD, an ending address in AVAIL, and calling the routine BTSIUP. On entry, AVAIL-HEAD should be greater than thirty-two, and zero mod sixteen. On exit, HEAD will not be modified and will point to the head node, and AVAIL will point to the free nodes list.

(5) All tree routines are object code relocatable.

(6) Quickie symbol table for these listings:

<table>
<thead>
<tr>
<th>Routine</th>
<th>Symbol</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BTSIUP</td>
<td>E151</td>
<td>tree initialization</td>
</tr>
<tr>
<td>FINDIT</td>
<td>E168</td>
<td>tree search</td>
</tr>
<tr>
<td>BTSI</td>
<td>E17D</td>
<td>tree insertion</td>
</tr>
<tr>
<td>DEFNOT</td>
<td>E660</td>
<td>default tree overflow subroutine</td>
</tr>
<tr>
<td>K</td>
<td>D0</td>
<td>key for search &amp; insertion, 8 bytes</td>
</tr>
<tr>
<td>HEAD</td>
<td>C2</td>
<td>pointer to tree</td>
</tr>
<tr>
<td>AVAIL</td>
<td>C4</td>
<td>pointer to free nodes list</td>
</tr>
<tr>
<td>VTV</td>
<td>C0</td>
<td>overflow transfer vector</td>
</tr>
</tbody>
</table>
I would like to point out a possible misconception that I noticed in one of the judge’s comments on page 54 in the special FD on Case Structures. The third item listed as an "advantage" states "(The) case selector is kept on (the) return stack instead of in a special variable. This allows nesting of CASE constructs." I'd like to point out that the FORTH-85 CASE structure, which uses a variable (VCASE), is also nestable. The reason for this is that once a match has been made and execution is in progress between CASE ...END-CASE the contents of VCASE have served their purpose. Further nesting at this point can alter the contents of VCASE without problems. When the unnesting occurs, END-CASE shoots the Forth instruction pointer to the words after the end of the case structure. END-CASE does not need the older contents of VCASE. If the programmer would like to retain the selector value, a simple "VCASE @" directly after CASE will preserve the contents of the stack. Then, for any following Forth words having nested DO-CASE structures, the problem of overwriting is solved. The variable storage method takes a little longer to retrieve the current selector value (i.e. VCASE @ versus DUP, or versus I), but retrieving VCASE has not been very common in my experience. To me VCASE @ is more self-explanatory in the context of the program than either DUP or I. In addition, my feeling is that messing up the return stack so the normal index values (I & J) cannot be used within a CASE . . . END-CASE phrase, is a definite disadvantage. To solve return stack problems like this, advanced Forth Systems, such as the one now at Kitt Peak or STOIC, have three stacks. The extra stack is used explicitly for LOOP indices while the return stack is used for return addresses and temporary storage. In lieu of a third stack, the VCASE variable presents a clear way of handling this situation. The variable storage method would need to be changed to user variable storage if multi-tasking was to be implemented. This is only slightly more complicated than the current version. In my extension, I tried both return stack and variable methods. I selected the variable storage due to speed improvements as well as the arguments above. Also, in regards to speed, the CALL's and JMP's within the code statement for CASES are weak in style since the objective in code statements is speed. These really should be expanded out (i.e. MACRO'd!). My original intent was to make the article do double duty be demonstrating these techniques as a stepping stone to some debugging methods I came up with.

Bob Giles
Tulsa, OK

LETTERS
THE EXECUTION VARIABLE AND ARRAY:

Michael A. McCourt
University of Rochester

A useful programming construct is the jump table or 'COMPUTED GO TO' type of structure. In Forth the execution variable and array can be used. The Forth word EXECUTE executes the code address on the top of the stack. If one defines:

: XEQ <BUILDS, DOES> @ EXECUTE;

a word containing a code address as its parameter can be created. As an example

: TEST ." THIS IS A TEST" CR ;
0 XEQ FRED ' TEST CFA ' FRED 2+ !

The word TEST can now be executed by typing FRED. You might ask--why not type TEST to execute TEST? The reason is that FRED is now a variable--of sorts. By changing the contents of the parameter stored in FRED the action of FRED can be changed. Execution arrays are similar, however, here several code addresses can be stored and later accessed by index number. In our Forth system (an updated URTH system to Forth-79 running on a PDP-11) the Forth code address of zero is disallowed and will cause execution of the current ABORT procedure which itself is contained in a variable, i.e.

: ABORT ABEND @ EXECUTE ;

All execution variables and arrays are initialized to zero so that they will have predictable results.

Three words shown in block 502 listed below are used to change the contents of execution variables and arrays.

INSTALL <name>

returns the code field address of <name>.

<code addr> IN <XEQ var name>

stores the code address in the parameter field of XEQ name.

<code addr><array offset> OFFSET.IN < ()XEQ array name>

stores the code address at the offset in the ()XEQ array.

Thus the previous example could be written as

0 XEQ FRED INSTALL TEST IN FRED

Note that INSTALL and IN work within a colon definition, e.g.,

: DUMMY ;
: TURN.ON INSTALL TEST IN FRED;
: TURN.OFF INSTALL DUMMY IN FRED;

Execution variables are useful for a variety of functions such as creating forward references, switching output and/or input routines among several terminals, debug routines and of course implementing a jump table.

Examples

1. JUMP TABLE

Problem:

Define a function that will perform one of 26 operations depending on which control key was typed.

Possible Solution:

26 ()XEQ CTRL.KEY
INSTALL FUNCTION 1 OFFSET.IN CTRL.KEY
INSTALL FUNCTION 2 OFFSET.IN CTRL.KEY

: OPERATOR? BEGIN KEY DUP 27 <=
   IF CTRL.KEY ELSE DROP THEN AGAIN;

One could implement the above with a case or select statement, but the execution array has less overhead in execution speed and memory usage.

2. MULTITERMINAL DRIVERS

Problem:

One has a video terminal with addressable cursor and a 'dumb' hard-copy terminal. The latter terminal does not accept cursor control characters gracefully.

Possible Solution:

One solution which alleviates this problem is shown listed below in block 500. (Publ. note: we're not printing block 500.) The word CTRL is an execution variable. When the video terminal is operating (TT1) all control characters are EMIT'ed; however, when the printer is installed (TTO) the control characters are DROP'ed.

The words EMIT and KEY are defined as state variables as is ABEND (user variables might be a familiar name to some) and are addressed for multi-tasking. They permit each task access to its own terminal driver.

: TEST2 0 0 TPC." TESTING" ;
   ( POSITION CURSOR AND PRINT )

TT1 TEST2 (TESTING' WILL START AT POSITION <0,0>)

TTO TEST2 (CONTROL CHARACTERS FOR 0 0 TPC HAVE NO EFFECT)

22 LIST (LISTING SENT TO PRINTER)
TT1 (BACK TO DISPLAY)

3. FORWARD REFERENCE

At times early in an application program one needs to define an error handling routine. However, since none of the higher level words have been defined the error handling is rather primitive. Execution variables allow one to 'leave a blank' for the error routine.

Suppose one has

O XEQ DERROR

<device function code>

: DIO GO.BIT OR DEVICE.CONTROL !
   WAIT.FOR.DEVICE.DONE
   DEVICE.STATUS @ 0< IF DERROR THEN;

Assume DIO is for control of a mag tape drive. At this point in the application program DERROR would normally be able to do only an ABORT. With a tape drive one would prefer to have some sort of recovery procedure on write errors to either delete the last file or at least write an End of File mark. With the execution variable one can install such a high level routine at a later time after all the necessary words (such as skip record, read record, and write EOF) have been defined. DERROR could also be defined as an ()XEQ array and each error would have its own associated error handling.

The previous examples demonstrate the power of the <BUILDS ... DOES> Forth constructs. XEQ and ()XEQ are just two examples of defining words. It is possible to build a wide range of such defining words from words that build simple linear arrays to ones that define complex relational data bases. In all cases one is associ-
ating a data structure (here, a simple code address) with an algorithm for using the data (here, EXECUTE the code address) and as Wirth has written DATA STRUCTURES + ALGORITHMS = PROGRAMS*


Bill Ragsdale announced the Asilomar FORTH retreat (cf., FD Vol. II No. 3 for details).

Kim Harris described OPTIMIST, a program which reminded me of a cantankerous ELIZA. This FORTH program, originally written in PL/1 by Kildall, exemplifies a SECURED vocabulary as part of Kim's tutorial on PRIVATE VOCABULARIES. He showed how they are produced, tested and sealed.

Howard Pearlmutter discussed FIGGRAPH and the "human interface" of FORTH. The FIGGRAPH committee is to generate and articulate hardware specs, goals, and a vocabulary. Howard advised us to attend the HOME BREW COMPUTER CLUB'S showing, via a G.E. LIGHT VALVE, of computer graphics. (I saw it and it was as entertaining as LASERIUM).

Handouts included:

- Harris' OPTIMIST and PRIVATE VOCABULARY support
- Zimmer's TERMINAL, a program to teach a FORTHed Ohio Scientific Instruments OS-650v3 to act dumb

**HELP WANTED**

SENIOR PROGRAMMER to produce new poly-FORTH systems and applications.

Contact: Carol Ritscher
FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254
PROJECT BENCHMARK

A small, informal group of microcomputer enthusiasts here in Albuquerque read with interest "Project Benchmark" in the June issue of the magazine "INTERFACE AGE." We have amongst us a variety of systems and languages, including 8080, 6800, and the AM-100, interpreter and compiler versions of BASIC, and fig-FORTH on the three system types. We ran the benchmark program all around and have attached the results of our testing.

We found the results to be most interesting and offer them to the members of the Forth Interest Group. In addition to the timing results, there was also a significant advantage in memory for the FORTH programs. The compiled AlphaBasic program size was 192 bytes while the FORTH benchmark program size was 166 bytes. All three implementations of FORTH were based on the fig model, and the program ran without modification on all systems demonstrating the transportability achievable with FORTH.

I have attached a listing of the FORTH program. The implementation of the language for the 8080 and the 6800 were from fig, while the Alpha Micro version was provided by Sierra Computer Co., Albuquerque, NM.

George O. Young III
Albuquerque, NM

FORTH PROGRAMMERS (or ASSEMBLY programmers who want to learn FORTH).
Contact: Gary Osumi (714) 453-2345
Hydro Products, San Diego, CA
IPS
A GERMAN FORTH-DIALECT

Dr. Karl Meinzer
Marbach, W. Germany

The AMSAT-Phase III communication satellites for radio-amateurs utilize a computer on board for a variety of tasks. In order to simplify the programming and to allow a simple dialogue with the spacecraft the language IPS was developed (in 1976). It is a Forth-derivative geared very strongly towards engineering applications (real-time control) and by now it is also used in a variety of control-related areas. The following lines describe the rationale of the system and its main differences as compared to FORTH.

Area of Application

The IPS development was aimed in particular towards the "low" end of computers. Most control applications do not justify a larger computer for cost reasons. On the other hand, these applications profit most from a powerful language processor since the common techniques are very clumsy to use. The computer I had in mind when I designed IPS was at about the level of the TRS-80 with 16K bytes of RAM (integral video memory and cassette for mass storage). For real-world interactions control-I/O and a 20ms interrupt must be added to complete the system.

The IPS Language

An introduction to IPS was given in BYTE, Jan. 1979, pp. 146; so here I want to explain the difference to FORTH. First: for the names I tried to find words which are more logical in a postfix environment. Take the IF ELSE THEN construct, e.g., in IPS it is replaced by YES? NO: and THEN. This seemed more logical since the IF implies a test following. But with the preceding test YES? is more appropriate. Of course these fine points may not be very important. Others are more so: numbers used as truth-variable on the stack use only the least significant bit. This allows the 16-bit logic operators like AND OR or XOR to be used consistently with truth-variables.

A major difference is the way names are encoded. I did not like the limitations coming from the 3 characters plus length codes; but then neither did I want to use more than 4 bytes for the code. The following technique was adopted: from all characters of the name (up to 63), a division remainder using the polynomial X24 + X7 + X2 + X1 + 1 is computed (3 bytes) and stored with the length of the name. This technique allows arbitrary names; e.g., MACHINE-A1 and MACHINE-A2 are distinct and not confused by the system.

Theoretically there is a small (10 to the -7) probability of a collision -- in practice I never yet encountered one. In any case, no harm can come from this because in IPS the system does not allow the redefinition of names. This "advantage" of FORTH was dropped very early because from our user-feedback it soon became clear that it was--directly or indirectly--one of the major causes for programming errors.

Other plausibility checks were added to make the system more forgiving against the typical programming blunders. (I do not believe in the FORTH-assumption that the programmer can be perfect--I am a good example to the contrary). In fact, a few checks can make the system virtually crashproof. Of course, one has to be careful not to get carried away with this--if the integrity of the system is reduced, much of the power of a FORTH-like language goes away.
Three examples within IPS:

- During definitions the colon puts an unused address on the stack. The semicolon checks for this number: if it finds a different number, most likely a structuring error has occurred. The definition is removed and an error message is written.

- Each word has a unique 2-bit identification in the name field defining its use in the interpretive mode. Words like YES?, for example, are not executed outside definitions—so no "magic effects" can result.

- The number of interpreter states the programmer has to keep in mind is minimized. The base for number conversions is set explicitly. Numbers like 40 or -721 are treated as decimal, #03 or #AF07 as hexadecimal numbers.

Real-Time Multiprogramming

The typical situation with real-time control has the processor waiting for some event, then executing a task—usually very fast—and then again waiting for other events. In practice, typically the computer must attend to a number of such tasks. This allows for a fairly simple multiprogramming concept. The tasks are put in a cyclic "chain," an array containing the addresses of the tasks to be executed. The system executed them periodically in a roundrobin fashion. Provided that none of the tasks "grabs" the processor this results in a reasonably fair arbitration of processor time and was found sufficient for most control applications. Two operators are provided to allow dynamic and static task allocations: INCHAIN and DECHAIN.

The interpreter/compiler is also a task in this sense—it executes one word at a time before it returns to the chain. This keeps all the debugging capability of the interpreter a hand while other tasks are executing.

The system is augmented by the concept of "pseudo-interrupts." The address interpreter (NEXT) is effectively a stack-machine which has ideal properties for interrupting it—no saving is required. If the address interpreter can accept these pseudo-interrupts between the execution of code-routines, a very powerful high-level interrupt-concept is possible. In IPS such a pseudo-interrupt is executed every 20ms to keep the keyboard alive and for timekeeping purposes. Other pseudo-interrupts may be added as required.

Signalling to the address interpreter the pseudo-interrupt request without creating additional overhead is a bit involved with most processors. Only with the CDP 1802, this is straightforward—the address interpreter contains a jump that can be made conditional on an external signal (External flag). With the other processors a real interrupt is used to modify the code of NEXT; admittedly a less than desirable way of programming. Since this occurs only at a single point, it was considered to be the lesser evil over a possibly increased duration of NEXT.

Handling and Testing

IPS is strongly TV-screen oriented. This allowed the stack to be continuously visible by putting a display-program into the chain. For debugging it is a great help not having to request the stack-content, but seeing it continuously. During the operation of chain-operators the system remains "live," you always can go after problems and investigate.
Typically, programs are first written on cassette with the integral text-editor as blocks of 512 bytes each. Then the blocks are compiled and tested. If necessary, blocks may be edited on the cassette and recompiled to solve bugs. Eventually a binary dump of the whole program (IPS plus application) is produced to facilitate fast reloading.

Experiences So Far

Primarily, the system was developed for the Phase III spacecraft that was launched in May 1980. It gave the handling of the satellite an unprecedented degree of flexibility and at the same time helped to solve the rather complex attitude control problems with a minimum of pain. The spherical trigonometry of the satellite was solved very elegantly by CORDIC-type rotation operators rather than the conventional solution using sines and cosines. This allows a geometrical analysis of the problems rather than the much more complicated algebraic analysis.

Unfortunately the launcher (ARIANE L02) failed and the spacecraft was destroyed—a repeat is scheduled for early 1982. The ground equipment also uses IPS. An English version for the 8080 using an S-100 bus computer was used for the safety surveillance computer.

Furthermore, a large number of COSMAC based computers within the University of Marburg utilize IPS for a number of research-data-acquisition tasks. All in all, our experience with the system has fully met our goals—to simplify real-time control.

The Problem of Distribution

With the real-time capabilities of IPS, portability of the system is much more difficult to achieve than with more common language processors—the hardware configurations have much more connections with the system than say with a BASIC interpreter. Typically we modify the IPS meta-source to match the hardware at hand and then run the source through a meta-compiler producing the new system. The lack of suitable "standard-computers" having the required real-time hardware extensions so far has prevented a very widespread distribution of IPS. Now we have a version running on the TRS-80 with a few restrictions; by adding some hardware these restrictions go away. As a next step we intend to build a meta-compiler running on an unmodified TRS-80. Hopefully this way we can get "out of the cycle" and thus enable a widespread distribution of IPS. The large number of letters I received after the BYTE paper convinced me that the need for such a system is very real. I should be pleased if this letter also presents a stimulus to FORTH programmers to add some of the IPS concepts to enhance its usefulness for real-time control.

AUTHORS WANTED

Mountain View Press, the source for printed FORTH, will publish, advertise and distribute your FORTH in printed form. Substantial royalty arrangement.

Contact:  Roy Martens
Mountain View Press
PO Box 4656
Mt. View, CA 94040

HELP WANTED

PROJECT MANAGER to supervise applications and special systems projects.
Contact:  Carol Ritscher
FORTH, Inc.
2309 Pacific Coast Hwy.
Hermosa Beach, CA 90254
THE CASE, SEL, AND COND STRUCTURES:

Peter H. Helmers
University of Rochester

The following is a description of the three "case-like" structures which have been added to URTH for the Ultrasound Lab in the Department of Radiology at the University of Rochester. These three structures were evolved from a simpler prototype CASE statement developed by Rich Marisa at the University's Towne House Computer Center and by Larry Forsley at the University's Laboratory for Laser Energetics.

Execution Time Operation

The three structures to be described are the CASE, SEL and COND statements. Referring to the examples given in figure 1, it can be seen that each of these structure types consists of a series of one or more clauses delimited by the "<< and >>" words, and enclosed within the appropriate structure defining words:

CASE ... ENDCASE
SEL ... ENDSSEL
or, COND ... ENDCOND

Each can have an optional OTHERWISE clause which is executed if none of the other clauses is executed.

These structure types differ in how a given clause is selected for execution; thus the description of each type which follows will try to elucidate their difference.

The COND structure is a more readable syntax for a series of nested IF...ELSE...THEN statements. The COND structure consists of a series of clauses with explicitly specified conditions and associated actions which are executed if the condition is satisfied. Only the first clause whose condition is met is executed in a given execution of the structure. The integer on the top of the parameter stack is destroyed after execution. The TEST-COND definition shown in figure 1 is an example of the syntax of this structure.

The SEL structure is similar to the COND structure except that it uses an implicit test for equality to an explicitly specified integer value. Thus when the top of the parameter stack value matches that used within the SEL clause, the associated action is taken. As with the COND statement, only the first clause selected will be executed in a single pass through the structure. Additionally, the integer value tested is removed from the top of the stack after execution. An example of this structure is the TEST-SEL definition shown in figure 1.

The CASE structure is in turn similar to the SEL structure except that it uses both an implicit test for equality, and an implicit numbering of the case clauses, starting with 1 for the first clause. Thus an explicit test value does not have to be specified. In operation, for example, a value of three on the top of the parameter stack would cause execution of the third clause in a CASE statement, if it exists. Note that the CASE value on the top of the parameter stack is dropped after each pass through the structure.

Compiler Operation

The words "<<, WHEN, and >>" are used in common by all three types of structures; thus these words' compiling operations are dependent on the type of structure being used. This "type" information is determined by the integer on the top of the parameter stack at compile time—which is
set in turn by the words: CASE, SEL, or COND. These structure defining words each put two integer values on the stack. The next to top of the stack value is a flag value of zero which is used by the structure terminating words (ENDSEL, etc.) when they link up branch addresses. The top of stack value reflects the type of structure being used as summarized here:

-2 COND structure
-1 SEL structure
>0 CASE structure; this integer is actually the value of the previous CASE clause which was compiled.

The <<, WHEN, and >> words thus analyze the top of stack value to determine what words are to be compiled into the new word's parameter list. For example, WHEN for a SEL structure compiles the words OVER = and IF into the new word's definition.

The examples of the structures in figure 1 illustrate their respective syntaxes. Figures 2 through 4 are outputs from a FORTH debugger (decompiler) which emphasize the different compilations of <<, WHEN, and >> for each type of structure. (Note that the results of the compilation process are listed to the left, while the corresponding high level compiler words are at the right.) By studying the definitions of these structural words in figure 5 in conjunction with the examples and the debugger outputs, operation should be easily adapted to other FORTH systems.
Dave Lion announced availability of his 8600 assembler in FORTH occupying 1.5 Kbytes of 4 screens.

Tom Zimmer announced availability of his Tiny Pascal in FORTH; Ragsdale again lauded Tom's effort as a benchmark (cf., MEETING REPORT, FD vol. 11 No. 3, p. 59).

Martin Schaaf announced committee formation for specifying a FORTH machine's hardware.

Henry Laxen of ORTHOCODE Corp. made freely available a FORTH "WORDSTAR"-styled Editor and announced sale of GOING FORTH, the tutorial package on 8" disk by CREATIVE SOLUTIONS.

Eric Welch, the FORTH Programming Team Manager for FRIENDS-AMIS' pocket computer project, gave an in-depth description of his job. A philosophy of team organization and control was graphed and an iterative planning strategy delineated. Some problems encountered and solved by this management strategy included:

- wheel-reinvention, duplication and redundancy prevention
- tool development (much effort was spent on tracers, patches, simulators, target compiler, breakpoints and documentation and its maintenance)
- style adherence (readability and maintainability) in development and documentation
- programming environment (which, in FORTH, is relatively worse due to newness and inexperience)—here the solution entails the project manager's close involvement and intense team interaction
- accountability of time spent at each level of the plan

---

MEETINGS
NORTHERN CALIFORNIA
9/27/80

---

FORTH DIMENSIONS II/4
How to form a FIG Chapter:

1. You decide on a time and place for the first meeting in your area. (Allow about 8 weeks for steps 2 and 3.)

2. Send to FIG in San Carlos, CA a meeting announcement on one side of 8-1/2 x 11 paper (one copy is enough). Also send list of ZIP numbers that you want mailed to (use first three digits if it works for you).

3. FIG will print, address and mail to members with the ZIP's you want from San Carlos, CA.

4. When you've had your first meeting with 5 or more attendees then FIG will provide you with names in your area. You have to tell us when you have 5 or more.

Northern California
4th Saturday FIG Monthly Meeting, 1:00 p.m., at Liberty House Department Store, Hayward, CA. FORML Workshop at 10:00 a.m.

Southern California

FIGGRAPH
11/15/80 FORTH for computer graphics. 2:00 p.m. at Stanford Medical School, #M-112 at Palo Alto, CA.

12/13/80

Massachusetts
3rd Wednesday MMSFORTH Users Group, 7:00 p.m., Cochituate, MA. Call Dick Miller at (617) 653-6136 for site.

San Diego
Thursdays FIG Meeting, 12:00 noon. Call Guy Kelly at (714) 268-3100 x 4784 for site.

Seattle
Various times Contact Chuck Pliske or Dwight Vandenburg at (206) 542-8370.

Potomac
Various times Contact Paul van der Eijk at (703) 354-7443 or Joel Shprentz at (703) 437-9218.

Texas
Various times Contact Jeff Lewis at (713) 729-3320 or John Earls at (214) 661-2928 or Dwayne Gustaus at (817) 387-6976. John Hastings (512) 835-1918

Arizona
Various times Contact Dick Wilson at (602) 277-6611 x 3257.

Oregon
Various times Contact Ed Krammerer at (503) 644-2688.

New York
Various times Contact Tom Jung at (212) 746-4062.

Detroit
Various times Contact Dean Vieau at (313) 493-5105.

Japan
Various times Contact Mr. Okada, President, ASR Corp. Int'l, 3-15-8, Nishi-Shimbashi Manato-ku, Tokyo, Japan.

Publishers Note:

Please send notes (and reports) about your meetings.