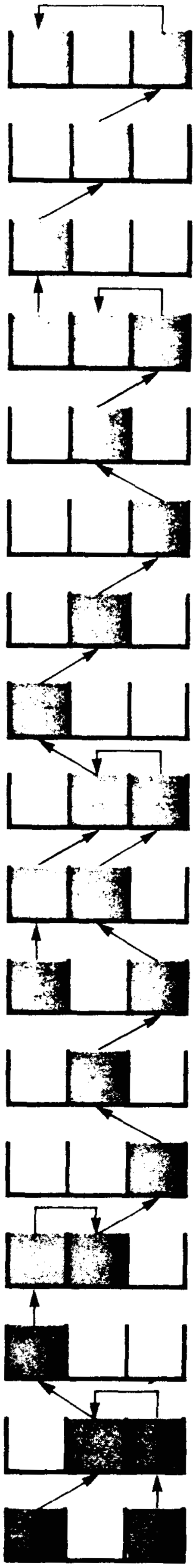


FORTH DIMENSIONS

FORTH INTEREST GROUP
P.O. Box 1105
San Carlos, CA 94070

Volume II
Number 4
Price $2.00

INSIDE

FORTH DIMENSIONS

Published by Forth Interest Group

Volume II No. 4 November/December 1980

Publisher Roy C. Martens

Subscription to FORTH DIMENSIONS is free with membership in the Forth Interest Group at $12.00 per year ($15.00 overseas). For membership, change of address and/or to submit material, the address is:

Forth Interest Group
P.O. Box 1105
San Carlos, CA 94070

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.

PUBLISHER'S COLUMN

We're deep into the planning and arrangements for the FIG Convention and the FORML Conference. If you haven't made your reservations, call right away, we might be able to get you into the FORML Conference or the Convention Banquet. Plan on coming to the Convention anyway. Remember the dates and places are:

FORML Conference, November 26, 27, & 28
Asilomar, CA

FIG Convention, November 29
Villa Hotel, San Mateo, CA

The other big news! FORTH-79 STANDARD is available!!! Call (415) 962-8653 or send in your order, today! $10.00!

Many publications are printing information about FORTH. We don't get them all, so please send in copies so we can thank the editors and add to our collection.

FIG had a booth at the Mini/Micro show and much interest was generated among attendees which carried over into a number of manufacturers that were exhibiting.

Membership is fast approaching 2,000. We now have members all over the world including the People's Republic of China and Yugoslavia. See the listings of meetings for information about how you can form a FIG chapter. Just a few easy steps and you'll have a time and place to share information.

Look forward to seeing everyone at the FORML Conference and the FIG Convention.

Roy Martens

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 data base 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 microinstruction. 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 enviroment. 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.

*Cyblok is a registered trademark of Functional Automation/Gould Inc.

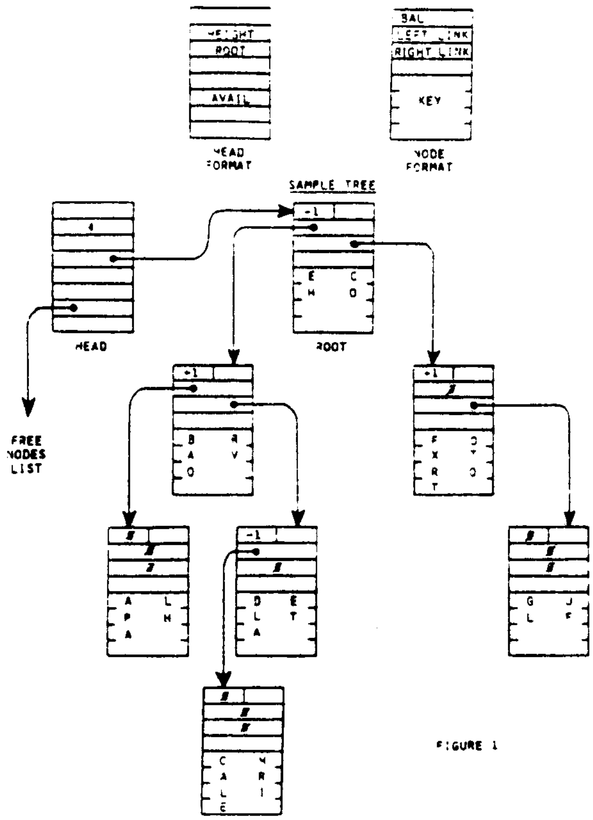


FIGURE 1

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.

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

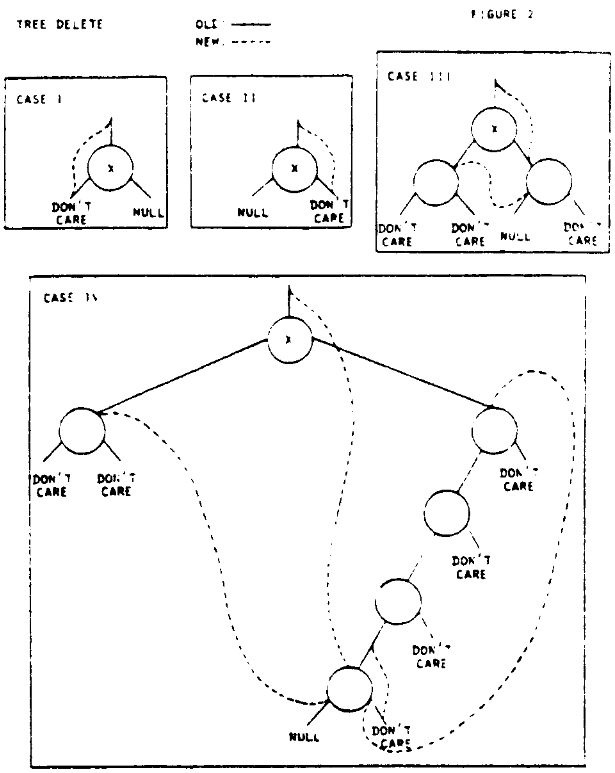
A deletion algorithm for binary trees, and the steps required to adapt this algorithm to balanced trees are provided by Knuth (The Art of Computer Programming, Vol. 3, Sections 6.2.2 and 6.2.3). The details of the balanced tree deletion algorithm are presented here, but first a review of binary tree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

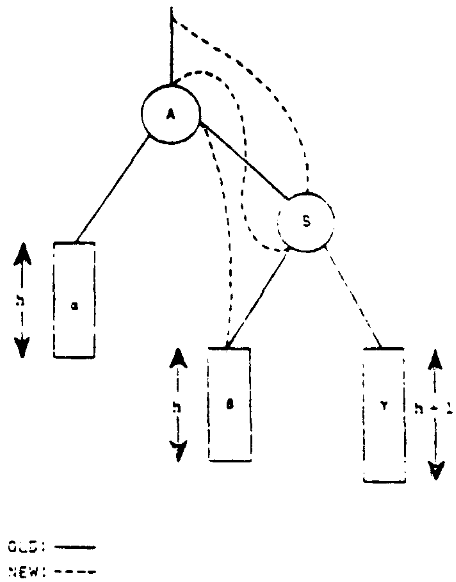
(10a) Bal(A) = Bal(B) (Figure 3);
Set Bal(A) and Bal(B) to 0.
(single rotation) -
Set Link(-f.k , A) to Link(f.k , B).
Set Link(f.k , B) to A.
Top(N.k , f.k), set Link(f.k , N.k) to B.
Go to (8) taking one more step back along the path.

(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.

FIGURE 3

REBALANCE

CASE I (TWO SITUATIONS - REFLECT DIAGRAM LEFT/RIGHT)



NEW BALANCE
A 0
B 0

NEW SUBROOT B

KEEP FIXING...

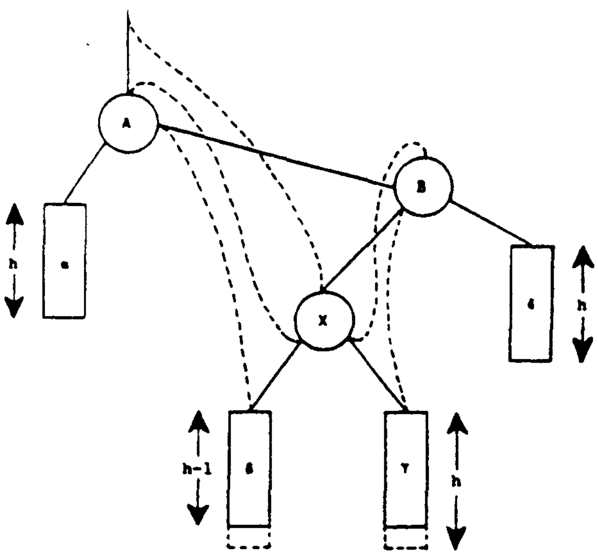
(10c) Bal(B) = 0 (Figure 5);
Set Bal(B) to -Bal(A).
(single rotation) -
Set Link(-f.k , A) to Link(f.k , B).
Set Link(f.k , B) to A.
Top(N.k , f.k), set Link(f.k , N.k) to B.
Go to (11) terminating the algorithm.

(11) Deallocate path stack. Done!

FIGURE 4

REBALANCE

CASE II (TWO SITUATIONS - REFLECT DIAGRAM LEFT/RIGHT)



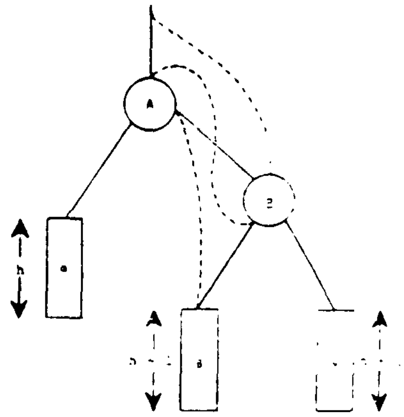
OLD: ——
NEW: ----

NEW BALANCE	BAL(X) = BAL (A)	OTHERWISE
A	-BAL(X)	0
B	0	-BAL(X)
X	0	0
NEW SUBROOT	X	
KEEP FIXING...		

FIGURE 5

REBALANCE

CASE III (TWO SITUATIONS - REFLECT DIAGRAM LEFT/RIGHT)



OLD: ——
NEW: ----

NEW BALANCE	
A	BAL(A)
B	-BAL(A)
NEW SUBROOT	B
DONE!	

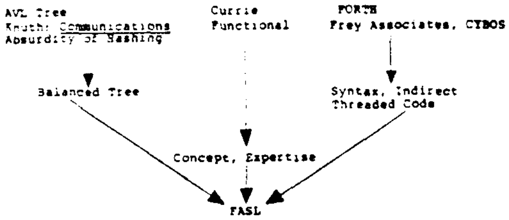
Implementing the Algorithm in FASL

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 (W0, 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.

```
( 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 + @ ;
: RLNK@ 4 + @ ;
( 2,0 )
: LLNK! 2 + ! ;
: RLNK! 4 + ! ;

( 1,0 )
: PUSH 4 'D @ ! 2 4 'D +! ;
( 0,1 )
: POP OFFFE 4 'D +! 4 'D @ @ ;
( 1,1 )
: RCRUMB DUP PUSH OFFFF PUSH ;
: LCRUMB DUP PUSH SUCCEED PUSH :

( 3,2 )
: SINGLROT OVER2 LTZ?
    IF DUP RLNK@ OVER2 LLNK!
      SWAP OVER RLNK!
    ELSE DUP LLNK@ OVER2 RLNK!
      SWAP OVER LLNK!
    FI ;

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

: ROTCASE3 OVER C@ NEG OVER C!
    SINGLROT ;

: ROTCASE2 OVER2 OVER2 OVER2 OVER2 - 3 + @
    SINGLROT
    SWAP NEG SWAP OVER2 SWAP
    SINGLROT SWAPDROP
    OVER2 C@ OVER C@ -
      IF DUP C@ NEG SROT C! FAIL SROT C!
      ELSE FAIL SROT C! DUP C@ NEG SROT C! FI
    FAIL OVER C!
    SWAPDROP FAIL SWAP;

: MOVELZR + DUP 6 'D ! @ ;

( 2,0 )
( <sname> <tname> )
: DROPLEAF
    30 PROC
    8 'D 4 'D !
    SWAP OVER
    RCRUMB
    4 MOVELZR
    WHILE DUP
      IF OVER OVER 8 + SLT? DUP
        IF OVER 10 + W0 @ -
        ELSE W1 @ 1 - C@ FI
      ELSE FAIL FAIL FI
    CONTINUE
      IF LCRUMB 2
      ELSE RCRUMB 4 FI
      MOVELZR
    WHILEND
    DROP
    SWAPDROP
    DUP
      IF DUP RLNK@
        IF DUP LLNK@
          IF DUP RLNK@ DUP LLNK@
            IF 4 'D @ 2 'D ! RCRUMB
              DUP
              REPEAT LCRUMB SWAPDROP DUP LLNK@ DUP LLNK@ ZERO?
              UNTIL
              OVER2 LLNK@ OVER LLNK!
              DUP RLNK@ OVER2 LLNK!
              OVER2 RLNK@ OVER RLNK!
              SWAPDROP
              DUP 2 'D @ :
            ELSE OVER LLNK@ OVER LLNK!
              RCRUMB
            FI
            OVER C@ OVER C!
          ELSE DUP RLNK@ FI
        ELSE DUP LLNK@ FI
        6 'D @ !
        OVER 0A + @ OVER RLNK! OVER 0A + !
        REPEAT
          POP POP OVER2 OVER SWAP -
            IF DUP C@ DUP
              IF OVER2 + OFF AND
                IF OVER 3 + OVER + @ DUP C@
                  IF OVER2 OFF AND OVER C@ -
                    IF ROTCASE2
                    ELSE ROTCASE1 FI
                  ELSE ROTCASE3 FI
                  POP POP DUP PUSH SWAP DUP PUSH - 3 + !
                ELSE FAIL SWAP C! DROP FAIL FI
              ELSE DROP C! SUCCEED FI
            ELSE 2 + +! SUCCEED FI
          UNTIL
        ELSE DROP FI
      DROP
      ENDPROC
    ;
;S
```



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 exerpts 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:

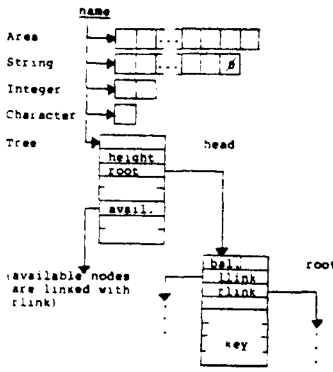
BTSIUP	E151	tree initialization
FINDIT	E168	tree search
BTSI	E17D	tree insertion
DEFNOT	E660	default tree overflow subroutine
K	D0	key for search & insertion, 8 bytes
HEAD	C2	pointer to tree
AVAIL	C4	pointer to free nodes list
VTV	C0	overflow transfer vector

```
                            ; BALANCED TREE SEARCH AND INSERT

                            ; DIRECT MEMORY DATA DECLARATIONS

00C0          VTV:    EQU 0C0        ; TREE OVERFLOW TRANSFER VECTOR TO SUBR ERROR HANDLI
00C2          HEAD:   EQU VTV+2      ; POINTER TO TREE DESCRIPTER NODE
00C4          AVAIL:  EQU HEAD+2     ; POINTER TO ROOT OF AVAILABLE NODES LIST
                                     ; ********************************************
                                     ; THE ABOVE THREE ITEMS ARE INPUTS TO BTSI
                                     ; VTV <- ADDRESS OF ERROR HANDLING SUBROUTINE
                                     ;        FOR OVERFOW OF ALLOTTED NODES
                                     ; HEAD <- POINTER TO TREE DESCRIPTER NODE, OR START
                                     ;        OF FREE SPACE FOR INITIALIZATION "BTSIUP"
                                     ; AVAIL <- POINTER TO LIST OF FREE NODES, OR END OF
                                     ;        FREE SPACE PLUS ONE FOR "BTSIUP"
                                     ; ****************
                                     ; BTSIUP USES HEAD AND AVAIL TO CREATE A NUL BALANCI
                                     ; TREE AND A FREE NODES LIST.  AVAIL IS MODIFIED BY
                                     ; BTSIUP AND ALLOCT.
                                     ; ********************************************
00C6          T:      EQU AVAIL+2
00C8          S:      EQU T+2
00CA          R:      EQU S+2
00CC          Q:      EQU R+2
00CE          P:      EQU Q+2
00D0          K:      EQU P+2        ; KEY, EIGHT BYTES

                      ; NODE FORMAT
                      ;       NODE(0) BALANCE 1
                      ;       NODE(1) FLAG    1
                      ;       NODE(2) LEFTLINK 2
                      ;       NODE(4) RIGHTLINK 2
                      ;       NODE(6) VALUE   2
                      ;       NODE(8) KEY     8

                      ; FINDING AND INSERTING
E11F 96 C8    NEXT:   LDAA S         ; S -> R<Q>
E121 DE CC            LDX Q
E123 A7 04            STAA X 04
E125 E7 05            STAB X 05

E127 DE C8            LDX S          ; INCREMENT S BY 16.
E129 DF CC    INIT:   STX Q          ; Q IS PARENT OF S
E12B D6 C9            LDAB S+1
E12D CB 10            ADDB #10       ; THIS IS THE SIZE OF A NODE
E12F 24 03            BCC SOK
E131 7C 00C8          INC S
E134 D7 C9    SOK:    STAB S+1       ; NOTE THAT B ALWAYS HAS S+1

E136 DE C8            LDX S          ; CHECK AGAINST LIMIT
E138 9C C4            CPX AVAIL
E13A 26 E3            BNE NEXT

E13C DE CC            LDX Q          ; SIGNAL END OF LIST
E13E 6F 04            CLR X 04
E140 6F 05            CLR X 05

E142 96 C6            LDAA T         ; T -> R<HEAD> , SAVE POINTER TO ROOT
E144 D6 C7            LDAB T+1
E146 DE C2            LDX HEAD
E148 A7 04            STAA X 04
E14A E7 05            STAB X 05

E14C DE CA            LDX R          ; R -> AVAIL , SAVE POINTER TO FREE LIST
E14E DF C4            STX AVAIL
E150 39               RTS            ; END OF BTSI INITIALIZATION OF FREE LIST
                                     ; AND NULL TREE

                      ; BTSI INITIALIZATION

E151 4F       BTSIUP: CLRA
E152 5F               CLRB           ; MAKE A TREE DESCRIPTER, A NULL ROOT
E153 DE C2            LDX HEAD       ; A FREE LIST BASED ON PARAMETERS
E155 8D 5A            BSR SAXBI      ;      HEAD    START OF FREE SPACE
E157 DF C6            STX T          ;      AVAIL   END OF FREE SPACE P
E159 8D 56            BSR SAXBI      ; SAXBI CLEARS A NODE SINCE A=B=0
E15B DF C8            STX S
E15D DF CA            STX R          ; POINTER TO ROOT IN T
                                     ; POINTER TO AVAIL IN R
E15F 20 C8            BRA INIT

E161 EE 02    MOVLF:  LDX X 02       ; ALONG LEFTLINK
E163 26 0D            BNE FIND

E165 4F       NOTFND: CLRA
E166 4C               INCA           ; FAILURE !! CC Z = 0
E167 39               RTS

                      ; LEXPPC -->

E168 DE C2    FINDIT: LDX HEAD
E16A EE 04            LDX X 04       ; ROOT OF TREE
E16C 20 04            BRA FIND

E16E EE 04    MOVRF:  LDX X 04
E170 27 F3            BEQ NOTFND     ; MOVE ALONG RIGHTLINK

E172 DF CE    FIND:   STX P
E174 8D 48            BSR KMP
E176 22 F6            BHI MOVRF
E178 26 E7            BNE MOVLF
E17A 39               RTS            ; SUCCESS !! CC Z = 1

E17B 84 FF            ANDA #0FF      ; NOPE

                      ; BTSI BALANCED TREE SEARCH AND INSERT

E17D DE C2    BTSI:   LDX HEAD       ; HEAD -> T
E17F DF C6            STX T

E181 EE 04            LDX X 04       ; R<HEAD> -> S , S POINTS TO REBALANCE
E183 DF C8            STX S
E185 20 21            BRA SEARCH

E187 DE C0    OVRFLW: LDX VTV        ; TREE OVERFLOW TRANSFER VECTOR
E189 AD 00            JSR X 00

E18B DE C4    ALLOCT: LDX AVAIL      ; ALLOCATE A FREE NODE TO THE TREE
E18D 27 F8            BEQ OVRFLW     ; CHECK FOR EMPTY FREE LIST

E18F DF CC            STX Q          ; Q <- AVAIL

E191 EE 04            LDX X 04       ; R<AVAIL> -> AVAIL
E193 DF C4            STX AVAIL
E195 DE CE            LDX P          ; SETUP PARAMETERS FOR CALLER
E197 96 CC            LDAA Q
E199 D6 CD            LDAB Q+1
E19B 39               RTS

E19C A6 00    COMMOV: LDAA X 00      ; CHECK BALANCE FACTOR
E19E 27 06            BEQ FINMOV

E1A0 DF C8            STX S          ; Q -> S
E1A2 DE CE            LDX P          ; P -> T
E1A4 DF C6            STX T
E1A6 DE CC    FINMOV: LDX Q          ; Q

E1A8 DF CE    SEARCH: STX P          ; -> P

E1AA 8D 12            BSR KMP        ; K - K<P>
E1AC 22 4D            BHI MOVR
E1AE 26 3D            BNE MOVL
E1B0 39               RTS            ; SUCCESS!!!
                                     ; RETURN WITH CC Z = 1

E1B1 8D 00    SAXBI:  BSR SAX4I
E1B3 8D 00    SAX4I:  BSR SAX2I
E1B5 8D 00    SAX2I:  BSR SAXINX
E1B7 A7 00    SAXINX: STAA X 00      ; STORE ACCUMULATORS INDEXED
E1B9 E7 01            STAB X 01
E1BB 08               INX
E1BC 08               INX            ; POST INCREMENTED
E1BD 39               RTS

                      ; KEY COMPARE SUBROUTINE

E1BE 96 D0    KMP:    LDAA K         ; K - KEY<X>
E1C0 A1 08            CMPA X 08
E1C2 26 28            BNE RTN        ; RETURN IF NOT EQUAL

E1C4 96 D1            LDAA K+1
E1C6 A1 09            CMPA X 09
E1C8 26 22            BNE RTN

E1CA 96 D2            LDAA K+2
E1CC A1 0A            CMPA X 0A
E1CE 26 1C            BNE RTN

E1D0 96 D3            LDAA K+3
E1D2 A1 0B            CMPA X 0B
E1D4 26 16            BNE RTN

E1D6 96 D4            LDAA K+4
E1D8 A1 0C            CMPA X 0C
E1DA 26 10            BNE RTN

E1DC 96 D5            LDAA K+5
E1DE A1 0D            CMPA X 0D
E1E0 26 0A            BNE RTN

E1E2 96 D6            LDAA K+6
E1E4 A1 0E            CMPA X 0E
E1E6 26 04            BNE RTN

E1E8 96 D7            LDAA K+7
E1EA A1 0F            CMPA X 0F
E1EC 39       RTN:    RTS            ; DONE COMPARE OF EIGHT BYTES

E1ED EE 02    MOVL:   LDX X 02       ; L<P> -> Q
E1EF DF CC            STX Q
E1F1 26 A9            BNE COMMOV     ; CONTINUE DOWN THE LEFT LINK

E1F3 8D 96            BSR ALLOCT     ; DEAD END , ALLOCATE NEW NODE
E1F5 A7 02            STAA X 02      ; L.LINK TO P
E1F7 E7 03            STAB X 03      ; I.E. Q -> L<P>
E1F9 20 0C            BRA INSERT

E1FB EE 04    MOVR:   LDX X 04       ; R<P> -> Q
E1FD DF CC            STX Q
E1FF 26 9B            BNE COMMOV     ; CONTINUE DOWN THE RIGHT LINK

E201 8D 88            BSR ALLOCT     ; DEAD END , ALLOCATE NEW NODE
E203 A7 04            STAA X 04      ; R.LINK TO P
E205 E7 05            STAB X 05      ; I.E. Q -> R<P>

E207 DE CC    INSERT: LDX Q          ; INITIALIZE THE NEW NODE
E209 4F               CLRA
E20A 5F               CLRB           ; CLEAR B<Q>, L<Q>, R<Q>, D<Q>, F<Q>
E20B 8D A6            BSR SAX4I

E20D 96 D0            LDAA K
E20F D6 D1            LDAB K+1       ; K -> K<Q>
E211 8D A4            BSR SAXINX

E213 96 D2            LDAA K+2
E215 D6 D3            LDAB K+3
E217 8D 9E            BSR SAXINX

E219 96 D4            LDAA K+4
E21B D6 D5            LDAB K+5
E21D 8D 98            BSR SAXINX

E21F 96 D6            LDAA K+6
E221 D6 D7            LDAB K+7
E223 8D 92            BSR SAXINX
                            ; ******************************
```

```
                        ; ADJUST BALANCE FACTORS ...
E225 DE C8   ADJ0:      LDX S           ; K - K(S)
E227 8D 95              BSR KMP
E229 22 06              BHI ADJ1
E22B C6 FF              LDAB #0FF       ; FLAG LT ( -1 -> A )
E22D EE 02              LDX X 02        ; L(S) ...
E22F 20 04              BRA ADJ2
E231 C6 01   ADJ1:      LDAB #01        ; FLAG GE ( 1 -> A )
E233 EE 04              LDX X 04        ; R(S) ...
E235 DF CA   ADJ2:      STX R           ; ... -> R
E237 20 10              BRA ADJ5        ; ENTER LOOP
E239 6F 00   ADJ3:      CLR X 00        ; 0 -> B(P)
E23B 8D 81              BSR KMP         ; K - K(P)                TIGHT!!!!!
E23D 22 06              BHI ADJ4
E23F 6A 00              DEC X 00        ; -1 -> B(P)
E241 EE 02              LDX X 02        ; L(P) ...
E243 20 04              BRA ADJ5
E245 6C 00   ADJ4:      INC X 00        ; 1 -> B(P)
E247 EE 04              LDX X 04        ; R(P) ...
E249 DF CE   ADJ5:      STX P           ; ... -> P
E24B 9C CC              CPX Q           ; UNTIL WE REACH Q
E24D 26 EA              BNE ADJ3
                                        ; *************

                        ; BALANCING ACT ...
E24F DE C8   BAL0:      LDX S           ; CHECK BALANCE FACTOR OF S
E251 A6 00              LDAA X 00
E253 26 07              BNE BAL1
E255 E7 00              STAB X 00       ; A -> B(S)
E257 DE C2              LDX HEAD        ; INCREMENT HEIGHT OF TREE
E259 6C 03              INC X 03
E25B 39                 RTS             ; FAIL!!!!
                                        ; RETURN CC Z = 0
E25C E1 00   BAL1:      CMPB X 00       ; CHECK B(S) AGAINST A
E25E 27 05              BEQ BAL2
E260 4F                 CLRA
E261 A7 00              STAA X 00       ; 0 -> B(S)
E263 4C                 INCA
E364 39                 RTS             ; FAIL!!!!
                                        ; RETURN CC Z = 0
E265 DE CA   BAL2:      LDX R           ; TREE NEEDS BALANCING
E267 5D                 TSTB
E268 2B 46              BMI BAL3
E26A E1 00              CMPB X 00       ; CHECK BALANCE FACTOR OF R
E26C 27 62              BEQ SROTL
                                        ; **********************

                        ; DOUBLE ROTATE LEFT ...
E26E EE 02   DROTL:     LDX X 02        ; L(R) -> P
E270 DF CE              STX P
E272 A6 04              LDAA X 04       ; R(P) -> L(R)
E274 E6 05              LDAB X 05
E276 DE CA              LDX R
E278 A7 02              STAA X 02
E27A E7 03              STAB X 03
E27C 6F 00              CLR X 00        ; 0 -> B(R)
E27E 96 CA              LDAA R          ; R -> R(P)
E280 D6 CB              LDAB R+1
E282 DE CE              LDX P
E284 A7 04              STAA X 04
E286 E7 05              STAB X 05
E288 A6 02              LDAA X 02       ; L(P) -> R(S)
E28A E6 03              LDAB X 03
E28C DE C8              LDX S
E28E A7 04              STAA X 04
E290 E7 05              STAB X 05
E292 6F 00              CLR X 00        ; 0 -> B(S)
E294 96 C8              LDAA S          ; S -> L(P)
E296 D6 C9              LDAB S+1
E298 DE CE              LDX P
E29A A7 02              STAA X 02
E29C E7 03              STAB X 03
E29E E6 00              LDAB X 00       ; CHECK BALANCE FACTOR OF P
E2A0 27 48              BEQ TUPLX
E2A2 2B 06              BMI SOML
E2A4 6F 00              CLR X 00        ; 0 -> B(P)
E2A6 DE C8              LDX S
E2A8 20 7E              BRA TUP0        ; -1 -> B(S)           V TIGHT !!!
E2AA 6F 00   SOML:      CLR X 00        ; 0 -> B(P)
E2AC DE CA              LDX R
E2AE 20 3C              BRA TUP1LX      ; 1 -> B(R)

                        ; ******
E2B0 E1 00   BAL3:      CMPB X 00
E2B2 26 3A              BNE DROTR       ; CHECK BALANCE FACTOR OF R

                        ; SINGLE ROTATE RIGHT ...
E2B4 DF CE   SROTR:     STX P           ; R -> P
E2B6 6F 00              CLR X 00        ; 0 -> B(R)
E2B8 A6 04              LDAA X 04
E2BA E6 05              LDAB X 05       ; R(R) -> L(S)
E2BC DE C8              LDX S
E2BE A7 02              STAA X 02
E2C0 E7 03              STAB X 03
E2C2 6F 00              CLR X 00        ; 0 -> B(S)
E2C4 96 C8              LDAA S          ; S -> R(R)
E2C6 D6 C9              LDAB S+1
E2C8 DE CA              LDX R
E2CA A7 04              STAA X 04
E2CC E7 05              STAB X 05
E2CE 20 62              BRA TUCHUP

                        ; SINGLE ROTATE LEFT ..
E2D0 DF CE   SROTL:     STX P           ; R -> P
E2D2 6F 00              CLR X 00        ; 0 -> B(R)
E2D4 A6 02              LDAA X 02
E2D6 E6 03              LDAB X 03       ; L(R) -> R(S)
E2D8 DE C8              LDX S
E2DA A7 04              STAA X 04
E2DC E7 05              STAB X 05
E2DE 6F 00              CLR X 00        ; 0 -> B(S)
E2E0 96 C8              LDAA S          ; S -> L(R)
E2E2 D6 C9              LDAB S+1
E2E4 DE CA              LDX R
E2E6 A7 02              STAA X 02
E2E8 E7 03              STAB X 03
E2EA 20 46   TUPLX:     BRA TUCHUP
E2EC 20 42   TUP1LX:    BRA TUP1

                        ; DOUBLE ROTATE RIGHT ...
E2EE EE 04   DROTR:     LDX X 04        ; R(R) -> P
E2F0 DF CE              STX P
E2F2 A6 02              LDAA X 02       ; L(P) -> R(R)
E2F4 E6 03              LDAB X 03
E2F6 DE CA              LDX R
E2F8 A7 04              STAA X 04
E2FA E7 05              STAB X 05
E2FC 6F 00              CLR X 00        ; 0 -> B(R)
E2FE 96 CA              LDAA R          ; R -> L(P)
E300 D6 CB              LDAB R+1
E302 DE CE              LDX P
E304 A7 02              STAA X 02
E306 E7 03              STAB X 03
E308 A6 04              LDAA X 04       ; R(P) -> L(S)
E30A E6 05              LDAB X 05
E30C DE C8              LDX S
E30E A7 02              STAA X 02
E310 E7 03              STAB X 03
E312 6F 00              CLR X 00        ; 0 -> B(S)
E314 96 C8              LDAA S          ; S -> R(P)
E316 D6 C9              LDAB S+1
E318 DE CE              LDX P
E31A A7 04              STAA X 04
E31C E7 05              STAB X 05
E31E E6 00              LDAB X 00       ; CHECK BALANCE FACTOR OF P
E320 27 10              BEQ TUCHUP
E322 2B 08              BMI DAUL
E324 6F 00              CLR X 00        ; 0 -> B(P)
E326 DE CA              LDX R           ; -1 -> B(R)
E328 6A 00   TUP0:      DEC X 00
E32A 20 06              BRA TUCHUP
E32C 6F 00   DAUL:      CLR X 00        ; 0 -> B(P)
E32E DE C8              LDX S           ; 1 -> B(S)
E330 6C 00   TUP1:      INC X 00

                        ; TOUCHUP ...
E332 96 CE   TUCHUP:    LDAA P          ; PREPARATION ...
E334 D6 CF              LDAB P+1
E336 DE C6              LDX T
E338 EE 04              LDX X 04        ; R(T) - S , COMPARE
E33A 9C C8              CPX S
E33C 27 09              BEQ TUP4
E33E DE C6              LDX T           ; P -> L(T)
E340 A7 02              STAA X 02
E342 E7 03              STAB X 03
E344 CA FF              ORAB #0FF
E346 39                 RTS             ; FAIL !!!!
                                        ; RETURN CC Z = 0
E347 DE C6   TUP4:      LDX T           ; P -> R(T)
E349 A7 04              STAA X 04
E34B E7 05              STAB X 05
E34D CA FF              ORAB #0FF
E34F 39                 RTS             ; FAIL !!!!
                                        ; RETURN CC Z = 0

                        ; ++++++++++++++++++++++++++++++++++++++++++++++++++++++++

E660                    ORG 0E660
E660 31      DEFNOT:    INS
E661 31                 INS
E662 31                 INS
E663 31                 INS
E664 4F                 CLRA            ; NO MORE ROOM IN TREE -> FAIL!
E665 39                 RTS
E666 01                 NOP
E667 01                 NOP
```

FASL HANDY REFERENCE

PREDEFINED DATA TYPES



name	size
Area	zero to n bytes
String	one to n bytes
Integer	two bytes
Character	one byte
Tree	thirty-two to n bytes

(each node is sixteen bytes)

(balance is one byte)
(each link is two bytes)
(key is eight bytes)
(three user defined bytes)

Stack inputs and outputs are shown; top of stack on right.

Operand Key:

n	two byte number
s	two byte signed number
u, addr, to, from	two byte unsigned number
b	one byte number or character
f	two byte boolean flag (zero or on
addr?	conditionally present addr
d	four byte signed number

(Digits are sometimes appended to these operand names.)
(Unless unsigned operands are indicated, arithmetic operations are twos complement.)

STACK MANIPULATION

DUP	(n -- n n)	Duplicate top of stack.
DROP	(n --)	Throw away top of stack.
SWAP	(n1 n2 -- n2 n1)	Reverse top two stack items.
OVER	(n1 n2 -- n1 n2 n1).	Make copy of second item on top.
OVER2	(n1 n2 n3 -- n1 n2 n3 n1)	Make copy of third item on top.
SROT	(n1 n2 n3 -- n2 n3 n1)	Rotate third item to top.
SWAPDROP	(n1 n2 -- n2)	Throw away second item on top.
DROP2	(n n --)	Throw away top two.
DROP3	(n n n --)	Throw away top three.
RPUSH	(n --)	Move top item to return stack.
RPOP	(-- n)	Retrieve top item from return stack.
'R	(s -- addr)	Compute address of sth byte on return stack.
'S	(s -- addr)	Compute address of sth byte on top (2 'S @ = OVER).

ARITHMETIC AND LOGICAL

+	(s1 s2 -- sum)	Add.
-	(s1 s2 -- difference)	Subtract (s1 - s2).
*	(s1 s2 -- product)	Multiply.
/	(s1 s2 -- quotient)	Divide (s1 ÷ s2).
MOD	(s1 s2 -- modulo)	Modulo (s1 mod s2).
MULE	(s1 s2 -- d)	Multiply extended.
DIVE	(d s -- quot mod)	Divide extended.
DIVMOD	(s1 s2 -- quot mod)	Divide modulus.
SEXT	(s -- d)	Sign Extend.
NEG	(s -- negation)	Negate.
ABS	(s -- absolute)	Absolute Value.
MIN	(s1 s2 -- min)	Minimum.
MAX	(s1 s2 -- max)	Maximum.
AND	(u1 u2 -- intersection)	Bitwise And.
OR	(u1 u2 -- conjunction)	Bitwise Or.
XOR	(u1 u2 -- disjunction)	Bitwise Exclusive Or.
NOT	(u -- complement)	Bitwise Inversion.
SUCCEED	(-- 1)	One (true).
FAIL	(-- 0)	Zero (false).
SHL	(n u -- n)	Shift Left (n, u times).
SHR	(n u -- n)	Shift Right (n, u times).
ROL	(n u -- n)	Rotate Left (n, u times).
ROR	(n u -- n)	Rotate Right (n, u times).

COMPARISON

		flag is true (one) if:
LTZ?	(s -- f)	Less than zero (s < 0)?
ZERO?	(n -- f)	Zero (n = 0)?
GTZ?	(s -- f)	Greater than zero (s > 0)?
LT?	(s1 s2 -- f)	Less than (s1 < s2)?
LE?	(s1 s2 -- f)	Less than or Equal (s1 ≤ s2)?
EQ?	(n1 n2 -- f)	Equal (n1 = n2)?
NE?	(n1 n2 -- f)	Not Equal (n1 ≠ n2)?
GE?	(s1 s2 -- f)	Greater than or Equal (s1 ≥ s2)?
GT?	(s1 s2 -- f)	Greater than (s1 > s2)?
ADDRGT?	(u1 u2 -- f)	Address Greater than? (u1 > u2)?
SLT?	(addr1 addr2 -- f)	String Less Than? (string at addr1 < string at addr2)?
SEQ?	(addr1 addr2 -- f)	Strings Equal? (string at addr1 = string at addr2)?

MEMORY

@	(addr -- n)	Replace address by contents.
!	(n addr --)	Store second item at address on top.
C@	(addr -- b)	Replace address by contents, one byte only (right justify zero padded).
C!	(b addr --)	Store right byte of second item at address on top.
+!	(n addr --)	Add second item to contents of address on top.
@SWAP	(addr1 addr2 --)	Swap contents of addr1 and addr2.
CMOVE	(from to u --)	Move u bytes in memory.
MOVE	(from to u --)	Move u double-bytes in memory.
SMOVE	(from to --)	Move string in memory.
LEAF	(addr1 addr2 -- addr? f)	Add key (string) at addr1 to tree at addr2. If f = true, then key was inserted at addr?, otherwise the key was already in tree (or tree is full).
FIND	(addr1 addr2 -- addr? f)	Locate key (string) at addr1 in tree at addr2. If f = true then key is at addr?, otherwise not found.
F$@	(addr -- b n)	Read data from tree node at addr.
F@!	(b n addr --)	Store data in tree node at addr.
'D	(s -- addr)	Compute address of nth byte in current Local Area.

CONTROL STRUCTURES

DO...LOOP	do: (end+1 start --)	Set up loop, give index range.
I	(-- index)	Place current index value on stack.
DO...+LOOP	+loop: (n --)	Like DO...LOOP except adds stack value (rather than one) to index.
IF...(true)...FI	if: (f --)	If top of stack true (non-zero), execute.
IF...(true)...ELSE...(false)...FI		Same, but if false, execute ELSE clause.
DO...IF...(true)...LOOP ELSE...(false)...EXIT FI		The EXIT in ELSE clause terminates loop prematurely. +LOOP may be used in place of LOOP, and the LOOP and EXIT words may be reversed.
REPEAT...UNTIL	until: (f --)	Loop back to REPEAT until true at UNTIL.
WHILE... CONTINUE...(true)... WHILEND... (false)...	continue: (f --)	Continue while true at CONTINUE, otherwise leave loop; WHILEND loops unconditionally.

INPUT/OUTPUT

MESS	(addr --)	Type message (string) at addr.
TYPE	(addr b --)	Type message at addr terminated by byte b.
=	(n --)	Type number on top of stack.
C=	(b --)	Type one byte number on top.
CRLF	(--)	Type a Carriage Return, Line Feed.
SP	(--)	Type a Space.
DUMP	(addr u --)	Type u bytes starting at addr.
PRTREE	(addr --)	Type tree at addr.
GETKEY	(--)	Read characters until delimiter to Global Area K.
CHECKKEY	(-- f)	True if K is non-numeric.
CONVERTK	(-- n)	Converts string at K to number.
ASK	(addr delim count --)	Read characters to addr until delimiter or count.
WORD	(addr delim --)	Read characters to addr until delimiter.

DEFINING WORDS

: xxx	(--)	Begin colon-word definition of xxx.
;	(--)	End colon-word definition.
, xxx	(addr --)	Used to name machine language operation.
GLOBAL xxx	(n --) xxx: (-- addr)	Create Global Variable xxx with initial value n; returns address when executed.
CONSTANT xxx	(n --) xxx: (-- n)	Create Constant Variable xxx with value n; returns value when executed.
AREA xxx	(n --) xxx: (-- addr)	Create Global Area xxx of size n, with no initial value; returns address when executed.
" xxx ...	(--) xxx: (-- addr)	Create Global String xxx with initial value of text typed in after xxx delimited by quote ("); returns address when executed.
TREE xxx	(n --) xxx: (-- addr)	Create Global Tree xxx of size n nodes, and initialize; returns address when executed.
TREEINIT	(addr1 addr2 --)	Initialize Tree from addr1 to addr2-1 (used for Local or preallocated Trees).
PROC...ENDPROC	proc: (n --)	Allocate/Deallocate n bytes of Local Area on return stack (only used inside colon-words).

SYSTEM & MISCELLANEOUS

LOAD...;S	load: (addr1 addr2 --)	LOAD modifies current Input pointers (addr 1 is address of input string, addr2 is address of machine level input subroutine), ;S restores previous values (uses return stack...be careful).
(	(--)	Begin Comment, delimited by right paren. (up to 8K characters are allowed).
PGMOVE	(u1 u1 --)	Block Move of 8Kbytes from page u1 to page u2.
INTO	(u --)	Block Move from Inbox to page u.
OUTOF	(u --)	Block Move from Page u to Outbox.
PEREAD	(u --)	DUMBOS Read from Outbox of Cyblok u to Inbox.
DUMBWAIT	(--)	Wait for DUMBOS command slot acknowledge.
PEMESS	(addr u --)	Send message at addr to Cyblok u.
PEWAIT	(addr u --)	Receive message from Cyblok u to addr.
PESLOT	(u -- addr)	Compute inslot address for Cyblok u.

LETTERS

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 itemlisted 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 rturn 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 aguments above. Also, in regards to speed, the CALL's and JMP's within the code statement for CASES are weak in style snce 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

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 1FUNCTION 1 OFFSET.IN CTRL.KEY
INSTALL 2FUNCTION 2 OFFSET.IN CTRL.KEY
        .
        .
        .
INSTALL 26 FUNCTION 26 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 (TT0) 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 multitasking. 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> )

TT0 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

```
     0 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 associating 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*

*Wirth, Niklaus, "Algorithms + Data Structures = Programs," Englewood Cliffs, Prentice-Hall, Inc. 1976.

```
                  ********  BLOCK 501  ********
                 EXECUTION VARIABLES AND ARRAYS

: DUMMY ;                            ( DUMMY XEQ ROUTINE )

: ()XEQ                              ( #OF VECTORS>-<>, XEQ ARRAY DEF'NG WORD )
  <BUILDS DUP ,                      ( FIRST PARAM=MAX # OF VECTORS )
  2* DUP HERE SWAP 0 FILL ALLOT      ( INIT ALL VECTORS TO 0 )
  DOES> DUP @ 3 PICK >               ( CHECK FOR MAX > INDEX )
            3 PICK 0>=               ( AND FOR INDEX >=0 )
        AND IF 2+ SWAP 2* + @ EXECUTE
          ELSE 2DROP 33 ABORT
          THEN ;

: XEQ <BUILDS ,                      ( <CODE ADDR>-<>, CREATE EXECUTION VECTOR )
  DOES> @ EXECUTE ;

                  ********  BLOCK 502  ********
              EXECUTION VARIABLES AND ARRAYS CONT'D )

        ( FOR INSTALLATION:  INSTALL <ROUTINE NAME> in <XEQ NAME> )

: INSTALL ( INSTALL <NAME> IN 'XEQ' VARIABLE -- SET VECTOR ADDR )
  ]'[ STATE @ IF COMPILE CFA ELSE CFA THEN ; IMP INSTALL

: IN ( <XEQ ADDR>-<>, IN <XEQ VAR NAME>--STORE ADDR IS XEQ VAR )
  ]'[ STATE @ IF COMPILE ! ELSE ! THEN ; IMP IN

: OFFSET.IN ( <XEQ ADDR><XEQ ARRAY WORD OFFSET+1>-<>, )
  DUP 0>= IF 1+ 2* ]'[ + ! [illegible] CAN'T USE IN COMPILE STATE )
      ELSE 2DROP THEN ;
```

MEETINGS

NORTHERN CALIFORNIA

8/23/80

Ray Dessey, a chemist from Virginia Polytechnical Institute in Blacksberg, was visiting and he described his recent trip to China. FORTH accompanied him embodied in an AIM and students at Futan University, Shanghai, got a taste of FORTH. Dr. Dessey said the University already had 3 LSI-11's with Pertec floppies. He also described Virginia Tech's teaching/research machine which is a network with 3 three terminal hosts each having 15 satellite processors. FORTH runs under an RT-11 operating system. Instrumentation simulation (a function generator + noise) is one use.

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
- FORTH MODIFICATION LABORATORY's CALL FOR PAPERS: (Programming methodology, Virtual Machine Implementation, Concurrency, Language & Compiler, Applications, and Standardization.

HELP WANTED

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

```
RUN.BENCH ' OK
1000 RUN.BENCH
STARTING
   1   2   3   5   7  11  13  17  19  23  29  31  37  41  43  47  53  59  61  67
  71  73  79  83  89  97 101 103 107 109 113 127 131 137 139 149 151 157 163 167
 173 179 181 191 193 197 199 211 223 227 229 233 239 241 251 257 263 269 271 277
 281 283 293 307 311 313 317 331 337 347 349 353 359 367 373 379 383 389 397 401
 409 419 421 431 433 439 443 449 457 461 463 467 479 487 491 499 503 509 521 523
 541 547 557 563 569 571 577 587 593 599 601 607 613 617 619 631 641 643 647 653
 659 661 673 677 683 691 701 709 719 727 733 739 743 751 757 761 769 773 787 797
 809 811 821 823 827 829 839 853 857 859 863 877 881 883 887 907 911 919 929 937
 941 947 953 967 971 977 983 991 997
FINISHED
```

INTERFACE AGE Benchmark Program
Results from the Albuquerque Group

CPU/System	Clock	Language	Execution Time
6800	.9 mhz	FORTH	6' 13"
8080 North Star DOS	1.84 mhz	FORTH	5' 49"
AM-100	2 mhz	FORTH	2' 53"
AM-100 w/ polled serial I/O	2 mhz	AlphaBasic	9' 37"
8080 Heath H8	2.2 mhz	Benton Harbor Basic	42'
8080 North Star DOS	1.84 mhz	MicroSoft Basic	21' 8"
8080 North Star DOS	1.84 mhz	MicroSoft Compiler Basic	8' 41"
8080 North Star DOS	1.84 mhz	North Star Basic	41' 13"
8080 North Star DOS	1.84 mhz	C-Basic V1.01	77'
Z-80 SuperBrain	?	C-Basic	53'
6502 Ohio Scientific	2 mhz	MicroSoft Basic	8' 25"
Z-80 North Star	4 mhz	North Star Basic	19'
Z-80 North Star w/ Floating Point Board	4 mhz	North Star	11' 25"
6800	.9 mhz	PERCOM Super Basic	73'
6800	.9 mhz	SWTP V2.3 8k Basic	81'
CYBER 176	?	FORTRAN	190 ms
CYBER 176	?	PASCAL	260 ms
CDC 6600	?	FORTRAN	1069 ms
CDC 6600	?	PASCAL	3500 ms

NOTE: Although speed improvements may be made to the basic algorithm as published in INTERFACE AGE, the programs used in the above test remained a true representation of the algorithm published in the June issue of INTERFACE AGE magazine.

```
************* FIG-FORTH ************
**** FOR THE ALPHA MICRO SYSTEM ****
fig-FORTH V 1.1
AM-FORTH VERSION A.4
Property of SIERRA COMPUTER COMPANY
617 Mark NE
Albuquerque, NM 87123
12 July 1980

SCR # 8
( INTERFACE AGE BENCHMARK PROGRAM ENTER W/ UPPER LIMIT-1 * )
: BENCH DUP 2 / 1+ SWAP ." STARTING " CR
        1 DO DUP I 1 ROT
          2 DO DROP DUP I /MOD
               DUP 0= IF DROP DROP 1 LEAVE
                      ELSE 1 = IF DROP 1
                               ELSE DUP 0 > IF DROP 1
                                            ELSE 0= IF 0 LEAVE
                                                    ENDIF
                                            ENDIF
                               ENDIF
                      ENDIF
               LOOP
          IF 4 .R ELSE DROP ENDIF
          LOOP DROP CR ." FINISHED. " ;
               -->

SCR # 9
( INTERFACE AGE BENCHMARK PROGRAM, CALLING ROUTINE        * )
: RUN.BENCH  TIME SWAP BENCH TIME
             SWAP - 60 / 60 /MOD
             CR ." ELAPSED TIME: "
              . ." MINUTES, "
              . ." SECONDS. "    ;
```

HELP WANTED

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 an 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 abitrary 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 alebraic 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.

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  ... ENDSEL
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 impliclit 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.

```
( STRUCTURE EXAMPLES - PHH - 8 22 80 )
: FIRST ;
: SECOND ;
: THIRD ;
: WHO-KNOWS? ;
: ONE ;
: NEG-THIRTY-THREE ;
: FIVE ;
: LESS-THAN-NEG-TWO ;
: GREATER-THAN-ONE ;

( STRUCTURE TESTS - CON'T - PHH - 8 22 80 )
: TEST-CASE
  CASE
     << FIRST >>
     << SECOND >>
     << THIRD >>
  OTHERWISE WHO-KNOWS?
  ENDCASE ;

: TEST-SEL
  SEL
     << 1 WHEN ONE >>
     << -33 WHEN NEG-THIRTY-THREE >>
     << 5 WHEN FIVE >>
  OTHERWISE WHO-KNOWS?
  ENDSEL ;

: TEST-COND
  COND
     << -2 < WHEN LESS-THAN-NEG-TWO >>
     << 2 >= WHEN GREATER-THAN-ONE >>
  OTHERWISE CR
  ENDCOND
;
```

FIGURE 1

```
OK DEBUG TEST-COND
TEST-COND LINKED TO 332D
: DEFINITION
3376 1439 DUP -----------------   <<
3378 0111 LIT  FFFE
337C 17DB <
337E 07FD $IF  3388 -----------   WHEN
3382 32B7 LESS-THAN-NEG-TWO
3384 0810 $ELSE  339A ----------  >>
3388 1439 DUP -----------------   <<
338A 1361 2
338C 1806 >=
338E 07FD $IF  3398 -----------   WHEN
3392 32CF GREATER-THAN-ONE
3394 0810 $ELSE  339A ----------  >>
339B 1A6B CR
339A 13BB DROP ----------------   ENDCOND
339C 01C8 $;
OK
```

FIGURE 2

```
OK DEBUG TEST-SEL
TEST-SEL  LINKED TO 32E3
: DEFINITION
332D 07B4 1
332F 142C OVER          )
3331 17BE =             )------   WHEN
3333 07FD $IF  333D     )
3337 327A ONE
3339 0810 $ELSE  3363 ---------   >>
333D 0111 LIT  FFDF
3341 142C OVER          )
3343 17BE =             )------   WHEN
3345 07FD $IF  334F     )
3349 3292 NEG-THIRTY-THREE
334B 0810 $ELSE  3363 ---------   >>
334F 0111 LIT  0005
3353 142C OVER          )
3355 17BE =             )------   WHEN
3357 07FD $IF  3361     )
335B 392E FIVE
335D 0810 $ELSE  3363 ---------   >>
3361 326F WHO-KNOWS?
3363 13BB DROP ----------------   ENDSEL
3365 01C8 $;
OK
```

FIGURE 3

```
OK DEBUG TEST-CASE
TEST-CASE  LINKED TO 32D2
: DEFINITION
32E3 0111 LIT  0001      )
32E7 142C OVER           )
32E9 17BE =              )------   <<
32EB 07FD $IF  32F5      )
32EF 3242 FIRST
32F1 0810 $ELSE  331B ---------   >>
32F5 0111 LIT  0002      )
32F9 142C OVER           )
32FB 17BE =              )------   <<
32FD 07FD $IF  3307      )
3301 3250 SECOND
3303 0810 $ELSE  331B ---------   >>
3307 0111 LIT  0003      )
330B 142C OVER           )
330D 17BE =              )------   <<
330F 07FD $IF  3319      )
3313 325D THIRD
3315 0810 $ELSE  331B ---------   >>
3319 326F WHO-KNOWS?
331B 13BB DROP ----------------   ENDCASE
331D 01C8 $;
OK
```

FIGURE 4

```
( FORTH CONTROL STRUCTURES ) BASE @ HEX
: !CADR WPARAM - , ;
: NOT
   IF 0 ELSE 1 THEN ;
: WHILE
   HERE ;  IMP WHILE
: PERFORM
   ' DUP !CADR
   ' <R !CADR ' $IF !CADR
   HERE 0 , ;     IMP PERFORM
: ENDWHILE
   HERE SWAP ! ' R> !CADR
   ' NOT !CADR ' $IF !CADR , ;
IMP ENDWHILE
BASE !   :S

( FORTH CONTROL STRUCTURES ) BASE @ HEX
: UNTIL ;    IMP UNTIL
: CASE 0 0 ;   IMP CASE
: SEL 0 -1 ;    IMP SEL
: COND 0 -2 ;    IMP COND  ( DO CONDITIONAL BRANCH )
: >>
   ' $ELSE !CADR 0 , HERE
   SWAP ! HERE 2 - SWAP ;   IMP >>
: ENDSEL DROP ( CASE#/FLAG )
   HERE
   WHILE OVER PERFORM
       DUP ROT ! ENDWHILE
   2DROP ' DROP !CADR ;
: ENDCASE ENDSEL ;              : ENDCOND SEL :
IMP ENDSEL   IMP ENDCASE   IMP ENDCOND
BASE !   :S

( FORTH CONTROL STRUCTURES ) BASE @ HEX
: WHEN
   DUP -2 =
   IF ' OVER !CADR
      ' = !CADR
   THEN
   ' $IF !CADR
   HERE 0 , ;
: << DUP 0< IF
   DUP -2 = IF ' DUP !CADR THEN ( COND )
   ELSE ' LIT !CADR 1+ DUP , WHEN THEN ;
IMP <<   IMP WHEN
: OTHERWISE ;   IMP OTHERWISE
BASE !   ;S
```

FIGURE 5

MEETINGS

NORTHERN CALIFORNIA

9/27/80

Dave Lion announced availablility of his 6800 assembler in FORTH occupying 1.5 Kbytes of 4 screens.

Tom Zimmer annonced 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

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
4th Saturday — FIG Meeting, 11:00 a.m. Allstate Savings, 8800 So. Sepulveda, L.A. Call Phillip Wass, (213) 649-1428.

FIGGRAPH
11/15/80
12/13/80 — FORTH for computer graphics. 2:00 p.m. at Stanford Medical School, #M-112 at Palo Alto, CA.

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.