SYMBOL TABLE ALGORITHMS

W. M. McKEEMAN

and

N. McDANIEL

CEP REPORT, VOLUME 2, NO. 1

APRIL 1970

The Computer Evolution Project
Applied Sciences
The University of California at
Santa Cruz, California 95060

ABSTRACT

Implementation of four methods for accessing symbol tables is presented. Particular attention is given to the problem of local variables resulting from scopes of declarations in ALGOL and PL like languages. Evaluations of the methods are made considering critical resources. Some useful variations and hints toward selecting proper algorithms are discussed.

INTRODUCTION

During the translation of many programming languages it is necessary to connect each occurrence of an identifier with its declaration. This is accomplished by means of a symbol table which holds relevant information about all active identifiers encountered in the source text. Information required for translation, and held in the symbol table, may include the name, type, location, diagnostic information, scope nesting, etc. An entry is made into the symbol table when a new identifier is declared. When an identifier is used the symbol table is interrogated for the information on which to base translation decisions. If the same identifier has been declared in nested scopes, the declaration in the innermost scope controls its use. As a result the symbol table search mechanism must insure that the first occurrence found will be the innermost. All entries local to a scope (e.g., local variables to a procedure) become irrelevant upon exit from that scope and are removed from the symbol table. This serves the dual function of freeing space in the symbol table and "uncovering" any previous use of those symbols in outer scopes.

Symbol table access consumes a major portion of the processor time during translation. For example, a study of an efficient translator for the PL-like language XPL revealed that one-fourth of its translation time was spent

interrogating the symbol table when a linear search method was employed. Changing to a hashed table lookup algorithm, a faster access method for this application, saved nearly all of that time.

There are four methods for symbol table access presented here for evaluation and comparison (linear, hash, sorted, and tree). None are new and all have their merits depending on the application for which they are being used. Implementation and testing of the four methods has been made as similar as possible to aid the reader in comparison.

IDENTIFIER DECLARATION AND SYMBOL TABLE MECHANISM

Certain programming languages, such as ALGOL-60 and PL/1, have nested scopes of application for their identifiers. A scope is delimited by matching bracketing symbols (such as begin-end). The appearance of a declaration for an identifier within the brackets makes the identifier local to that scope (i.e., not available outside the brackets). When a single identifier is declared in more than one level of nested scopes, the innermost declaration takes precedence (see Figure 1).

```
 1    /* A PL/1 PROGRAM FRAGMENT */
 2    BEGIN;
 3       DECLARE (A, B) FIXED;
 4       A = 1;
 5       BEGIN;
 6          DECLARE (C, A) FIXED;
 7          A, B, C = 1;
 8          BEGIN;
 9             A = 1;
10          END;
11          BEGIN;
12             DECLARE A FIXED;
13             A = 1;
14          END;
15          A = 1;
16       END;
17       A = 1;
18    END;
```

Figure 1.

The symbol table for a nested language holds a record of identifiers and the information associated with them. Upon encountering the declaration of an identifier (explicit or implicit), a translator must first check that there has been no previous declaration of it in the present scope and then enter it into the table. Upon encountering the use of an identifier, the translator must find the symbol table entry for the corresponding declaration to make available the associated information. The scope bracketing symbols must also cause the translator to react appropriately.

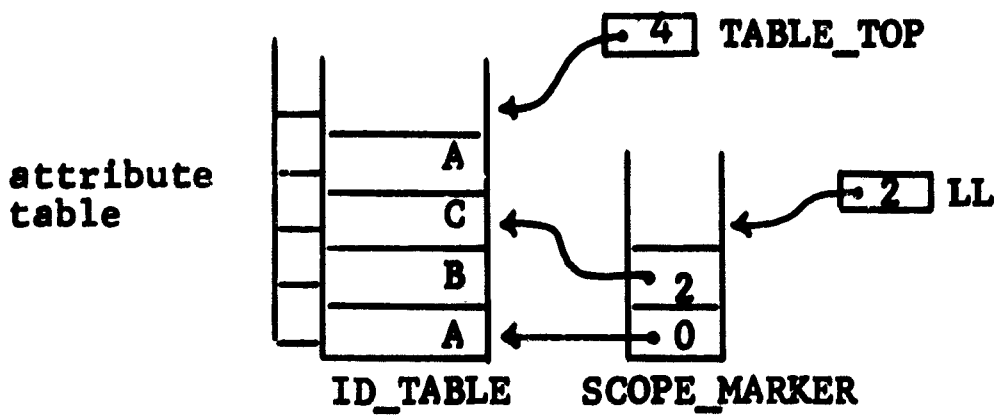
The simplest table organization for symbols in a nested language is a stack. Upon scope entry the stack must be marked to delimit the new scope; upon encountering a declaration the new identifier is stacked; upon encountering the use of an identifier the stack is searched from newest entry to oldest to find the most recently declared occurrence of that name; upon scope exit the identifiers local to the scope must be discarded.

The speed with which the above operations can be accomplished is often a critical design criterion for the symbol table mechanism. There are several access methods that can be superimposed on the stack organized table to improve performance. The choice between them is based on various considerations of table size and frequency of access; several choices are given below. In each case there are four procedures corresponding to the four access actions as noted in Table 1.

procedure name	action accomplished
new_id	An identifier has just been declared. Check that no previous declaration has been given and then enter the symbol into the table.
old_id	An identifier has just been used. Find the table location corresponding to its most recent declaration.
scope_entry	A beginning bracket for a scope has been encountered. Prepare a new local naming scope.
scope_exit	An ending bracket for a scope has been encountered. Discard identifiers no longer accessible and reestablish the next outer scope.

TABLE I

LINEAR SYMBOL TABLE ACCESS



LINEAR ACCESS METHOD
SYMBOL TABLE CONFIGURATION

Figure 2

The diagram in Figure 2 depicts the linear search method symbol table as it would appear after a translator has processed lines 1-6 of the program fragment in Figure 1. ID_TABLE holds the identifiers A, B, C, and A while SCOPE_MARKER signifies that the first two are in the outer scope and the second two are local to the inner scope. The variables LL and TABLE_TOP point to the next available cell in their respective tables. The attributes of the identifiers are held in another table which is accessed by the same pointer as ID_TABLE allowing the translator to first locate an identifier then use its location to sample or add to the associated attributes.

The linear access method is expressed and tested in the following XPL program. (The language XPL is close enough to PL/1 so that the reader should easily understand the meaning of its constructs.) The output of the program consists of a series of snapshots of the table contents and pointers after each action triggered by the program in Figure 1. INDEX records the location found by the procedure invoked. The signals IN and OUT correspond to scope entry and exit caused by BEGIN and END; NEW A signifies the declaration of the identifier A and OLD A the use of A. The table configuration in Figure 2 is reached after the seventh event in the test run.

```
1  |            /*   S Y M B O L   T A B L E   A L G O R I T H M S   */
2  |            /*   L I N E A R   S E A R C H   M E T H O D       */
3  |
4  | /*  DATA STRUCTURE DEFINITIONS:
5  |
6  |    ID_TABLE() HOLDS THE IDENTIFIERS,
7  |    TABLE_TOP POINTS TO THE NEXT AVAILABLE CELL IN ID_TABLE(),
8  |    TABLE_LIMIT IS THE BOUND ON TABLE_TOP,
9  |
10 |    SCOPE_MARKER POINTS TO THE FIRST ENTRY IN EACH SCOPE,
11 |    LL IS THE PRESENT LEVEL OF PROGRAM NESTING,
12 |    LL_LIMIT IS THE BOUND ON LL,
13 |
14 |    INDEX IS THE SYMBOL TABLE LOCATION FOUND BY THE ACCESS PROCEDURES.
15 | */
16 |
17 |DECLARE TABLE_LIMIT LITERALLY '100', TABLE_TOP FIXED,
18 |    ID_TABLE(TABLE_LIMIT) CHARACTER;
19 |DECLARE LL_LIMIT LITERALLY '10', LL FIXED, SCOPE_MARKER(LL_LIMIT) FIXED;
20 |DECLARE INDEX FIXED;
21 |
22 |ERROR: PROCEDURE; OUTPUT,OUTPUT = 'ERROR'; CALL EXIT; END ERROR;
23 |
24 |NEW_ID:
25 |    PROCEDURE(IDENTIFIER);
26 |       DECLARE IDENTIFIER CHARACTER;
27 |       DECLARE SB FIXED;
28 |
29 |       /* SEARCH FOR DUPLICATE DECLARATION */
30 |       SB = SCOPE_MARKER(LL-1);
31 |       INDEX = TABLE_TOP;
32 |       DO WHILE INDEX > SB;
33 |          INDEX = INDEX - 1;
34 |          IF IDENTIFIER = ID_TABLE(INDEX) THEN CALL ERROR;
35 |       END;
36 |
37 |       /* CHECK FOR ID_TABLE OVERFLOW */
38 |       IF TABLE_TOP = TABLE_LIMIT THEN CALL ERROR;
39 |
40 |       /* ENTER NEW IDENTIFIER */
41 |       INDEX = TABLE_TOP;   TABLE_TOP = TABLE_TOP + 1;
42 |       ID_TABLE(INDEX) = IDENTIFIER;
43 |    END NEW_ID;
44 |
45 |OLD_ID:
46 |    PROCEDURE(IDENTIFIER);
47 |       DECLARE IDENTIFIER CHARACTER;
48 |
49 |       /* SEARCH  ID_TABLE FOR THE IDENTIFIER */
50 |       INDEX = TABLE_TOP;
51 |       DO WHILE INDEX > 0;
52 |          INDEX = INDEX - 1;
53 |          IF IDENTIFIER = ID_TABLE(INDEX) THEN RETURN;
54 |       END;
```

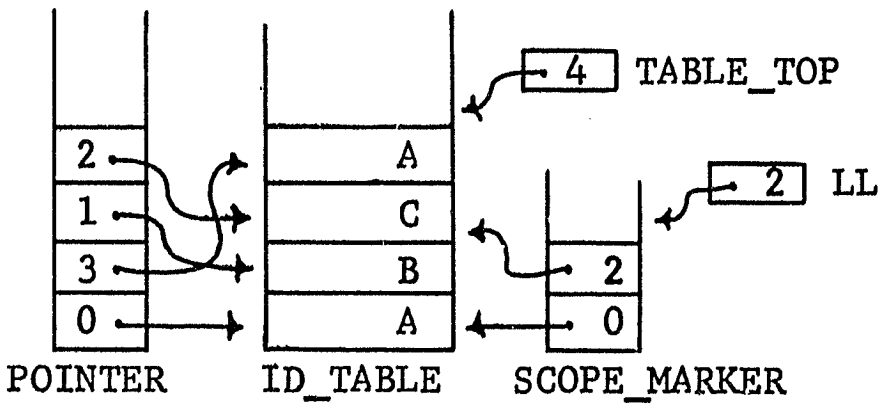
```
55 |
56 |       /* RECORD FAILURE TO FIND THE IDENTIFIER */
57 |       CALL ERROR;
58 |    END OLD_ID;
59 |
60 |SCOPE_ENTRY:
61 |    PROCEDURE;
62 |       /* MAKE SURE PROGRAM TEXT IS NOT TOO DEEPLY NESTED */
63 |       IF LL = LL_LIMIT THEN CALL ERROR;
64 |       SCOPE_MARKER(LL) = TABLE_TOP;      /* POINT TO FIRST LOCAL */
65 |       LL = LL + 1;  /* INCREASE LEXIC LEVEL */
66 |    END SCOPE_ENTRY;
67 |
68 |SCOPE_EXIT:
69 |    PROCEDURE;
70 |       LL = LL - 1;
71 |       TABLE_TOP = SCOPE_MARKER(LL);
72 |    END SCOPE_EXIT;
73 |
74 | /* TEST PROGRAM FOR SYMBOL TABLE ALGORITHMS */
75 |DECLARE (CARD,LINE) CHARACTER;
76 |DECLARE I FIXED;
77 |OUTPUT = '      SIMULATION OF EVENTS DURING TRANSLATION ';
78 |OUTPUT = '';
79 |OUTPUT = 'EVENT:     TABLE STATUS:';
80 |OUTPUT = '';
81 |LL = 0;  TABLE_TOP = 0;
82 |DO WHILE LL >= 0;
83 |
84 |    /* PRINT STATUS OF TABLES AND POINTERS */
85 |    OUTPUT = '           TABLE_TOP='||TABLE_TOP||'  LL='||LL||'  INDEX='||INDEX;
86 |    LINE = '          ID_TABLE()=    ';
87 |    DO I = 0 TO TABLE_TOP-1;
88 |       LINE = LINE || ID_TABLE(I) || ' ';
89 |    END;
90 |    OUTPUT = LINE;
91 |    LINE = '          SCOPE_MARKER()=';
92 |    DO I = 0 TO LL-1;
93 |       LINE = LINE || SCOPE_MARKER(I) || ' ';
94 |    END;
95 |    OUTPUT = LINE;
96 |
97 |    /* SIMULATE ACTIONS OF A TRANSLATOR */
98 |    CARD, OUTPUT = INPUT;
99 |    IF SUBSTR(CARD,0,2) = 'IN' THEN CALL SCOPE_ENTRY;
100|    ELSE IF SUBSTR(CARD,0,3) = 'OUT' THEN CALL SCOPE_EXIT;
101|    ELSE IF SUBSTR(CARD,0,3) = 'NEW' THEN CALL NEW_ID(SUBSTR(CARD,5,1));
102|    ELSE IF SUBSTR(CARD,0,3) = 'OLD' THEN CALL OLD_ID(SUBSTR(CARD,5,1));
103|END;
104|EOF EOF
```

```
                TABLE_TOP=0   LL=0   INDEX=0
                ID_TABLE()=
                SCOPE_MARKER()=
IN
                TABLE_TOP=0   LL=1   INDEX=0
                ID_TABLE()=
                SCOPE_MARKER()=0
NEW   A
                TABLE_TOP=1   LL=1   INDEX=0
                ID_TABLE()=     A
                SCOPE_MARKER()=0
NEW   B
                TABLE_TOP=2   LL=1   INDEX=1
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
OLD   A
                TABLE_TOP=2   LL=1   INDEX=0
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
IN
                TABLE_TOP=2   LL=2   INDEX=0
                ID_TABLE()=     A B
                SCOPE_MARKER()=0 2
NEW   C
                TABLE_TOP=3   LL=2   INDEX=2
                ID_TABLE()=     A B C
                SCOPE_MARKER()=0 2
NEW   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   B
                TABLE_TOP=4   LL=2   INDEX=1
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   C
                TABLE_TOP=4   LL=2   INDEX=2
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
IN
                TABLE_TOP=4   LL=3   INDEX=2
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2 4
OLD   A
                TABLE_TOP=4   LL=3   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2 4
OUT
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
IN
```

```
                TABLE_TOP=0   LL=0   INDEX=0
                ID_TABLE()=
                SCOPE_MARKER()=
IN
                TABLE_TOP=0   LL=1   INDEX=0
                ID_TABLE()=
                SCOPE_MARKER()=0
NEW   A
                TABLE_TOP=1   LL=1   INDEX=0
                ID_TABLE()=     A
                SCOPE_MARKER()=0
NEW   B
                TABLE_TOP=2   LL=1   INDEX=1
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
OLD   A
                TABLE_TOP=2   LL=1   INDEX=0
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
IN
                TABLE_TOP=2   LL=2   INDEX=0
                ID_TABLE()=     A B
                SCOPE_MARKER()=0 2
NEW   C
                TABLE_TOP=3   LL=2   INDEX=2
                ID_TABLE()=     A B C
                SCOPE_MARKER()=0 2
NEW   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   B
                TABLE_TOP=4   LL=2   INDEX=1
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   C
                TABLE_TOP=4   LL=2   INDEX=2
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
IN
                TABLE_TOP=4   LL=3   INDEX=2
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2 4
OLD   A
                TABLE_TOP=4   LL=3   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2 4
OUT
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
IN
```

```
                TABLE_TOP=4   LL=3   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2 4
NEW   A
                TABLE_TOP=5   LL=3   INDEX=4
                ID_TABLE()=     A B C A A
                SCOPE_MARKER()=0 2 4
OLD   A
                TABLE_TOP=5   LL=3   INDEX=4
                ID_TABLE()=     A B C A A
                SCOPE_MARKER()=0 2 4
OUT
                TABLE_TOP=4   LL=2   INDEX=4
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OLD   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                SCOPE_MARKER()=0 2
OUT
                TABLE_TOP=2   LL=1   INDEX=3
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
OLD   A
                TABLE_TOP=2   LL=1   INDEX=0
                ID_TABLE()=     A B
                SCOPE_MARKER()=0
OUT
                TABLE_TOP=0   LL=0   INDEX=0
                ID_TABLE()=
                SCOPE_MARKER()=
OUT
```

SORTED SYMBOL TABLE ACCESS



SORTED TABLE ACCESS METHOD
SYMBOL TABLE CONFIGURATION

Figure 3.

If a table is ordered and we can locate the middle item, half of the table can be discarded with a single comparison. The remaining half can be treated similarly, etc., until only one item remains. Since the requirement of nested language scope predetermines the table order, a second ordering is imposed via a table of pointers.

Figure 3 depicts the sorted table method after processing lines 1-6 of Figure 1. The order:

```
ID_TABLE(POINTER(1)) = A
ID_TABLE(POINTER(2)) = A
ID_TABLE(POINTER(3)) = B
ID_TABLE(POINTER(4)) = C
```

is in increasing collating sequence allowing the subdivision to be accomplished. Multiple entries are handled by always taking the higher one (e.g., ID_TABLE(POINTER(2)) for A in this instance).

Exercise Verify that the simulation output corresponds to the correct symbol table actions.

Exercise Hand simulate the look-up implied by action 8 (OLD A) of the simulation.

```
1  |            /*   S Y M B O L   T A B L E   A L G O R I T H M S          */
2  |            /*   S O R T E D   T A B L E   A C C E S S   M E T H O D    */
3  |
4  | /*  DATA STRUCTURE DEFINITIONS:
5  |
6  |    ID_TABLE() HOLDS THE IDENTIFIERS,
7  |    TABLE_TOP POINTS TO THE NEXT AVAILABLE CELL IN ID_TABLE(),
8  |    TABLE_LIMIT IS THE BOUND ON TABLE_TOP,
9  |
10 |    SCOPE_MARKER POINTS TO THE FIRST ENTRY IN EACH SCOPE,
11 |    LL IS THE PRESENT LEVEL OF PROGRAM NESTING,
12 |    LL_LIMIT IS THE BOUND ON LL,
13 |
14 |    INDEX IS THE SYMBOL TABLE LOCATION FOUND BY THE ACCESS PROCEDURES.
15 | */
16 |
17 |DECLARE TABLE_LIMIT LITERALLY '100', TABLE_TOP FIXED,
18 |    ID_TABLE(TABLE_LIMIT) CHARACTER;
19 |DECLARE LL_LIMIT LITERALLY '10', LL FIXED, SCOPE_MARKER(LL_LIMIT) FIXED;
20 |DECLARE INDEX FIXED;
21 |
22 | /*  POINTERS FOR INDIRECT SORT */
23 |DECLARE POINTER(TABLE_LIMIT) FIXED;
24 |
25 |ERROR: PROCEDURE; OUTPUT,OUTPUT = 'ERROR'; CALL EXIT; END ERROR;
26 |
27 |NEW_ID:
28 |    PROCEDURE(IDENTIFIER);
29 |       DECLARE IDENTIFIER CHARACTER;
30 |       DECLARE (B, M, T) FIXED;
31 |
32 |       /* SEARCH FOR DUPLICATE DECLARATION */
33 |       B = -1;  M, T = TABLE_TOP;
34 |       DO WHILE B+1 < T;
35 |          M = SHR(B+T, 1);
36 |          IF IDENTIFIER < ID_TABLE(POINTER(M)) THEN T = M;
37 |          ELSE B = M;
38 |       END;
39 |       IF B = M THEN
40 |          IF IDENTIFIER = ID_TABLE(POINTER(M)) THEN
41 |             IF POINTER(M) >= SCOPE_MARKER(LL-1) THEN CALL ERROR;
42 |
43 |       /* CHECK FOR ID_TABLE OVERFLOW */
44 |       IF TABLE_TOP = TABLE_LIMIT THEN CALL ERROR;
45 |
46 |       /* ENTER NEW IDENTIFIER */
47 |       INDEX = TABLE_TOP;   TABLE_TOP = TABLE_TOP + 1;
48 |       ID_TABLE(INDEX) = IDENTIFIER;
49 |
50 |       /* KEEP THE POINTER TABLE IN ORDER */
51 |       T = INDEX;
52 |       DO WHILE B+1 < T;
53 |          POINTER(T) = POINTER(T-1);
54 |          T = T - 1;
```

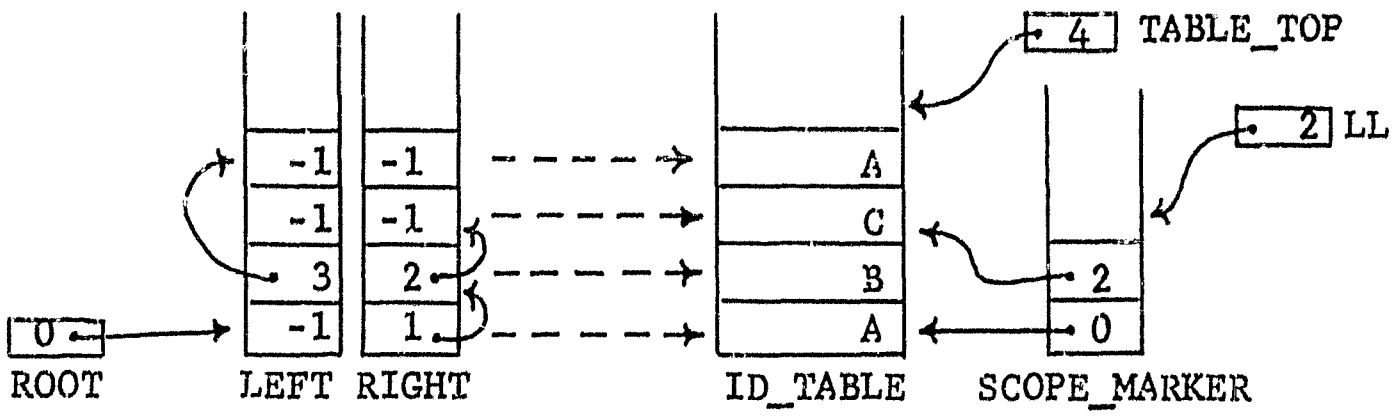
```
55 |       END;
56 |       POINTER(T) = INDEX;
57 |    END NEW_ID;
58 |
59 |OLD_ID:
60 |    PROCEDURE(IDENTIFIER);
61 |       DECLARE IDENTIFIER CHARACTER;
62 |       DECLARE (B, M, T) FIXED;
63 |
64 |       /* SEARCH  ID_TABLE FOR THE IDENTIFIER */
65 |       IF TABLE_TOP = 0 THEN CALL ERROR;
66 |       B = -1;  M, T = TABLE_TOP;
67 |       DO WHILE B+1 < T;
68 |          M = SHR(B+T, 1);
69 |          IF IDENTIFIER < ID_TABLE(POINTER(M)) THEN T = M;
70 |          ELSE B = M;
71 |       END;
72 |
73 |       /* RECORD FAILURE TO FIND THE IDENTIFIER */
74 |       IF B < 0 THEN CALL ERROR;
75 |       INDEX = POINTER(B);
76 |       IF IDENTIFIER ¬= ID_TABLE(INDEX) THEN CALL ERROR;
77 |    END OLD_ID;
78 |
79 |SCOPE_ENTRY:
80 |    PROCEDURE;
81 |       /* MAKE SURE PROGRAM TEXT IS NOT TOO DEEPLY NESTED */
82 |       IF LL = LL_LIMIT THEN CALL ERROR;
83 |       SCOPE_MARKER(LL) = TABLE_TOP;      /* POINT TO FIRST LOCAL */
84 |       LL = LL + 1;  /* INCREASE LEXIC LEVEL */
85 |    END SCOPE_ENTRY;
86 |
87 |SCOPE_EXIT:
88 |    PROCEDURE;
89 |       DECLARE (SB, B, T) FIXED;
90 |
91 |       LL = LL - 1;
92 |       /* DISCARD POINTERS INTO LIMBO */
93 |       T, B = 0;  SB = SCOPE_MARKER(LL);
94 |       DO WHILE T < TABLE_TOP;
95 |          IF POINTER(T) < SB THEN
96 |             DO;  /* SAVE GOOD ONES */
97 |                POINTER(B) = POINTER(T);
98 |                B = B + 1;
99 |             END;
100|          T = T + 1;
101|       END;
102|       TABLE_TOP = SB;
103|    END SCOPE_EXIT;
104|
105| /* TEST PROGRAM FOR SYMBOL TABLE ALGORITHMS */
106|DECLARE (CARD,LINE,LINE1) CHARACTER;
107|DECLARE I FIXED;
108|OUTPUT = '     SIMULATION OF EVENTS DURING TRANSLATION ';
109|OUTPUT = '';
110|OUTPUT = 'EVENT:     TABLE STATUS:';
111|OUTPUT = '';
112|LL = 0;  TABLE_TOP = 0;
113|DO WHILE LL >= 0;
114|
115|    /* PRINT STATUS OF TABLES AND POINTERS */
```

```
116 |    OUTPUT = '           TABLE_TOP='||TABLE_TOP||'  LL='||LL||'  INDEX='||INDEX;
117 |    LINE = '          ID_TABLE()=    ';
118 |    LINE1 = '          POINTER()=     ';
119 |    DO I = 0 TO TABLE_TOP-1;
120 |       LINE = LINE || ID_TABLE(I) || ' ';
121 |       LINE1 = LINE1 || POINTER(I) || ' ';
122 |    END;
123 |    OUTPUT = LINE;
124 |    OUTPUT = LINE1;
125 |    LINE = '          SCOPE_MARKER()=';
126 |    DO I = 0 TO LL-1;
127 |       LINE = LINE || SCOPE_MARKER(I) || ' ';
128 |    END;
129 |    OUTPUT = LINE;
130 |
131 |    /* SIMULATE ACTIONS OF A TRANSLATOR */
132 |    CARD, OUTPUT = INPUT;
133 |    IF SUBSTR(CARD,0,2) = 'IN' THEN CALL SCOPE_ENTRY;
134 |    ELSE IF SUBSTR(CARD,0,3) = 'OUT' THEN CALL SCOPE_EXIT;
135 |    ELSE IF SUBSTR(CARD,0,3) = 'NEW' THEN CALL NEW_ID(SUBSTR(CARD,5,1));
136 |    ELSE IF SUBSTR(CARD,0,3) = 'OLD' THEN CALL OLD_ID(SUBSTR(CARD,5,1));
137 |END;
138 |EOF EOF
```

```
                TABLE_TOP=0   LL=0   INDEX=0
                ID_TABLE()=
                POINTER()=
                SCOPE_MARKER()=
IN
                TABLE_TOP=0   LL=1   INDEX=0
                ID_TABLE()=
                POINTER()=
                SCOPE_MARKER()=0
NEW   A
                TABLE_TOP=1   LL=1   INDEX=0
                ID_TABLE()=     A
                POINTER()=      0
                SCOPE_MARKER()=0
NEW   B
                TABLE_TOP=2   LL=1   INDEX=1
                ID_TABLE()=     A B
                POINTER()=      0 1
                SCOPE_MARKER()=0
OLD   A
                TABLE_TOP=2   LL=1   INDEX=0
                ID_TABLE()=     A B
                POINTER()=      0 1
                SCOPE_MARKER()=0
IN
                TABLE_TOP=2   LL=2   INDEX=0
                ID_TABLE()=     A B
                POINTER()=      0 1
                SCOPE_MARKER()=0 2
NEW   C
                TABLE_TOP=3   LL=2   INDEX=2
                ID_TABLE()=     A B C
                POINTER()=      0 1 2
                SCOPE_MARKER()=0 2
NEW   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                POINTER()=      0 3 1 2
                SCOPE_MARKER()=0 2
OLD   A
                TABLE_TOP=4   LL=2   INDEX=3
                ID_TABLE()=     A B C A
                POINTER()=      0 3 1 2
                SCOPE_MARKER()=0 2
OLD   B
                TABLE_TOP=4   LL=2   INDEX=1
                ID_TABLE()=     A B C A
                POINTER()=      0 3 1 2
                SCOPE_MARKER()=0 2
OLD   C
                TABLE_TOP=4   LL=2   INDEX=2
                ID_TABLE()=     A B C A
                POINTER()=      0 3 1 2
                SCOPE_MARKER()=0 2
IN
                TABLE_TOP=4   LL=3   INDEX=2
```

```
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2 4
OLD   A
          TABLE_TOP=4   LL=3   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2 4
OUT
          TABLE_TOP=4   LL=2   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2
IN
          TABLE_TOP=4   LL=3   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2 4
NEW   A
          TABLE_TOP=5   LL=3   INDEX=4
          ID_TABLE()=     A B C A A
          POINTER()=      0 3 4 1 2
          SCOPE_MARKER()=0 2 4
OLD   A
          TABLE_TOP=5   LL=3   INDEX=4
          ID_TABLE()=     A B C A A
          POINTER()=      0 3 4 1 2
          SCOPE_MARKER()=0 2 4
OUT
          TABLE_TOP=4   LL=2   INDEX=4
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2
OLD   A
          TABLE_TOP=4   LL=2   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      0 3 1 2
          SCOPE_MARKER()=0 2
OUT
          TABLE_TOP=2   LL=1   INDEX=3
          ID_TABLE()=     A B
          POINTER()=      0 1
          SCOPE_MARKER()=0
OLD   A
          TABLE_TOP=2   LL=1   INDEX=0
          ID_TABLE()=     A B
          POINTER()=      0 1
          SCOPE_MARKER()=0
OUT
          TABLE_TOP=0   LL=0   INDEX=0
          ID_TABLE()=
          POINTER()=
          SCOPE_MARKER()=
OUT
```

TREE SYMBOL TABLE ACCESS



BINARY TREE ACCESS METHOD
SYMBOL TABLE CONFIGURATION
Figure 4

An access tree is a structure that has nodes corresponding to each identifier in the table. Starting from some root position, an identifier is compared with the identifier at the root and either the left or right branch taken (unless marked with -1, signifying the node is a leaf of the tree). The tree corresponding to the table configuration above (which itself corresponds to the processing of lines 1-6 of Figure 1 as usual) is more readily understood from the diagram in Figure 5.

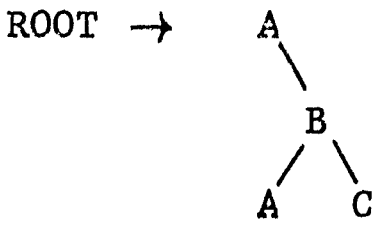


Figure 5

A is at the root. Since nothing will sort "<" than A, everything else hangs off the right branch of the tree. The next node, B, allows the second A to sort left and the C to sort right where the tree terminates in leaves. Because the tree is grown from oldest entry out to recent entries at the leaves, the last entry

found is the correct one.

Exercise Hand simulate the access algorithm for action 8 (OLD A) of the simulation output.

```
1  |            /*   S Y M B O L   T A B L E   A L G O R I T H M S       */
2  |            /*   B I N A R Y   T R E E   A C C E S S   M E T H O D   */
3  |
4  | /*  DATA STRUCTURE DEFINITIONS:
5  |
6  |   ID_TABLE() HOLDS THE IDENTIFIERS,
7  |   TABLE_TOP POINTS TO THE NEXT AVAILABLE CELL IN ID_TABLE(),
8  |   TABLE_LIMIT IS THE BOUND ON TABLE_TOP,
9  |
10 |   SCOPE_MARKER POINTS TO THE FIRST ENTRY IN EACH SCOPE,
11 |   LL IS THE PRESENT LEVEL OF PROGRAM NESTING,
12 |   LL_LIMIT IS THE BOUND ON LL,
13 |
14 |   INDEX IS THE SYMBOL TABLE LOCATION FOUND BY THE ACCESS PROCEDURES.
15 |  */
16 |
17 |DECLARE TABLE_LIMIT LITERALLY '100', TABLE_TOP FIXED,
18 |   ID_TABLE(TABLE_LIMIT) CHARACTER;
19 |DECLARE LL_LIMIT LITERALLY '10', LL FIXED, SCOPE_MARKER(LL_LIMIT) FIXED;
20 |DECLARE INDEX FIXED;
21 |
22 | /* DATA STRUCTURES FOR TREE ACCESS */
23 |DECLARE (LEFT, RIGHT)(TABLE_LIMIT) FIXED, ROOT FIXED;
24 |
25 |ERROR: PROCEDURE; OUTPUT,OUTPUT = 'ERROR'; CALL EXIT; END ERROR;
26 |
27 |NEW_ID:
28 |   PROCEDURE(IDENTIFIER);
29 |      DECLARE IDENTIFIER CHARACTER;
30 |      DECLARE (I,K) FIXED;
31 |
32 |      /* SEARCH FOR DUPLICATE DECLARATION */
33 |      K = ROOT;  INDEX = -1;
34 |      DO WHILE K ¬= -1;
35 |         IF IDENTIFIER = ID_TABLE(K) THEN INDEX = K;
36 |         I = K;
37 |         IF IDENTIFIER < ID_TABLE(K) THEN K = LEFT(K); ELSE K = RIGHT(K);
38 |      END;
39 |      IF INDEX >= SCOPE_MARKER(LL-1) THEN CALL ERROR;
40 |
41 |      /* CHECK FOR ID_TABLE OVERFLOW */
42 |      IF TABLE_TOP = TABLE_LIMIT THEN CALL ERROR;
43 |
44 |      /* ENTER NEW IDENTIFIER */
45 |      INDEX = TABLE_TOP;   TABLE_TOP = TABLE_TOP + 1;
46 |      ID_TABLE(INDEX) = IDENTIFIER;
47 |      IF ROOT = -1 THEN ROOT = INDEX;
48 |      ELSE IF IDENTIFIER < ID_TABLE(I) THEN LEFT(I) = INDEX;
49 |      ELSE RIGHT(I) = INDEX;
50 |      LEFT(INDEX), RIGHT(INDEX) = -1;
51 |   END NEW_ID;
52 |
53 |OLD_ID:
54 |   PROCEDURE(IDENTIFIER);
```

```
55 |       DECLARE IDENTIFIER CHARACTER;
56 |       DECLARE K FIXED;
57 |
58 |       /* SEARCH  ID_TABLE FOR THE IDENTIFIER */
59 |       K = ROOT;  INDEX = -1;
60 |       DO WHILE K ^= -1;
61 |          IF IDENTIFIER = ID_TABLE(K) THEN INDEX = K;
62 |          IF IDENTIFIER < ID_TABLE(K) THEN K = LEFT(K); ELSE K = RIGHT(K);
63 |       END;
64 |
65 |       /* RECORD FAILURE TO FIND THE IDENTIFIER */
66 |       IF INDEX = -1 THEN CALL ERROR;
67 |    END OLD_ID;
68 |
69 |SCOPE_ENTRY:
70 |    PROCEDURE;
71 |       /* MAKE SURE PROGRAM TEXT IS NOT TOO DEEPLY NESTED */
72 |       IF LL = LL_LIMIT THEN CALL ERROR;
73 |       SCOPE_MARKER(LL) = TABLE_TOP;      /* POINT TO FIRST LOCAL */
74 |       LL = LL + 1;  /* INCREASE LEXIC LEVEL */
75 |    END SCOPE_ENTRY;
76 |
77 |SCOPE_EXIT:
78 |    PROCEDURE;
79 |    DECLARE I FIXED;
80 |       LL = LL - 1;
81 |       TABLE_TOP = SCOPE_MARKER(LL);
82 |       /* DISCARD LEAVES CORRESPONDING TO LOCAL IDENTIFIERS */
83 |       IF ROOT >= TABLE_TOP THEN ROOT = -1;
84 |       ELSE
85 |          DO I = 0 TO TABLE_TOP - 1;
86 |             IF LEFT(I) >= TABLE_TOP THEN LEFT(I) = -1;
87 |             IF RIGHT(I) >= TABLE_TOP THEN RIGHT(I) = -1;
88 |          END;
89 |    END SCOPE_EXIT;
90 |
91 | /* TEST PROGRAM FOR SYMBOL TABLE ALGORITHMS */
92 |DECLARE (CARD,LINE,LINE1,LINE2) CHARACTER;
93 |DECLARE I FIXED;
94 |OUTPUT = '      SIMULATION OF EVENTS DURING TRANSLATION ';
95 |OUTPUT = '';
96 |OUTPUT = 'EVENT:     TABLE STATUS:';
97 |OUTPUT = '';
98 |LL = 0;  TABLE_TOP = 0;
99 |ROOT = -1;
100 |DO WHILE LL >= 0;
101 |
102 |    /* PRINT STATUS OF TABLES AND POINTERS */
103 |    OUTPUT = '           TABLE_TOP='||TABLE_TOP||'  LL='||LL||'  INDEX='||INDEX;
104 |    LINE = '          ID_TABLE()=    ';
105 |    LINE1 = '           LEFT()=      ';
106 |    LINE2 = '           RIGHT()=     ';
107 |    DO I = 0 TO TABLE_TOP-1;
108 |       LINE = LINE || ID_TABLE(I) || ' ';
109 |       LINE1 = LINE1 || LEFT(I) || ' ';
110 |       LINE2 = LINE2 || RIGHT(I) || ' ';
111 |    END;
112 |    OUTPUT = LINE;
113 |    OUTPUT = '           ROOT='|| ROOT;
114 |    OUTPUT = LINE1;
115 |    OUTPUT = LINE2;
```

```
116 |    LINE = '           SCOPE_MARKER()=';
117 |    DO I = 0 TO LL-1;
118 |       LINE = LINE || SCOPE_MARKER(I) || ' ';
119 |    END;
120 |    OUTPUT = LINE;
121 |
122 |    /* SIMULATE ACTIONS OF A TRANSLATOR */
123 |    CARD, OUTPUT = INPUT;
124 |    IF SUBSTR(CARD,0,2) = 'IN' THEN CALL SCOPE_ENTRY;
125 |    ELSE IF SUBSTR(CARD,0,3) = 'OUT' THEN CALL SCOPE_EXIT;
126 |    ELSE IF SUBSTR(CARD,0,3) = 'NEW' THEN CALL NEW_ID(SUBSTR(CARD,5,1));
127 |    ELSE IF SUBSTR(CARD,0,3) = 'OLD' THEN CALL OLD_ID(SUBSTR(CARD,5,1));
128 |END;
129 |EOF EOF
```

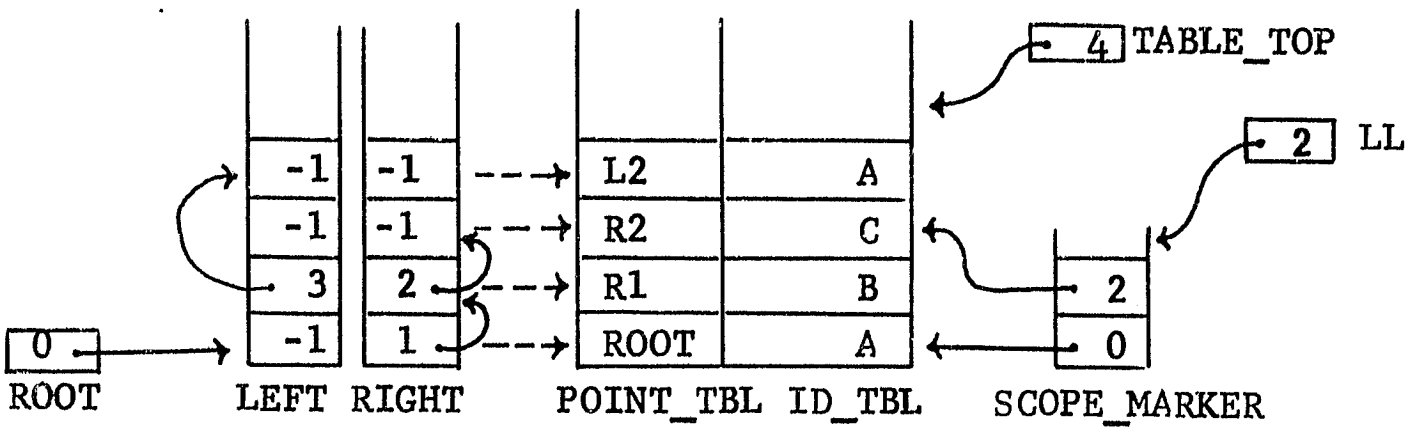
```
EVENT:    TABLE STATUS:

          TABLE_TOP=0  LL=0  INDEX=0
          ID_TABLE()=
          ROOT=-1
          LEFT()=
          RIGHT()=
          SCOPE_MARKER()=
IN
          TABLE_TOP=0  LL=1  INDEX=0
          ID_TABLE()=
          ROOT=-1
          LEFT()=
          RIGHT()=
          SCOPE_MARKER()=0
NEW  A
          TABLE_TOP=1  LL=1  INDEX=0
          ID_TABLE()=    A
          ROOT=0
          LEFT()=        -1
          RIGHT()=       -1
          SCOPE_MARKER()=0
NEW  B
          TABLE_TOP=2  LL=1  INDEX=1
          ID_TABLE()=    A B
          ROOT=0
          LEFT()=        -1 -1
          RIGHT()=       1 -1
          SCOPE_MARKER()=0
OLD  A
          TABLE_TOP=2  LL=1  INDEX=0
          ID_TABLE()=    A B
          ROOT=0
          LEFT()=        -1 -1
          RIGHT()=       1 -1
          SCOPE_MARKER()=0
IN
          TABLE_TOP=2  LL=2  INDEX=0
          ID_TABLE()=    A B
          ROOT=0
          LEFT()=        -1 -1
          RIGHT()=       1 -1
          SCOPE_MARKER   =0 2
NEW  C
          TABLE_TOP=3  LL=2  INDEX=2
          ID_TABLE()=    A B C
          ROOT=0
          LEFT()=        -1 -1 -1
          RIGHT()=       1 2 -1
          SCOPE_MARKER()=0 2
NEW  A
          TABLE_TOP=4  LL=2  INDEX=3
          ID_TABLE()=    A B C A
          ROOT=0
          LEFT()=        -1 3 -1 -1
          RIGHT()=       1 2 -1 -1
          SCOPE_MARKER()=0 2
OLD  A
```

```
            TABLE_TOP=4  LL=2  INDEX=3
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2
OLD  B
            TABLE_TOP=4  LL=2  INDEX=1
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2
OLD  C
            TABLE_TOP=4  LL=2  INDEX=2
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2
IN
            TABLE_TOP=4  LL=3  INDEX=2
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2 4
OLD  A
            TABLE_TOP=4  LL=3  INDEX=3
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2 4
OUT
            TABLE_TOP=4  LL=2  INDEX=3
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2
IN
            TABLE_TOP=4  LL=3  INDEX=3
            ID_TABLE()=    A B C A
            ROOT=0
            LEFT()=          -1 3 -1 -1
            RIGHT()=         1 2 -1 -1
            SCOPE_MARKER()=0 2 4
NEW  A
            TABLE_TOP=5  LL=3  INDEX=4
            ID_TABLE()=    A B C A A
            ROOT=0
            LEFT()=          -1 3 -1 -1 -1
            RIGHT()=         1 2 -1 4 -1
            SCOPE_MARKER()=0 2 4
OLD  A
            TABLE_TOP=5  LL=3  INDEX=4
            ID_TABLE()=    A B C A A
            ROOT=0
            LEFT()=          -1 3 -1 -1 -1
            RIGHT()=         1 2 -1 4 -1
```

```
          SCOPE_MARKER()=0 2 4
OUT
          TABLE_TOP=4   LL=2   INDEX=4
          ID_TABLE()=     A B C A
          ROOT=0
          LEFT()=         -1 3 -1 -1
          RIGHT()=        1 2 -1 -1
          SCOPE_MARKER()=0 2
OLD   A
          TABLE_TOP=4   LL=2   INDEX=3
          ID_TABLE()=     A B C A
          ROOT=0
          LEFT()=         -1 3 -1 -1
          RIGHT()=        1 2 -1 -1
          SCOPE_MARKER()=0 2
OUT
          TABLE_TOP=2   LL=1   INDEX=3
          ID_TABLE()=     A B
          ROOT=0
          LEFT()=         -1 -1
          RIGHT()=        1 -1
          SCOPE_MARKER()=0
OLD   A
          TABLE_TOP=2   LL=1   INDEX=0
          ID_TABLE()=     A B
          ROOT=0
          LEFT()=         -1 -1
          RIGHT()=        1 -1
          SCOPE_MARKER()=0
OUT
          TABLE_TOP=0   LL=0   INDEX=0
          ID_TABLE()=
          ROOT=-1
          LEFT()=
          RIGHT()=
          SCOPE_MARKER()=
OUT
```

A VARIATION ON TREE SCOPE EXIT

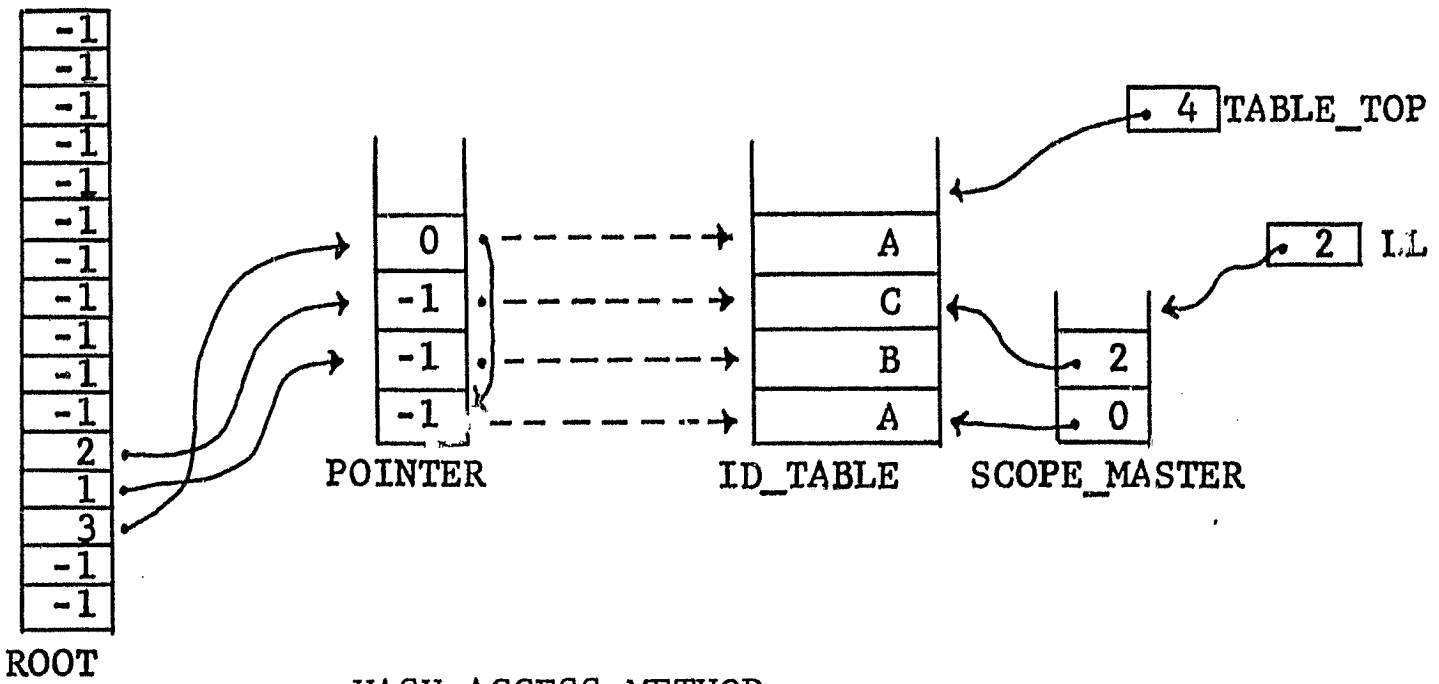
Scope exit and removal of identifiers from the tree symbol table method depicted in Figure 4 and the program requires a linear search of the left and right pointers to delete the entries greater than the SCOPE_MARKER. This is a linear process. A faster variation is shown in Figure 6. A table of pointers corresponding to each entry in the ID_TABLE is maintained. Each entry in this table points to the entry in the LEFT/RIGHT table for the corresponding identifier in the ID_TABLE. Scope exit removal of identifiers from the symbol table would be accomplished by deleting all entries in the LEFT/RIGHT table pointed to by entries in the POINTER table above the SCOPE_MARKER being returned to. This method would be an $N' \log_2 N$ process and the additional memory required for the backward pointers would be the cost.



BINARY TREE ACCESS METHOD
WITH BACKWARD POINTERS

FIGURE 6

HASH SYMBOL TABLE ACCESS



HASH ACCESS METHOD
SYMBOL TABLE CONFIGURATION

Figure 7.

A hash or scramble is an integer valued function on the identifier itself. A good hash will give an even distribution of integer values over the set of identifiers actually presented to the translator, effectively separating them into a large number of small classes. The access method only examines members of the hash class which reduces search time by roughly a factor equal to the range of scrambed values (in this case, 16).

The diagram in Figure 7 depicts the table as it would appear after the translator had processed lines 1-6 of Figure 1 (refer to the previous example).

Two arrays, ROOT and POINTER, appear in addition to the identifier table and scope marker. Each entry in ROOT points to the first identifier in its hash class (or is -1 if the class is empty). Each entry in POINTER points to the next member of the hash class (or is -1 if there are no more). To access an identifier, we scramble it, look into ROOT then follow the pointers until we find the identifier or come to the end of the chain. The pointers are arranged to point from new to old so that the first match found is correct.

Exercise Verify that the simulation output corresponds to the correct symbol table actions.

Exercise Hand simulate the algorithm for action 8 (OLD A) in the simulation.

```
 1 |            /*   S Y M B O L   T A B L E   A L G O R I T H M S   */
 2 |            /*   H A S H   S E A R C H   M E T H O D           */
 3 |
 4 | /*  DATA STRUCTURE DEFINITIONS:
 5 |
 6 |    ID_TABLE() HOLDS THE IDENTIFIERS,
 7 |    TABLE_TOP POINTS TO THE NEXT AVAILABLE CELL IN ID_TABLE(),
 8 |    TABLE_LIMIT IS THE BOUND ON TABLE_TOP,
 9 |
10 |    SCOPE_MARKER POINTS TO THE FIRST ENTRY IN EACH SCOPE,
11 |    LL IS THE PRESENT LEVEL OF PROGRAM NESTING,
12 |    LL_LIMIT IS THE BOUND ON LL,
13 |
14 |    INDEX IS THE SYMBOL TABLE LOCATION FOUND BY THE ACCESS PROCEDURES.
15 | */
16 |
17 |DECLARE TABLE_LIMIT LITERALLY '100', TABLE_TOP FIXED,
18 |    ID_TABLE(TABLE_LIMIT) CHARACTER;
19 |DECLARE LL_LIMIT LITERALLY '10', LL FIXED, SCOPE_MARKER(LL_LIMIT) FIXED;
20 |DECLARE INDEX FIXED;
21 |
22 | /*  DATA STRUCTURES REQUIRED FOR HASH ACCESS METHOD */
23 |DECLARE HASH_SIZE LITERALLY '15', ROOT(HASH_SIZE) FIXED,
24 |    POINTER(TABLE_LIMIT) FIXED;
25 |
26 |SCRAMBLE:
27 |    PROCEDURE(IDENTIFIER) FIXED;
28 |       DECLARE IDENTIFIER CHARACTER;
29 |       /* FIND A NUMBER BETWEEN 0 AND 15 */
30 |       RETURN (LENGTH(IDENTIFIER)+BYTE(IDENTIFIER))&HASH_SIZE;
31 |    END SCRAMBLE;
32 |
33 |ERROR: PROCEDURE; OUTPUT,OUTPUT = 'ERROR': CALL EXIT; END ERROR;
34 |
35 |NEW_ID:
36 |    PROCEDURE(IDENTIFIER);
37 |       DECLARE IDENTIFIER CHARACTER;
38 |       DECLARE (SB, K) FIXED;
39 |
40 |       /* SEARCH FOR DUPLICATE DECLARATION */
41 |       K = SCRAMBLE(IDENTIFIER): INDEX = ROOT(K);
42 |       SB = SCOPE_MARKER(LL-1);
43 |       DO WHILE INDEX >= SB;
44 |          IF IDENTIFIER = ID_TABLE(INDEX) THEN CALL ERROR;
45 |          INDEX = POINTER(INDEX);
46 |       END;
47 |
48 |       /* CHECK FOR ID_TABLE OVERFLOW */
49 |       IF TABLE_TOP = TABLE_LIMIT THEN CALL ERROR;
50 |
51 |       /* ENTER NEW IDENTIFIER */
52 |       INDEX = TABLE_TOP:   TABLE_TOP = TABLE_TOP + 1;
53 |       ID_TABLE(INDEX) = IDENTIFIER;
54 |       POINTER(INDEX) = ROOT(K);  ROOT(K) = INDEX;
```

```
55 |    END NEW_ID;
56 |
57 |OLD_ID:
58 |    PROCEDURE(IDENTIFIER);
59 |       DECLARE IDENTIFIER CHARACTER;
60 |
61 |       /* SEARCH  ID_TABLE FOR THE IDENTIFIER */
62 |       INDEX = ROOT(SCRAMBLE(IDENTIFIER));
63 |       DO WHILE INDEX ¬= -1;
64 |          IF IDENTIFIER = ID_TABLE(INDEX) THEN RETURN;
65 |          INDEX = POINTER(INDEX);
66 |       END;
67 |
68 |       /* RECORD FAILURE TO FIND THE IDENTIFIER */
69 |       CALL ERROR;
70 |    END OLD_ID;
71 |
72 |SCOPE_ENTRY:
73 |    PROCEDURE;
74 |       /* MAKE SURE PROGRAM TEXT IS NOT TOO DEEPLY NESTED */
75 |       IF LL = LL_LIMIT THEN CALL ERROR;
76 |       SCOPE_MARKER(LL) = TABLE_TOP;      /* POINT TO FIRST LOCAL */
77 |       LL = LL + 1;  /* INCREASE LEXIC LEVEL */
78 |    END SCOPE_ENTRY;
79 |
80 |SCOPE_EXIT:
81 |    PROCEDURE;
82 |       DECLARE K FIXED;
83 |       DECLARE K FIXED;
84 |       INDEX = TABLE_TOP;
85 |       LL = LL - 1;
86 |       TABLE_TOP = SCOPE_MARKER(LL);
87 |
88 |       /* DE-LINK IDENTIFIERS BEING DISCARDED */
89 |       DO WHILE INDEX > TABLE_TOP;
90 |          INDEX = INDEX - 1;
91 |          K = SCRAMBLE(ID_TABLE(INDEX));
92 |          ROOT(K) = POINTER(ROOT(K));
93 |       END;
94 |    END SCOPE_EXIT;
95 |
96 | /* TEST PROGRAM FOR SYMBOL TABLE ALGORITHMS */
97 |DECLARE (CARD, LINE, LINE1) CHARACTER;
98 |DECLARE I FIXED;
99 |OUTPUT = '     SIMULATION OF EVENTS DURING TRANSLATION ';
100 |OUTPUT = '';
101 |OUTPUT = 'EVENT:     TABLE STATUS:';
102 |OUTPUT = '';
103 |DO I = 0 TO HASH_SIZE;
104 |    ROOT(I) = -1;  /* MARK ALL HASH CLASSES EMPTY */
105 |END;
106 |
107 |LL = 0;  TABLE_TOP = 0;
108 |DO WHILE LL >= 0;
109 |
110 |    /* PRINT STATUS OF TABLES AND POINTERS */
111 |    OUTPUT = '          TABLE_TOP='||TABLE_TOP||'  LL='||LL||'  INDEX='||INDEX;
112 |    LINE = '          ID_TABLE()=    ';
113 |    LINE1 = '          POINTER()=     ';
114 |    DO I = 0 TO TABLE_TOP-1;
115 |       LINE = LINE || ID_TABLE(I) || ' ';
```

```
116 |       LINE1 = LINE1 || POINTER(I) || ' ';
117 |    END;
118 |    OUTPUT = LINE;
119 |    OUTPUT = LINE1;
120 |    LINE = '           SCOPE_MARKER()=';
121 |    DO I = 0 TO LL-1;
122 |       LINE = LINE || SCOPE_MARKER(I) || ' ';
123 |    END;
124 |    OUTPUT = LINE;
125 |    LINE = '           ROOT()=         ';
126 |    DO I = 0 TO HASH_SIZE;
127 |       LINE = LINE || ROOT(I) || ' ';
128 |    END;
129 |    OUTPUT = LINE;
130 |
131 |    /* SIMULATE ACTIONS OF A TRANSLATOR */
132 |    CARD, OUTPUT = INPUT;
133 |    IF SUBSTR(CARD,0,2) = 'IN' THEN CALL SCOPE_ENTRY;
134 |    ELSE IF SUBSTR(CARD,0,3) = 'OUT' THEN CALL SCOPE_EXIT;
135 |    ELSE IF SUBSTR(CARD,0,3) = 'NEW' THEN CALL NEW_ID(SUBSTR(CARD,5,1));
136 |    ELSE IF SUBSTR(CARD,0,3) = 'OLD' THEN CALL OLD_ID(SUBSTR(CARD,5,1));
137 |END;
138 |EOF EOF
```

```
EVENT:    TABLE STATUS:

          TABLE_TOP=0  LL=0  INDEX=0
          ID_TABLE()=
          POINTER()=
          SCOPE_MARKER()=
          ROOT()=          -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
IN
          TABLE_TOP=0  LL=1  INDEX=0
          ID_TABLE()=
          POINTER()=
          SCOPE_MARKER()=0
          ROOT()=          -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
NEW  A
          TABLE_TOP=1  LL=1  INDEX=0
          ID_TABLE()=      A
          POINTER()=       -1
          SCOPE_MARKER()=0
          ROOT()=          -1 -1 0 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
NEW  B
          TABLE_TOP=2  LL=1  INDEX=1
          ID_TABLE()=      A B
          POINTER()=       -1 -1
          SCOPE_MARKER()=0
          ROOT()=          -1 -1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD  A
          TABLE_TOP=2  LL=1  INDEX=0
          ID_TABLE()=      A B
          POINTER()=       -1 -1
          SCOPE_MARKER()=0
          ROOT()=          -1 -1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
IN
          TABLE_TOP=2  LL=2  INDEX=0
          ID_TABLE()=      A B
          POINTER()=       -1 -1
          SCOPE_MARKER()=0 2
          ROOT()=          -1 -1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
NEW  C
          TABLE_TOP=3  LL=2  INDEX=2
          ID_TABLE()=      A B C
          POINTER()=       -1 -1 -1
          SCOPE_MARKER()=0 2
          ROOT()=          -1 -1 0 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
NEW  A
          TABLE_TOP=4  LL=2  INDEX=3
          ID_TABLE()=      A B C A
          POINTER()=       -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=          -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD  A
          TABLE_TOP=4  LL=2  INDEX=3
          ID_TABLE()=      A B C A
          POINTER()=       -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=          -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD  B
          TABLE_TOP=4  LL=2  INDEX=1
          ID_TABLE()=      A B C A
```

```
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD   C
          TABLE_TOP=4   LL=2   INDEX=2
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
IN
          TABLE_TOP=4   LL=3   INDEX=2
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2 4
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD   A
          TABLE_TOP=4   LL=3   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2 4
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OUT
          TABLE_TOP=4   LL=2   INDEX=4
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
IN
          TABLE_TOP=4   LL=3   INDEX=4
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2 4
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
NEW   A
          TABLE_TOP=5   LL=3   INDEX=4
          ID_TABLE()=     A B C A A
          POINTER()=      -1 -1 -1 0 3
          SCOPE_MARKER()=0 2 4
          ROOT()=         -1 -1 4 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD   A
          TABLE_TOP=5   LL=3   INDEX=4
          ID_TABLE()=     A B C A A
          POINTER()=      -1 -1 -1 0 3
          SCOPE_MARKER()=0 2 4
          ROOT()=         -1 -1 4 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OUT
          TABLE_TOP=4   LL=2   INDEX=4
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD   A
          TABLE_TOP=4   LL=2   INDEX=3
          ID_TABLE()=     A B C A
          POINTER()=      -1 -1 -1 0
          SCOPE_MARKER()=0 2
          ROOT()=         -1 -1 3 1 2 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OUT
          TABLE_TOP=2   LL=1   INDEX=2
          ID_TABLE()=     A B
          POINTER()=      -1 -1
```

```
            SCOPE_MARKER()=0
            ROOT()=          -1 -1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OLD   A
            TABLE_TOP=2   LL=1   INDEX=0
            ID_TABLE()=      A B
            POINTER()=       -1 -1
            SCOPE_MARKER()=0
            ROOT()=          -1 -1 0 1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OUT
            TABLE_TOP=0   LL=0   INDEX=0
            ID_TABLE()=
            POINTER()=
            SCOPE_MARKER()=
            ROOT()=          -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1 -1
OUT
```

DEVELOPING AND SELECTING A HASH ALGORITHM

Numerous algorithms can be developed for scrambling the bits of an identifier to produce an index for entering a symbol table lookup. Some standard methods employed in these algorithms include:

1. Concatenation of N select bits of the identifier where N is the size of the required index.
2. Splitting the identifier into sections and using the sum of the sections as the index.
3. Multiplication of a portion of the identifier, usually a machine word, by a constant value and selecting the middle bits of the product as the index.

Combinations of the above methods may also be used since the object is to employ the most efficient hashing algorithm for the intended application.

Execution speed of the algorithm and the distribution of the indices across the range of the table are the considerations for selecting an algorithm for calculating the hash index. In order to choose the most efficient hashing algorithm, the trade-off between the time taken to produce the hash and flatness of the distribution of indices over the hash table must be found for each candidate algorithm and evaluated in respect to the specific application.

Concerning the speed of an algorithm, when dealing

with filing systems of large size, where the hash tables tend to be large and not always resident in primary store it may be necessary to carefully develop an efficient hash in order to minimize the need to fetch tables. In translators the complete hash and symbol table reside in primary store and the time spent in developing the hash is a factor in the overall efficiency of the hash algorithm. The accuracy of the hash produced is not critical as long as it is sufficiently distributive. In this case the faster the hash algorithm the faster will be the symbol table access. However, the speed of the hash algorithm tends to become unimportant as the average number of symbols to be interrogated during each symbol table access increases. If more than one symbol is to be looked at the time taken in development of the hash will be masked out by the time spent in the comparison of each symbol. In the examples of the hash algorithms given in the next section the average number of symbols looked at was slightly less than two and the overall access speed was only slightly effected by the faster algorithms.

Concerning the randomness of the distribution of an algorithm the ideal case is the algorithm which, given a table of length L with positions $E_1E_2....E_L$, produces hash keys to cover the entire range of L without any predictable grouping.

Consider implementation on an IBM 360 of a hash table of 256 positions which would require an index of 8 bits length. The identifier is presented as the key and the hashing algorithm consists of the least significant four bits of the first letter plus the length shifted left four bits all masked to 8 bits. The indices produced by this algorithm will cover a range of only 144 numbers out of the possible 256; 112 of the Table locations would never be used. The reason is that the least significant four bits of the hexadicinal code for letters maps into only 9 of the entire range of 16 possible numbers. The least significant four bits of the key would always be 0-9 instead of 0-F. Even if two characters were added together the range would be 18 and overflow at 16 would cause the index 1 and 2 to occur more frequently. The index could be further massaged by some constant arithmetic function to produce a covering of the entire range but a grouping would still be predictable and the increase in execution time due to the arithmetics would cause a decrease in the overall efficiency of the algorithm. Use of the length of the identifier has its drawbacks as well since there would obviously not be an even distribution of identifier lengths across the range of 16 unless the programmers style provided it. Therefore in selecting a hashing

algorithm care must be taken to insure that the data selected as the hash key is potentially distributive over the entire range of the hash Table and is sufficiently random itself.

COMPARING HASH ALGORITHMS

A test of six hash algorithms was performed on an IBM 360/40 using as input the actual identifiers taken from a student translator interpreter program. The intent was to determine the overall efficiency of each algorithm, speed, against distribution, using data representative of a normal programming problem. There were 772 identifiers used including some duplicates as a result of scopes of declarations. The beginning letter of the identifiers used tended to group around the more common letters. The lengths were well distributed with 113 single letter identifiers, an average of about 50 for each length identifier up to 12, and a sharp fall off above that number. The test program simulated the normal symbol table operations of a compiler using the same hash access method previously described. The program also simulated the exact scope entries and exits of the program the test data was lifted from. A look-up of all identifiers present in the symbol table was performed at each scope exit and a total of 18099 symbol table accesses were made for the set of identifiers.

The algorithms are shown imbedded in the actual procedures used in the test program. In each procedure the parameter ID contains the identifier to be hashed and the procedure returns an index value of the range 0 to 255.

```
 1 |
 2 |
 3 |ALGORITHM_1:
 4 |    PROCEDURE (ID) FIXED;
 5 |       DECLARE ID CHARACTER;
 6 |       IF LENGTH(ID) = 1 THEN
 7 |          ID = ID || ' ';
 8 |       RETURN ((BYTE(ID)&"0F")+ (BYTE(ID,1) & "0F")+SHL(LENGTH(ID),4))&"FF";
 9 |    END ALGORITHM_1 ;
10 |
11 | /*  ALGORITHM_1   PRODUCES AN INDEX FROM THE SUM OF THE LOW ORDER
12 |     FOUR BITS OF THE FIRST TWO CHARACTERS CONCATENATED WITH THE LENGTH
13 |     OF THE IDENTIFIER AS THE HIGH ORDER FOUR BITS OF THE INDEX.
14 | */
15 |
16 |
17 |
18 |ALGORITHM_2:
19 |    PROCEDURE (ID) FIXED;
20 |       DECLARE ID CHARACTER, L FIXED;
21 |       L = LENGTH(ID);
22 |       RETURN((BYTE(ID) & "3F") + (BYTE(ID,L-1)&"3F") + SHL(L,4)) & "FF";
23 |    END ALGORITHM_2 ;
24 |
25 | /*  ALGORITHM_2    PRODUCES AN INDEX FROM THE SUM OF THE LOW ORDER
26 |     SIX BITS OF THE FIRST AND LAST CHARACTERS AND THE LENGTH OF THE
27 |     IDENTIFIER SHIFTED LEFT FOUR PLACES.
28 | */
29 |
30 |
31 |
32 |ALGORITHM_3:
33 |    PROCEDURE (ID) FIXED;
34 |       DECLARE ID CHARACTER;
35 |       RETURN (BYTE(ID) * LENGTH(ID)) & "FF";
36 |    END ALGORITHM_3 ;
37 |
38 | /*  ALGORITHM_3    PRODUCES AN INDEX FROM THE PRODUCT OF THE LENGTH
39 |     OF THE IDENTIFIER TIMES THE FIRST CHARACTER OF THE IDENTIFIER.
40 | */
41 |
42 |
43 |ALGORITHM_4:
44 |    PROCEDURE (ID) FIXED;
45 |       DECLARE ID CHARACTER, L FIXED;
46 |       L = LENGTH(ID);
47 |       RETURN (BYTE(ID) + BYTE(ID,L-1) + SHL(L,4)) & "FF";
48 |    END ALGORITHM_4 ;
49 |
50 | /*  ALGORITHM_4    PRODUCES AN INDEX FROM THE SUM OF THE EIGHT BITS
51 |     OF THE FIRST AND LAST CHARACTER AND THE LENGTH OF THE IDENTIFIER
52 |     SHIFTED LEFT FOUR PLACES.
53 | */
54 |
```

```
55 |
56 |
57 |ALGORITHM_5:
58 |    PROCEDURE (ID) FIXED;
59 |       DECLARE ID CHARACTER, L FIXED;
60 |       L = LENGTH(ID);
61 |       RETURN (BYTE(ID) + SHL(BYTE(ID,L-1),3) + SHL(L,4)) & "FF";
62 |    END ALGORITHM_5  ;
63 |
64 | /*  ALGORITHM_5    PRODUCES AN INDEX FROM THE SUM OF THE FIRST
65 |     CHARACTER AND THE LAST CHARACTER SHIFTED LEFT THREE PLACES AND
66 |     THE LENGTH OF THE IDENTIFIER SHIFTED LEFT FOUR PLACES.
67 | */
68 |
69 |
70 |
71 |ALGORITHM_6:
72 |    PROCEDURE (ID) FIXED;
73 |       DECLARE ID CHARACTER;
74 |       RETURN (SHR((BYTE(ID)*"5B5C3D5A"),20) + SHL(LENGTH(ID),4))& "FF";
75 |    END ALGORITHM_6;
76 |
77 | /*  ALGORITHM_6    PRODUCES AN INDEX BY MULTIPLYING THE FIRST CHARACTER
78 |     OF THE IDENTIFIER BY A CONSTANT AND EXTRACTING THE MOST RANDOM BITS
79 |     OF THE PRODUCT TO SUM WITH THE LENGTH OF THE IDENTIFIER IN THE
80 |     HIGH ORDER FOUR BITS.
81 | */
82 |
83 |
84 |
85 |
```

The results of the symbol table test on each algorithm is shown in Table 2.

ALGORITHM #	BUCKETS USED	SYMBOLS INTERROGATED	SECONDS CONSUMED
1	183	27,969	29.32
2	202	27,428	26.58
3	151	36,044	28.01
4	203	27,428	24.20
5	198	26,823	26.15
6	182	28,564	25.42

TABLE 2 SYMBOL TABLE TEST RESULTS

For each algorithm the number of buckets used, time consumed in seconds, and the total number of symbols interrogated is given.

The test exposed some side effects in several of the algorithms. In ALGORITHM_1 all of the single letter identifiers clustered into 9 buckets due to the padding of a space as the second character. The results of ALGORITHM_2 when compared with similar ALGORITHM_4 revealed that the masking of the two characters to a smaller bit field was a needless operation which had no affect on distribution.

Two additional tests were performed on the same algorithms using different sets of 832 staticly generated identifiers. In both sets the identifier lengths were evenly distributed from 1 to 16 and combination of the characters used in the hash and the length were unique for every identifier. Each identifier was scrambled only one time. The results given in Table 3 shows the number of unique indices and the time used in 60ths of a second. Even though all of the buckets were used by ALGORITHM_6 the distribution was not even for any of the algorithms.

ALGORITHM#	BUCKETS	TIME	BUCKETS	TIME
1	140	48	198	48
2	204	43	219	43
3	190	37	190	40
4	204	41	219	43
5	237	45	247	47
6	256	45	256	53

TABLE 3 STATIC DATA HASH TEST

The results of all of the tests are fairly inconclusive but do reveal how the overall efficiency of hash algorithms may very. It is not possible to pick one as being the best or most efficient without a knowledge of the intended application. We only wish to point out that care should be taken in selecting a hash method and that prior testing of a variety of methods is warranted.

EVALUATION OF ACCESS METHODS

The choice between the four methods presented (or others) depends upon which is most economical, a criterion easier to state than measure. In a practical situation one simply tries likely algorithms and measures gross performance against an actual computational load. We can also analyze the algorithms to give a reasonable predictive analysis to eliminate those not even near the optimum. The critical resource is memory and the twobottlenecks are memory access and memory residence. The most meaningful parameters are t, the average number of symbols in the table, t', the average number of symbols in the most local scope, t", the largest number of identifiers the algorithm must be prepared to tabulate at one time, H, the range of the scramble function, and f_1, f_2, f_3, f_4, the predicted relative frequency of the actions scope entry, scope exit, declaration of an identifier, and use of an identifier.

We want to predict the load on the mechanisms of memory residence and memory access due to the symbol table activity. We need not consider the load on the other devices of the computer (e.g., CPU) since they are largely idle and we can choose not to tabulate loads common to all methods (such as calling the access procedures). We will assume that the appearance of an

identifier is equivalent to a memory access and that certain loops in the algorithms are terminated on the average after 1/2 their maximum run. On that basis the following table defines the memory access load for each primitive symbol table action. (lnt stands for the logarithm of t to the base 2).

Memory Accesses

ACCESS METHOD	SCOPE ENTRY	SCOPE EXIT	DECLARATION	USE
Linear	7	5	7t'+14	3t+2
Hash	7	19t'+7	(8/H)t'+32	(3.5/H)t+10
Sort	7	13t +9	4t+111nt+20	111nt +12
Tree	7	10t +8	161nt+31	151nt +4

Table 4

Counting the actions in one of the small test programs for the symbol table algorithm and extrapolating for a somewhat larger program, we predict the actual access actions in the numbers:

Symbol Table Actions

Program Size	f_1 Entry	f_2 Exit	f_3 Declaration	f_4 Use
Small	5	5	20	100
Medium	10	10	100	700

Table 5

Then for each action we need to evaluate the formulas of Table 4 with weights given in Table 5.

The resulting formula

$$M = f_1M_1 + f_2M_2 + f_3M_3 + f_4M_4$$

simplifies to the formulae given in Table 6.

M_{linear}	=	335t		+ 540
M_{hash}	=	25.6t		+ 1710
M_{sort}	=	145t +	1.3201nt	+ 1680
M_{tree}	=	50t +	18201nt	+ 1095

Small program

Memory access as a function of table contents

Table 6.

In calculating the equations in Table 5 we have assumed t' - t/4 and H - 256. It might be more realistic to assign t' a constant value (such as 10) since it is not valid to assume that the number of variables declared locally in a program will increase with the size of the program. The effect of assigning a constant term would be to remove the linear term from the Linear and Hash equations. Their graph lines represented in Figure 8 would then slope slightly downward.

The only way to get a feeling for which algorithm makes the least demand on the memory access circuitry is to graph the functions (Figure 8) as a function of t over a reasonable range (t - 1, 100).

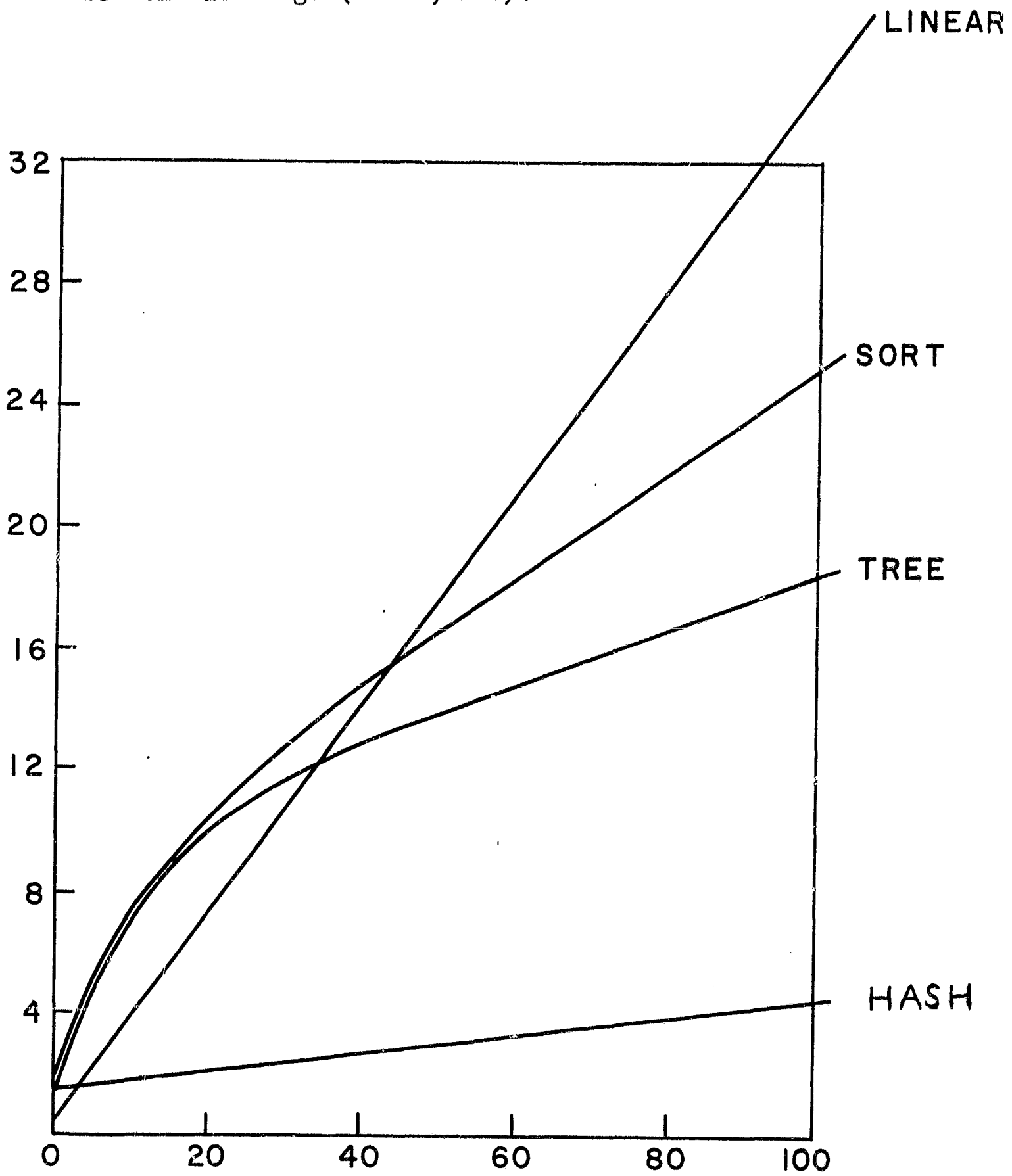


Figure 8. Memory Accesses vs. Active Table Contents

Observing Figure 8, we conclude that except for very small table sizes the hash scheme places the least load on the memory access mechanism. It does not make sense to extrapolate the curves further since they are based on frequency counts from small programs but the analysis can be repeated with new figures for the larger load as suggested in Table 5. The disappointing performance of the sorted table is due to the term 145t in the equation which is due to the necessity to re-sort upon scope exit and each new addition to the table. The larger weights reduce the importance of both actions so we should expect some improvement there. The tree algorithm should gain even more radically. We conclude that algorithm performance is sufficiently dependent upon environment that it is meaningless to ask which is "best" overall but performance can be estimated once the environment is specified.

Exercise Repeat the analysis of Table 6 and Figure 3 with the "medium" figures from Table 5.

Exercise Verify in detail the formulae in Table 4 by counting the potential memory references in each symbol table algorithm.

Another view of what has just been described is shown in Table 7. Here the order of the number of memory accesses caused by each symbol table operation is given for the four methods.

	SCOPE ENTRY	SCOPE EXIT	DECLARATION	USE
LINEAR	0	0	t''	t
HASH	0	t''	$\frac{t''}{H}$	$\frac{t}{H}$
SORT	0	t	t''	lnt
TREE	0	t''lnt	lnt	lnt

TABLE 7. MEMORY ACCESSES

Memory access is not the whole picture; memory residence is also expensive. If it were not we would simply increase the size of the scramble function so as reduce cost of hash access arbitrarily far. Ignoring the difference in actual program size, the extra memory required over the symbol table itself is given in Tables 8 and 9.

Linear	0
Hash	t'' + H
Sort	t''
Tree	2t''

Table 8
EXTRA MEMORY REQUIRED

Linear	0
Hash	356
Sort	100
Tree	200

Table 9
EXTRA MEMORY CELLS OCCUPIED

Assuming modest value of 100 for maximum table size and 256 for an 8-bit hash, we see that the speed of the hash algorithm is paid for in used memory. The combination of the figures for memory residence and memory access depends upon the computer organization in a way that cannot be predetermined. It depends upon where the critical resource is and what will happen to idle resources (e.g., can some other process use them via multiprogramming).

Exercise Pick a computer you are familiar with and attempt to weight the considerations of memory access vs. memory residence to make a choice of symbol table algorithms for a translator that is going to handle streams of small student jobs.

Exercise Gather an actual set of numbers $f_1 \ldots f_4$ by modifying an existing compiler, writing a special processing program or doing some manual examination of the input set to some compiler. Also obtain estimates of t, t', t''. How does the ratio f_4/f_3 vary with t and t'? What implication does this have on the choice of algorithms? State a mathematical analysis which allows a comparison of the algorithms over the distribution of values you have determined. Would a policy of using one algorithm for small programs and another for large programs pay large dividends?