BASIC HAL/S PROGRAMMING

06/23/97
Craig Schulenberg

SPACE SHUTTLE SYSTEMS

LOCKHEED MARTIN

Space Information Systems
1812 Space Park Drive
P.O. Box 58487
Houston, TX  77258
BASIC HAL/S PROGRAMMING CLASS INFORMATION

Course Number: OTP080
Course Length: Three two-hour classes
Intended Audience: HAL/S Programmers
Prerequisites: Some programming skills, but not necessarily HAL/S
Objectives: To provide an overview of the HAL/S language

Content:

Day 1)
Program structure, name scoping, data types, declares, arrays, initialization, replace macros, subscripting, integer/scalar conversions, expressions, assignments, conditionals and IF statements, statement groups (e.g. DO, DO FOR), procedures and functions

Day 2)
Bit strings (initialization, subscripting, expressions, assignments, comparisons), structures, raveling and unraveling, explicit conversions

Day 3)
Data storage and access (e.g. #D, #P), RIGID, exclusive and reentrant procedures and functions, CSECTs, stacks and stack frames, NAME variables, real time statements

Craig Schulenberg
Lockheed Martin
Space Shuttle Systems
COMPOOLS CONTAIN ONLY DATA (POSSIBLY REPLACE STATEMENTS).

PROGRAMS ARE "SCHEDULED" AS PROCESSES VIA THE OPERATING SYSTEM.

PROCEDURES ARE "CALLED".

FUNCTIONS ARE "INVOKED"
FORMATS

C1: COMPOOL,

DECLARES

CLOSE C1;

P1: PROGRAM

DECLARES

NESTED BLOCKS

CODE

CLOSE P1;

II: PROCEDURE (A, B) ASSIGN(C),

PARAMETER

DECLARES

ASSIGN PARAMETER

INPUT PARAMETERS

DECLARES

NESTED BLOCKS

CODE

CLOSE II;
FORMATS (CO, T.)

12: FUNCTION (A) MATRIX;
   
   PARAMETER DECLARES
   
   DECLARES
   
   NESTED BLOCKS
   
   CODE
   
   RETURN MAT;
CLOSE 12;

NOTE: FUNCTIONS HAVE NO ASSIGN PARAMETERS, BUT MUST HAVE A RETURN STATEMENT.
FORMATS (CON'T.)

COMPOOL

USED VIA REFERENCES TO DATA

PROGRAM

SCHEDULE P1 PRIORITY(100);

PROCEDURE

PROCEDURE

CALL I1(X, Y) ASSIGN(Z);  
  INPUT ARGUMENTS  ASSIGN ARGUMENT

FUNCTION

FUNCTION NAME

MATB = I2(S),  
  INPUT ARGUMENT
PROGRAM COMPLEX

COMPOOLS

C1  C2

COMSUBS

R1  R2

NESTED PROCEDURES/FUNCTIONS

P1: PROGRAM,
   DECLARE A;
I1: PROCEDURE;
   DECLARE B;
   CLOSE I1;
I2: PROCEDURE;
   DECLARE C;
   CLOSE I2;

PROGRAM
CODE

CLOSE P1;

1-5.
A COMPOOL TEMPLATE IS TEXT THAT IS VIRTUALLY IDENTICAL TO THE COMPOOL SOURCE ITSELF -- EXCEPT THAT KEYWORD "EXTERNAL" IS ADDED. IN ADDITION, THE COMPILER GENERATES A VERSION NUMBER ON AN APPENDED "D" (DIRECTIVE) CARD.
PROGRAM:

P1: PROGRAM;
    DECLARE R1;
    ...
    I1: PROCEDURE;
    ...
    CLOSE I1;
    ...
    PROGRAM
        CODE
    CLOSE P1;

PROGRAM TEMPLATE:

P1: EXTERNAL PROGRAM;
    CLOSE P1;
    D VERSION 1

COMPILER
PROCEDURE (COMSUB):

R1: PROCEDURE(A) ASSIGN(B);
    DECLARE A SCALAR;
    DECLARE B MATRIX;

OTHER DECLARATIONS

NESTED BLOCKS

CODE

CLOSE R1;

D VERSION 7
TEMPLATES (CON'T.)

TEMPLATES ARE "INCLUDED" VIA A COMPILER "D" (DIRECTIVE) CARD, E.G.,

COL 1
D INCLUDE TEMPLATE C1 {NOLIST}

RULE: FOR ALL COMPOOLS, COMSUBS, AND PROGRAMS REFERENCED BY A COMPILATION UNIT, A TEMPLATE MUST BE INCLUDED.

RULE: ALL INCLUDED TEMPLATES MUST PRECEDE THE FIRST LINE OF THE COMPILATION UNIT BEING COMPILED.
SAMPLE PROGRAM

P2: PROGRAM
DECLARE MATRIX, Q, R, S;
DECLARE X, Y, Z;
I1: PROCEDURE(V) ASSIGN(W);
DECLARE V, W;
...
CLOSE I1;

PROGRAM
SCHEDULE P1 PRIORITY(40);
...
COMSUB
CALL R1(X) ASSIGN(S);
...
COMPOOL VARIABLE
Q = C;
...
CLOSE P2,
C  OODAP+POOL REVISION 7
M  OODAP+POOL: COMPOOL;
M  DECLARE PRINT+DFCS+DATA INTEGER SINGLE INITIAL(0);
M  DECLARE JSLECT+PRINTFL BOOLEAN INITIAL(OFF);
C  C
C  **** FROM ENGINE+PRETHRUST+TRIM, OMS+ENG+CMD, DELTA+OMEGA+OMS+ENGINE, 000005
C  ENGINE+CO+CMD ****
C  C
M  DECLARE VECTOR(3) SINGLE INITIAL(9. 3.0, .3),
M  OMS1+CG, OMS2+CG; /* VEH COORDS, OMS TO CG */
M  DECLARE LBM+TO+KG CONSTANT(1. /2. 2046); 000009
M  DECLARE G+TO+MTRS+PER+SEC+SQ CONSTANT(9. 8066);
M  DECLARE LBF+TO+H CONSTANT(LBM+TO+KG G+TO+MTRS+PER+SEC+SQ);
M  DECLARE VEHICLE+CG VECTOR SINGLE INITIAL(9. 9974, -. 005, .6858);
M  DECLARE VEHICLE+MASS INITIAL(12247); 000009
M  DECLARE ARRAY(44) BOOLEAN INITIAL(OFF), JFAIL, JOHNLST;
M  DECLARE VEHICLE+INVERSE+INERTIA MATRIX(3, 3) SINGLE;
M  DECLARE VEHICLE+INERTIA MATRIX SINGLE 000010
N  INITIAL(1056182., 4067. 454, -216930. 9, 4067. 454, 7421747., 000011
N  -1355. 018, -216930. 9, -1355. 018, 7721383. ); 000011
M  DECLARE RAD+TO+DEG SCALAR SINGLE CONSTANT(57. 2957795);
M  DECLARE DEG+TO+RAD SCALAR SINGLE CONSTANT(. 0174532925);
M  DECLARE PI SCALAR DOUBLE CONSTANT(3. 141592654); 000013
M  DECLARE THO+PI SCALAR DOUBLE CONSTANT(2. PI);
M  DECLARE THR SCALAR SINGLE CONSTANT(26689. 2); 000015
M  DECLARE FT+TO+H CONSTANT(. 3048);
M  DECLARE IN+TO+M CONSTANT(FT+TO+M/12. ); 000015


SOURCE (CON'T.)

C ******* FROM MS&C ******* 000017.
C 000018
C 000019
M DECLARE BARDEQU+RATE SCALAR SINGLE INITIAL(2, DEG+TO+RAD)); 000020
M DECLARE BOOLEAN INITIAL(on), RCS+ROTATION, 000021
M ATT+HVR, RCS+TRANSLATION, RCS+TRANS+AUTO+MANUAL, 000022
M OFC+RESTART, RCS+ROT+AUTO+MANUAL, TVC+AUTO+MANUAL, 000023
M CLOSED+OPEN+LOOP+TRIM, ACCLFS+NEEDED; 000024
M DECLARE BOOLEAN INITIAL(off), TWO+AXIS, THREE+AXIS, 000025
M LCL+VERT+ATT, PAVL+SUP+CMS, TRACKING, BBO, OMS+FAIL+DETECT, 000026
M OMS1+ON+CMD, OMS2+ON+CMD, OMS+PRETHRUST, 000027
M OMS1+FAIL, OMS2+FAIL; 000028
M DECLARE ARRAY(C2) BOOLEAN INITIAL), OMS¢ARMtREG, OMS#ON¢REQs 000029
M DECLARE VECTOR SINGLE INITIAL(0), POINTING+VECTOR+CMD, 000030
M POSITION, VELOCITY, BODY+POINTING+VECTOR); 000031
M DECLARE ELF SCALAR SINGLE CONSTANT(11, DEG+TO+RAD)); 000032
M DECLARE COSELF SCALAR SINGLE CONSTANT(COS(ELF))); 000033
M DECLARE SINELF SCALAR SINGLE CONSTANT(SIN(ELF))); 000034
M DECLARE BODY+TO+N B MATRIX SINGLE CONSTANT(COSELF, 0, , SINELF, 000035
M 0, 1, 0, -SINELF, 0, , COSELF)); 000036
M DECLARE MATRIX SINGLE, NB+TO+SM, NB+TO+OA; 000037
M DECLARE SCALAR SINGLE INITIAL(0), SIN00A, SINMOA, COS00A, COSMOA) 000038
M DECLARE ARRAY(C3) SCALAR SINGLE INITIAL(0), OA+DESIRED, 000039
M AUTO+OA+DESIRED, OA+MANEUVER+TERMINAL; 000040
M DECLARE NEWORD1 BIT(16) INITIAL(BIN'0111000000111000'); 000041
M DECLARE NEWORD2 BIT(16) INITIAL(BIN'0000111000001111'); 000042
M DECLARE NEWORD3 BIT(16) INITIAL(BIN'0000101001000111'); 000043
M DECLARE ROT+OPTION+ACK ARRAY(C3) INTEGER INITIAL(O); 000044
M DECLARE TRANS+OPTION+ACK ARRAY(C3) BIT(1) INITIAL(OFF); 000045
M DECLARE VECTOR SINGLE, OMEGA+O; 000046
M CLOSE OODAP+POOL; 000047
DECLARE PRINT+DFCS+DATA INTEGER SINGLE INITIAL(0));
DECLARE JSLECT+PRINTFL BOOLEAN INITIAL(OFF));

DECLARE VECTOR(3> SINGLE INITIALCS. 3, 8, . 23>, OMS4+CQ, ONS2¢CG;
DECLARE LBM¢TO*KG CONSTANTC4. ¢ 2, 2046); DECLARE G+TOCHTRS+PER+SEC+SQ CONSTANTCS, 8966);
DECLARE LBF¢TOeN CONSTANTCLEM+TOEKG GtTOCNTRS+PERESECESQ): DECLARE VEHICLE+CO VECTOR SINGLE INITIALCS. 9974, ~, 85, DECLARE VEHICLE¢MASS INITIAL(122470)1
DECLARE ARRAVC44) BOOLEAN INITIALCOFF),

DECLARE VEHICLE+INVERSE+INERTIA MATRIX(3, 3) SINGLE;
DECLARE VEHICLE+INERTIA MATRIX SINGLE INITIAL(1036182., 4067.454, -216930.9,.
-1355.818, -216930.9, -1355.818, 7721393. ,
DECLARE RAD+TO+DEO SCALAR SINGLE CONSTANT(57. 2957795);
DECLARE DEO+TO+RAD SCALAR SINGLE CONSTANT(.0174532925),
DECLARE PI SCALAR DOUBLE CONSTANT(3.141592654);
DECLARE TWO+PI SCALAR DOUBLE CONSTANT(2. PI);
DECLARE THR SCALAR SINGLE CONSTANT(26689. 2);
DECLARE FT+TO+M CONSTANT(.3040);
DECLARE INT+TO+M CONSTANT(FT+TO+M / 12.))
COMPILED TEMPLATES (COMPOOL)

by

COMPILATION TEMPLATES (COMPOOL)

DECLARE PRINT+DFCS+DATA INTEGER SINGLE INITIAL ( 0 ); DECLARE SELECT+PRINTFL BOOLEAN INITIAL ( OFF ); DECLARE VECTOR ( 3 ) SINGLE INITIAL ( 9.3, 0, 3 ); OMS1+CO, OMS2+CO; DECLARE LBM+TO+KM CONSTANT ( 1. / 2.2046 ); DECLARE 0+TO+MTR5+PER+SEC+SQ CONSTANT ( 9.8066 ); DECLARE LBF+TO+NM CONSTANT ( LBM+TO+KM * 0+TO+MTR5+P ER+SEC+SQ ); DECLARE VEHICLE+CO VECTOR SINGLE INITIAL ( 9.9974, -0.05, 6089 ); DECLARE VEHICLE+MASS INITIAL ( 122470 ); DECLARE ARRAY ( 44 ) BOOLEAN INITIAL ( OFF ), JFAIL, JONLST, DECLARE VEHICLE+INERTIA+MATRIX SINGLE; DECLARE VEHICLE+INERTIA+MATRIX SINGLE INITIAL ( 1056182, 4067.454, -216930.9, 4067.454, 742174.7, -1355.818, -216930.9, -1355.818, 7721383.9 ); DECLARE RAD+TO+DEG SCALAR SINGLE CONSTANT ( 57.25957795 ); DECLARE DEG+TO+RAD SCALAR SINGLE CONSTANT ( 0.0174532925 ); DECLARE PI SCALAR DOUBLE CONSTANT ( 3.141592654 ); DECLARE THR SCALAR SINGLE CONSTANT ( 26689.2 ); DECLARE FT+TO+H CONSTANT ( 3048 ); DECLARE IN+TO+H CONSTANT ( FT+TO+H / 12. ); DECLARE BARBQUE+RATE SCALAR SINGLE INITIAL ( 2. DEG+TO+RAD ); DECLARE BOOLEAN INITIAL ( ON ), RC5+ROTATION, ATT+MVNR, RCS+TRANSLATION, RC5+TRANS+AUTO+MANUAL, OFC+RESTART, RCS+ROT+AUTO+MANUAL, TVC+AUTO+MANUAL, CLOSED+OPEN+LOOP+TRIM, ACCFLS+NEEDED; DECLARE BOOLEAN INITIAL ( OFF ), TWO+AXIS, THREE+AXIS, LCL+VERT+ATT, PAVL+SUB+CHDS, TRACKING, BBQ, OMS+FAIL+DET ECT, OMS1+ON+CMD, OMS2+ON+CMD, OMS1+FAIL, OMS2+FAIL; DECLARE ARRAY ( 2 ) BOOLEAN INITIAL ( OFF ), OMS+ARM+REQ, OMS+ON+REQ, OMS+ON+EQ; DECLARE VECTOR SINGLE INITIAL ( 0 ), POINTING+VECTOR+CMD, POSITI ON, VELOCITY, BODY+POINTING+VECTOR; DECLARE ELF SCALAR SINGLE CONSTANT ( 11. DEG+TO+RAD ); DECLARE COSELF SCALAR SINGLE CONSTANT ( COS ( ELF ) ); DECLARE SINEF SCALAR SINGLE CONSTANT ( SIN ( ELF ) ); DECLARE BODY+TO+MTR MATRIX SINGLE CONSTANT ( 0, -SINELF, 0, COSELF ); DECLARE MATRIX SINGLE, NA+TO+SN, NA+T 0+OA; DECLARE SCALAR SINGLE INITIAL ( 0 ); SINOGA, SINMA, COSOGA, COSMA; DECLARE ARRAY ( 3 ) SCALAR SINGLE INITIAL ( 0 ), GA+DESIRE D, AUTO+OA+DESIRED, QA+MANEUVER+TERMINAL; DECLARE NEWORD1 BIT ( 16 ) INITIAL ( BIN ’0110000000111000’ ); DECLARE NEWORD2 BIT ( 16 ) INITIAL ( BIN ’0000111000000111’ ); DECLARE NEWORD3 BIT ( 16 ) INITIAL ( BIN ’0001011000001111’ ); DECLARE ROT+OPTION+ACK ARRAY ( 3 ) INTEGER INITIAL ( 0 ); DECLARE TRANS+OPTION+ACK ARRAY ( 3 ) BIT ( 1 ) INITIAL ( OFF ); DECLARE VECTOR SINGLE, OMEGA+C; CLOSE;

D VERSION : 1

(OFC+RECON: EXTERNAL PROGRAM)

CLOSE
@@ECITOG

ECITOGEO: EXTERNAL PROCEDURE (T, R, V) ASSIGN (H, LAT, LONG, YMAQ, GAM, M, AZIM); DECLARE T SCALAR DOUBLE, R VECTOR DOUBLE, V VECTOR DOUBLE, H, LAT, LONG, YMAQ, GAMM, AZIM;
CLOSE;
D VERSION: 1

@@ROTATE

ROTATE: EXTERNAL FUNCTION (RZERO, DLONG) VECTOR DOUBLE; DECLARE SCALAR DOUBLE, DLONG; DECLARE VECTOR DOUBLE, RZERO;
CLOSE;
D VERSION: 1
Each of these represent separate compilation units and will produce templates.
A CORRECT ORDER:

C1  C3  C2  R1  R2  R3  R4  R5  P1  P2  P3  P4  P5
NAME-SCOPE

DEFINITION: THE NAME-SCOPE OF A BLOCK IS THE REGION WITHIN WHICH DATA DECLARED IN THE BLOCK IS VISIBLE. THE NAME-SCOPE ENCOMPASSES THE ENTIRE CONTENTS OF THE BLOCK INCLUDING ALL BLOCKS NESTED WITHIN IT.

RULES

(1) A NAME DEFINED IN A NAME-SCOPE IS KNOWN, AND THEREFORE ABLE TO BE REFERENCED, THROUGHOUT THAT NAME-SCOPE, INCLUDING ALL NESTED BLOCKS NOT REDEFINING IT. A NAME DEFINED IN A NAME-SCOPE IS NOT KNOWN OUTSIDE THAT NAME-SCOPE.

(2) ALL COMPOOL DATA IS CONSIDERED TO BE DEFINED IN ONE NAME-SCOPE WHICH ENCLOSES THE OUTERMOST CODE BLOCK OF THE COMPILATION UNIT.

ALSO,

THE NAME OF A CODE BLOCK IS CONSIDERED TO BELONG TO THE NAME-SCOPE IMMEDIATELY ENCLOSING THE BLOCK.
NAME-SCOPE (CON' T.)

EXAMPLE:

\[
\begin{align*}
\text{ALPHA: PROGRAM}
\quad & \text{DECLARE } X \quad \text{X KNOWN EVERYWHERE} \\
\quad & \text{DECLARE } Y \quad \text{Y KNOWN EVERYWHERE EXCEPT BETA} \\
\quad & \vdots
\text{BETA: PROCEDURE}
\quad & \text{DECLARE } Y \quad \text{NEW Y IS KNOWN IN BETA ONLY!} \\
\quad & \text{DECLARE } Z \quad \text{Z IS KNOWN IN BETA ONLY} \\
\quad & \vdots
\quad & \text{CLOSE BETA} \\
\quad & \vdots
\text{DELTA: } Y = 0 \quad \text{DELTA NOT KNOWN IN BETA} \\
\quad & \vdots
\quad & \text{CALL BETA} \\
\quad & \vdots
\quad & \text{CLOSE ALPHA}
\end{align*}
\]
NAME-SCOPE (CON'T.)

WHY IS BETA CONSIDERED TO BE IN THE NAME-SCOPE OF ALPHA?

BECAUSE ... IF IT WERE OTHERWISE

```
ALPHA: PROGRAM;
    DECLARE X;
    DECLARE Y;
    ...
BETA: PROCEDURE;
    ...
CLOSE BETA;
    ...
CALL BETA; ← ILLEGAL -- NAME BETA IS NOT VISIBLE!!
    ...
CLOSE ALPHA;
```
NAME-SCOPE (CON'T.)

VISUALIZE IT THIS WAY........

COMPOOL DATA

PROGRAM DATA

ALPHA:

PROCEDURE DATA

BETA:

GAMMA:

DELTA:

EPSILON:

OMEGA:

OUTERMOST NAME-SCOPE
NAME-SCOPE (CON'T.)

ALL RECURSION IS ILLEGAL IN HAL/S. OBVIOUS RECURSION IS DETECTED AT COMPILATION TIME. DEVIQUS RECURSION IS DETECTED BY HALLINK OR AP-101 LINKAGE EDITOR DURING STACK CALCULATION.

EXAMPLE:

```
Q: PROCEDURE
   CALL Qj  ← Recursive call (detected by compiler)
   CLOSE Qj
```
NAME-SCOPE (CON'T.)

FURTHER COMMENTS:

(1) AS A GENERAL RULE, DATA MUST BE DECLARED (DEFINED) BEFORE IT CAN BE USED.

(2) BLOCKS CAN BE CALLED FROM A POINT PRIOR TO THEIR DEFINITION, BUT WE RECOMMEND ALWAYS DEFINING A BLOCK BEFORE USING IT.

(3) BLOCK LABELS MUST BE UNIQUE THROUGHOUT A UNIT OF COMPILATION.

(4) AS AN EXCEPTION TO THE NAME-SCOPE RULES, STATEMENT LABELS ARE NOT VISIBILE WITHIN BLOCKS NESTED IN SCOPE WHERE LABEL IS DEFINED.
NAME-SCOPE (CON'T.)

THIS IS LEGAL,

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{CALL } Q;
\end{align*}
\]

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{[ } Q: \text{ PROCEDURE};
\end{align*}
\]

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{CLOSE } Q;
\end{align*}
\]

BUT WE PREFER,

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{[ } Q: \text{ PROCEDURE};
\end{align*}
\]

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{CLOSE } Q;
\end{align*}
\]

\[
\begin{align*}
\vdots
\end{align*}
\]

\[
\begin{align*}
\text{CALL } Q;
\end{align*}
\]
NAME-SCOPE (CON'T.)

WHY (?)

(1) UNLIKE HAL/S, MANY LANGUAGES DO NOT ALLOW FORWARD REFERENCES.
(2) A FORWARD REFERENCE USES UP A SYMBOL TABLE ENTRY.
(3) FORWARD REFERENCES TO FUNCTIONS REQUIRE AN ADDITIONAL DECLARE:

\[
\begin{align*}
\text{DECLARE } F & \text{ FUNCTION SCALAR;} \\
\vdots \\
S & = F(X), \\
\vdots \\
\text{CLOSE } F,
\end{align*}
\]
NAME-SCOPE (CON'T.)

Q: WHY THE SPECIAL RULE FOR STATEMENT LABELS?
A: SO ONE CANNOT USE "GO TO" TO EXIT A BLOCK.

EXAMPLE

\[
\begin{align*}
F & : \text{FUNCTION SCALAR,} \\
 & \text{DECLARE U,} \\
 & \vdots \\
 & \text{GO TO X,} \\
 & \vdots \\
 & \text{RETURN U,} \\
 & \text{CLOSE F,} \\
 & \vdots \\
X & : A = A + 1 \\
& \vdots 
\end{align*}
\]

ILLEGAL
HAL/S PRIMITIVES
3 MAJOR CLASSES

A. RESERVED WORDS (NAMES WITH SPECIAL MEANING TO THE COMPILER)
   (1) KEYWORDS
      EXAMPLES: 'IF, ELSE, GO, TO, VECTOR, TASK, IN, LOCK,
                  TRUE, READ ... 
   (2) %-MACRO NAMES
      EXAMPLES: %COPY, %SVC, %NAMECOPY
   (3) BUILT-IN FUNCTION NAMES
      EXAMPLES: ABS, SIGH, COS, LOG, UNIT, DET, RANDOM,
                  TRIM, SHR, XOR

B. IDENTIFIERS (NAMES INVENTED BY THE PROGRAMMER) -- LABELS AND
   VARIABLES
   EXAMPLES:
   [ALPHA]: PROGRAM;
   DECLARE © SCALAR;
   REPLACE 7 BY "6";

C. LITERALS (THINGS THAT EXPRESS THEIR OWN VALUE)
   EXAMPLES: 6 4.95E3 'ABC'
PRIMITIVES (CON'T.)

A. RESERVED WORDS -
SEE APPENDICES B, C, AND I OF LANGUAGE SPECIFICATION.

B. IDENTIFIERS -
RULES:
(1) MUST NOT CONFLICT WITH RESERVED WORDS!!
(2) MUST BE 32 CHARACTERS OR LESS
(3) FIRST CHARACTER MUST BE A-Z
(4) BREAK CHARACTER '_ ' (UNDERSCORE) MAY BE USED -- BUT MUST
    NOT BE FIRST OR LAST CHARACTER;
(5) ONLY ALPHANUMERIC CHARACTERS ARE LEGAL.

EXAMPLES:

<table>
<thead>
<tr>
<th>LEGAL</th>
<th>ILLEGAL</th>
</tr>
</thead>
<tbody>
<tr>
<td>STATE_VECTOR</td>
<td>3xY</td>
</tr>
<tr>
<td>Q174X203</td>
<td>_PT1</td>
</tr>
<tr>
<td>B_A_CVAL</td>
<td>VECTOR_</td>
</tr>
<tr>
<td>R</td>
<td>VECTOR</td>
</tr>
<tr>
<td></td>
<td>IN</td>
</tr>
<tr>
<td></td>
<td>A#B</td>
</tr>
</tbody>
</table>
Errors Detected:

<table>
<thead>
<tr>
<th>Error #</th>
<th>Cause</th>
<th>Fixup</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Argument outside range: (</td>
<td>x</td>
</tr>
<tr>
<td>12</td>
<td>Argument too near a singularity of the tangent function</td>
<td>Return maximum positive floating point number</td>
</tr>
</tbody>
</table>

Comments:

Error gets very large near a singularity, before error #12 is sent.

The value used in the routine for \( \pi \times 250 \) is hex '4DC90FDA' = 3.53711870600810E+15.

Registers Unsafe Across Call: F0,F1,F2,F3,F4,F5.

Algorithm:

Multiply \( x \) by \( \frac{4}{\pi} \), and give the characteristic of this to \( x'0000000000000000 \) for use as a comparand to determine nearness to a singularity.

The integer part of \( |x| \times \frac{4}{\pi} \) is the octant.

If the octant is even, let \( w = \text{fraction part of } |x| \times \frac{4}{\pi}. \)

If the octant is odd, let \( w = -(1 - \text{fraction}) \text{ part of } |x| \times \frac{4}{\pi}. \)

Next, compute two polynomials \( P(w) \) and \( Q(w) \).

If \( w \geq 2^{-6} \), then the forms of the polynomials are:

\[
P(w) = w(a_0 + a_1w^2 + a_2w^4 + w^6) \\
Q(w) = b_0 + b_1w^2 + b_2w^4 + b_3w^6
\]

If \( w < 2^{-6} \), then with \( u = w \) if \( |x| \times \frac{4}{\pi} < 1 \), and \( u = -w \) otherwise.

\[
P(w) = w(a_0 + u) \\
Q(w) = b_0 + b_2u
\]

where the values of the constants are:

\[
a_0 = X'C58AFDD0A41992D4' = -569309.04006345 \\
a_1 = X'44AFFA6393159226' = 45050.3889630777 \\
a_2 = X'C325FD4A87357CAF' = -607.8306953515 \\
b_0 = X'C5B0F82C871A3B68' = -724866.7829840012 \\
b_1 = X'4532644B1E45A133' = 206404.6948906228 \\
b_2 = X'C41926DBBB1F469B' = -6438.8583240077
\]
C. LITERALS -

ARITHMETIC

(NO DISTINCTION MADE BETWEEN INTEGERS AND SCALARS).

EXAMPLES:

+6  -80.5E-6
9   4B30
3.0 -6H-4
-3.14159 3106245

GENERIC FORM:

±ddd.ddd<exponents>

SIGNS AND DECIMAL POINT OPTIONAL

B  ~  POWER OF 2
E  ~  POWER OF 10
H  ~  POWER OF 16
PRIMITIVES (CON'T.)

C. LITERALS.

CHARACTER

TWO FORMS:

\"CCCCCCCC ... C\"

OR

CHAR <repetition> \"CCC ... CC\"

EXAMLES:

NULL STRING

\"

ABC

'ABC'

CHAR'ABC'

ABCBABC

'ABCBABC'

CHAR(2)'ABC'

ISN'T

'ISN''T'

QQQQQQ

CHAR(6)'Q'

'QQQQQQ'

NOTE: IF A SINGLE QUOTE IS DESIRED IN THE CHARACTER STRING, USE 2!

ALSO, CHARACTERS DO NOT HAVE TO BE ALPHANUMERIC NECESSARILY;

#, @, $, ...
PRIMITIVES (CON'T.)

C. LITERALS -

BOOLEAN

TRUE = ON = BIN'1'
FALSE = OFF = BIN'0'

BIT

TRUE = ON = BIN'000 ... 01'
FALSE = OFF = BIN'000 ... 00'

BIN <repetition>'bbbb' b = BINARY DIGIT
OCT <repetition>'oooo' o = OCTAL DIGIT
HEX <repetition>'hhhh' h = HEX DIGIT
DEC <repetition>'dddd' d = DECIMAL DIGIT
<repetition> = {n}) OPTIONAL

EXAMPLES:

BIN'10110' ———> 10110₂
BIN(4)'101' ———> 101101101101₂
DEC(3)'9' ———> 999₁₀
HEX(7)'F' ———> FFFFFFF₁₆
OCT'2037' ———> 2037₈
SOURCE FORMAT

(ORIENTED TOWARD IBM)

GENERALLY,

COL 1  2
  
PROCEDURE

COL 1  -- BLANK
COL 2-72 -- HAL/S STATEMENT
COL 73-78 -- STATEMENT REFERENCE NUMBER (SRN)
COL 79 -- DEFINED FOR USE AT IBM/HOUSTON

THERE ARE 2 SPECIAL TYPES OF EXCEPTIONS:

(1) COMMENT CARDS

COL 1

COLS 2-80 ARBITRARY

C THIS IS A COMMENT ....

(2) COMPILER DIRECTIVE CARDS

COL 1

D INCLUDE TEMPLATE ALPHA ...

(FOR OTHER "D" CARDS, SEE USER'S GUIDE)
SOURCE FORMAT (CON'T.)

(1) NOTE: IF CARD NUMBERS (SRNs) ARE PRESENT, OPTIONS 'SDL'
    OR 'SRN' MUST BE SPECIFIED TO THE COMPILER -- OTHERWISE,
    THE COMPILER WILL GO OUT TO COL. 80 LOOKING FOR
    TEXT.

(2) ALTHOUGH THE STREAM-ORIENTED ASPECT OF THE HAL/S SCANNER ALLOWS
    MULTIPLE STATEMENTS ON ONE CARD, THIS IS NOT GENERALLY RECOMMENDED.
    IN FACT, SINCE AN "IF ... THEN" CONSTRUCTION IS CONSIDERED TO
    BE 2 SEPARATE STATEMENTS, THINGS GET SLIGHTLY COMPLICATED.

(3) IT IS QUITE PERMISSABLE, ON THE OTHER HAND, TO EXTEND A SINGLE
    STATEMENT ACROSS MANY CARDS.

(4) ENDS OF STATEMENTS ARE INDICATED BY A SEMICOLON (;).
SOURCE FORMAT (CON'T.)

DO THIS:

\[
\begin{align*}
2 & \quad 73 \\
A &= 2; \quad 003100 \\
B &= 3; \quad 003105
\end{align*}
\]

NOT THIS:

\[
A = 2; B = 3; \quad 003100
\]

THIS IS OK:

\[
\begin{align*}
2 & \\
A &= Q + R + S + \text{SIN(T)} \quad 004000 \\
& \quad + U + V W; \quad 004005
\end{align*}
\]

AS LONG AS YOU DO NOT BREAK UP IDENTIFIERS AND RESERVED WORDS

THIS IS NOT:

\[
A = \text{CG1K}_Q\text{VECTOR}_; \quad 005000 \\
\text{FINAL} + 6; \quad 005005
\]

ALSO, (MORE ON THIS LATER)

WE PREFER: \quad \text{TO:}

\[
A, B, C = 0; \quad A = 0; B = 0; C = 0;
\]
EXAMPLE:

THIS IDENTIFIER IS REQUIRED!

ALPHA: PROGRAM;

DECLares

NESTED BLOCKS

PROGRAM CODE

CONSIDERED AN EXECUTABLE STATEMENT

CLOSE ALPHA;

OPTIONAL -- BUT IF PRESENT MUST MATCH LABEL ON PROGRAM DECLARATION.

NOTE: A PROGRAM CAN BE EXITED BY A "RETURN" STATEMENT -- BUT NORMALLY THIS IS ...
PROGRAM BLOCK (CON'T.)

... ACCOMPLISHED BY HITTING THE "CLOSE" STATEMENT.

ALSO, A "CLOSE" STATEMENT CAN HAVE A LABEL (AND CAN BE JUMPED TO
VIA A "GO TO" STATEMENT)

EXAMPLE

EXIT VIA RETURN

IF A = 0 THEN RETURN;
IF A = 2 THEN
  DO;
    B = B + 1;
    EXIT VIA JUMP TO CLOSE!
  GO TO EXEUNT;
END;
B = SIN(C) + PI;
EXIT BY FALLING
EXEUNT: CLOSE ALPHA;
EXIT INTO "CLOSE"
DECLARES (CON’T.)

FOR THE TIME BEING WE WILL RESTRICT ATTENTION TO THE FOLLOWING DATA TYPES:

<table>
<thead>
<tr>
<th>ARITHMETIC</th>
<th>STRING</th>
</tr>
</thead>
<tbody>
<tr>
<td>INTEGER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>SCALAR</td>
<td>BOOLEAN</td>
</tr>
<tr>
<td>VECTOR</td>
<td></td>
</tr>
<tr>
<td>MATRIX</td>
<td></td>
</tr>
</tbody>
</table>

AS FOR DATA ORGANIZATIONS, ONLY ARRAYS WILL BE CONSIDERED NOW.

A. INTEGERS
   - ESSENTIALLY FIXED-POINT "WHOLE" NUMBERS (SIGNED)
   - TWO CHOICES OF PRECISION
     
     360     FC
     SINGLE 2 BYTES = 1 HALFWORD
     DOUBLE 4 BYTES = (2 HALFWORDS) (1 FULLWORD)
DECLARES

(1) ALL DATA MUST BE DECLARED BEFORE IT CAN BE USED -- HAL/S HAS NO IMPLICIT DECLARATIONS.

(2) DATA IS DECLARED WITHIN A DECLARE GROUP WHICH FOLLOWS THE COMPOOL, PROGRAM, ETC. HEADER AND PRECEDES THE FIRST REAL EXECUTABLE STATEMENT.

NOTE (1): WITHIN A DECLARE GROUP, PARAMETERS SHOULD BE DECLARED FIRST.

NOTE (2): IT IS ALSO GOOD PROGRAMMING PRACTICE TO PLACE DATA DECLARED WITH THE CONSTANT ATTRIBUTE BEFORE OTHER DATA (THIS WILL BE DISCUSSED LATER UNDER THE SUBJECT OF COMPILE-TIME COMPUTATIONS).
DECLARES (CON'T.)

RANGES:

SINGLE \(-32,768 \leq i \leq 32,767\)

DOUBLE:
\(-2,147,483,648 \leq i \leq 2,147,483,647\)

USAGE:

SINGLE PRECISION INTEGERS ARE GENERALLY USED FOR LOOP VARIABLES.

CAUTION:

IT IS COMMONLY OBSERVED THAT USERS OVERESTIMATE THE RANGE OF A SHORT INTEGER, E.G.,

```
DECLARE I INTEGER;
   
DO FOR I = 1 TO 99999;
   
   END;
```
NOTE: ALTHOUGH HAL/S DOES NOT HAVE IMPLICIT DECLARATIONS, IT DOES HAVE DEFAULTS:

1. DEFAULT PRECISION IS SINGLE
2. DEFAULT DATA TYPE IS SCALAR

INTEGER DECLARATIONS

\[
\begin{cases}
\text{DECLARE } I \text{ INTEGER;} \\
\text{DECLARE } I \text{ INTEGER SINGLE;} \\
\text{DECLARE } I \text{ INTEGER DOUBLE;}
\end{cases}
\]

EQUIVALENT FORMS -- YIELDS SHORT INTEGER

NOTE: ORDER COUNTS. YOU CANNOT SAY

\[
\text{DECLARE } I \text{ DOUBLE INTEGER;}
\]
B. SCALARS

- ESSENTIALLY FLOATING-POINT (SCIENTIFIC NOTATION) NUMBERS
- TWO CHOICES OF PRECISION

\[
\begin{align*}
\text{SINGLE} & \quad 4 \text{ BYTES} = \left\{ \begin{array}{l}
2 \text{ HALFWORDS} \\
1 \text{ FULLWORD}
\end{array} \right. \\
\text{DOUBLE} & \quad 8 \text{ BYTES} = \quad 2 \text{ FULLWORDS}
\end{align*}
\]

RANGES:

Both single and double have a dynamic range from

\[ N \times 10^{-79} \text{ to } N \times 10^{75} \]

ACCURACY:

- SINGLE \( \sim 7 \) DECIMAL DIGITS
- DOUBLE \( \sim 17 \) (OR LESS)
DECLARES (CON'T.)

\{
  DECLARE S;
  DECLARE S SCALAR;
  DECLARE S SCALAR SINGLE;
\}

EQUIVALENT -- ALL YIELD SHORT SCALAR

\{
  DECLARE S DOUBLE;
  DECLARE S SCALAR DOUBLE;
\}

EQUIVALENT -- ALL YIELD LONG SCALAR

NOTE: MATRICES AND VECTORS ARE CONSIDERED TO BE MADE UP OF SCALARS.
C. VECTORS

- TWO CHOICES OF PRECISION
- LENGTHS MAY RANGE FROM
  
  \[ 2 \leq L \leq 64 \]

NOTE: DEFAULT LENGTH IS 3

VECTOR DECLARATIONS

DECLARE V VECTOR;
DECLARE V VECTOR(3);
DECLARE V VECTOR SINGLE;
DECLARE V VECTOR DOUBLE;
DECLARE V VECTOR(64) DOUBLE;
DECLARE V VECTOR(2) DOUBLE;

NOTE: HAL/S DOES NOT DISTINGUISH BETWEEN (NOR IS THERE A NEED TO) ROW AND COLUMN VECTORS.

--- MORE ON THIS LATER ---
D. MATRICES

- TWO CHOICES OF PRECISION
- ROW AND COLUMN LENGTHS MAY RANGE FROM
  \[ 2 \leq L \leq 64 \]

NOTE: DEFAULT ROW AND COLUMN LENGTHS ARE 3.

MATRIX DECLARATIONS

DECLARE M MATRIX;
DECLARE M MATRIX(3,3);
DECLARE M MATRIX SINGLE;
DECLARE M MATRIX DOUBLE;
DECLARE M MATRIX(2,64) DOUBLE;
DECLARE M MATRIX(2,2) DOUBLE;
E. CHARACTER STRINGS

PHYSICAL FORMAT:

DECLARE STRING CHARACTER(4);

NOTES: (1) DECLARED LENGTH OF CHARACTER STRING MAY RANGE FROM 1 TO 255
(2) ACTUAL LENGTH MAY RANGE FROM 0 (NULL) TO 255
F. BOOLEANS

- ESSENTIALLY DEGENERATE BIT STRINGS (BIT(1))

- ONLY TWO POSSIBLE VALUES

  TRUE = ON = BIN'1'
  FALSE = OFF = BIN'0'

**BOOLEAN DECLARATIONS**

DECLARE BOOL BOOLEAN;

NOTE: ALTHOUGH WE WILL DEFER DISCUSSION OF BIT STRINGS,

NOTE THAT IT IS ENTIRELY EQUIVALENT TO SAY,

DECLARE BOOL BIT(1);
ARRAYS

INTEGERS, SCALARS, VECTORS, MATRICES, CHARACTERS, AND BOOLEANS MAY BE ARRAYED (OR POSSESS ARRAYNESS).

TO CREATE AN ARRAY, INSERT

ARRAY(N) 1-DIM
ARRAY(N1,N2) 2-DIM
ARRAY(N1,N2,N3) 3-DIM

FOLLOWING THE IDENTIFIER IN THE DECLARE STATEMENT.
EXAMPLE:

DECLARE QMAT ARRAY(100) MATRIX;
DECLARE B ARRAY(50) BOOLEAN;
DECLARE S ARRAY(10,5) MATRIX(16,16) DOUBLE;

NOTE: ARRAY(.) MUST PRECEDE THE ATTRIBUTES.

ILLEGAL

DECLARE B BOOLEAN ARRAY(3);
IN THE FORMS:

ARRAY(N_1)
ARRAY(N_1, N_2)
ARRAY(N_1, N_2, N_3)

2 ≤ N_1 ≤ 32,767
2 ≤ N_2 ≤ 32,767
2 ≤ N_3 ≤ 32,767

THE BIGGEST DATA ITEM WE CAN THINK OF AT THIS POINT IS:

DECLARE A ARRAY(32767, 32767, 32767) MATRIX(64, 64) DOUBLES
= 1.15 x 10^{18} BYTES
INTEGERS AND SCALARS

-HAL/S UTILIZES CONTEXT TO DECIDE WHETHER LITERALS ARE INTEGERS OR SCALARS. IN GENERAL, WHENEVER A USER HAS A SCALAR THAT IS INTEGRAL IN FORM, WE ENCOURAGE WRITING IT AS THOUGH IT WERE AN INTEGER.

EXAMPLES:

IF...

DECLARE A, B, C;
THEN WRITE
A = 2 B;
INSTEAD OF
A = 2.0 B;

IN OTHER WORDS, ONE DOES NOT NEED TO APPEND ".0" TO WHOLE-NUMBER SCALARS.
INTEGERS AND SCALARS

-COMMENTS-

HAL/S IS VERY FORGIVING ABOUT MIXING INTEGERS (SINGLE OR DOUBLE) AND SCALARS (SINGLE OR DOUBLE) IN EXPRESSIONS. IMPLICIT CONVERSIONS ARE AUTOMATICALLY PERFORMED SO AS TO MAINTAIN MAXIMUM ACCURACY. OF COURSE, SINCE THESE CONVERSIONS AFFECT CPU AND CORE, USERS SHOULD BE AWARE OF THE POTENTIAL COST THAT CAN RESULT FROM THIS FREEDOM. (MORE ON THIS WHEN WE DISCUSS EXPRESSIONS AND ASSIGNMENTS.)
COMPOUND DECLARATIONS

SIMPLE DECLARATION

DECLARE S;
DECLARE M MATRIX DOUBLE;
DECLARE C CHARACTER(6);

COMPOUND DECLARATION

DECLARE S,
    M MATRIX DOUBLE,
    C CHARACTER(6);

(OR, ON ONE CARD)

DECLARE S, M MATRIX DOUBLE, C CHARACTER(6);

NOTE: IN A COMPOUND DECLARATION THE KEYWORD DECLARE APPEARS ONCE AND INDIVIDUAL ELEMENTS ARE DELIMITED BY ",".
FACTORED DECLARATIONS

IF THE IDENTIFIERS IN A COMPOUND DECLARATION HAVE SOME ATTRIBUTES IN COMMON, A FACTORED FORM MAY BE EMPLOYED:

THUS, THE PROGRESSION IS:

**SIMPLE**

```
DECLARE V1 VECTOR(4) INITIAL(0);
DECLARE V2 VECTOR(4) INITIAL(0);
DECLARE V3 VECTOR(4) INITIAL(0);
```

**TO**

**COMPOUND**

```
DECLARE V1 VECTOR(4) INITIAL(0),
V2 VECTOR(4) INITIAL(0),
V3 VECTOR(4) INITIAL(0);
```

**TO**

**FACTORED**

```
DECLARE VECTOR(4) INITIAL(0), V1, V2, V3;
```

**THIS IS A BIG KEYPUNCH SAVER:**

```
DECLARE ARRAY(16) MATRIX(4, 4) DOUBLE
INITIAL(0), M1, M2, M3;
```

**NOTE:** THE COMMA MUST BE PRESENT!
DATA INITIALIZATION

DECLARE ORDER:

DECLARE a <array> <type> <precision> <initialization>

UNARRAYED DATA

INTEGER:
   DECLARE I INTEGER INITIAL(0);
   DECLARE J INTEGER DOUBLE
       INITIAL(-20);
   DECLARE K INTEGER INITIAL(4**3);

SCALAR:
   DECLARE S INITIAL(0);  \(\text{NOTE: DO NOT NEED 0,0!}\)
   DECLARE PI SCALAR DOUBLE
       CONSTANT(3.14159265);

NOTES:

INITIAL AND CONSTANT BOTH CAUSE INITIALIZATION OF DATA. BUT
SOMETHING DECLARED WITH THE CONSTANT ATTRIBUTE CANNOT EVER BE
CHANGED.
DATA INITIALIZATION (CON'T.)

Also, data declared constant "may" turn out to be inaccessible to run-time diagnostics.

On the other hand, some data declared constant, since the value is known by the compiler, may be used in compile-time expressions.

<table>
<thead>
<tr>
<th>Reference</th>
<th>Assignable</th>
<th>Diagnostics</th>
<th>Compile-time Exp.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constant</td>
<td>Yes</td>
<td>No</td>
<td>Maybe</td>
</tr>
<tr>
<td>Initial</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>

Now, to be more precise!

If unarrayed integers, scalars, bit strings, or character strings, are declared constant, then these items can be utilized in compile-time expressions -- at the same time this implies that the data item is unavailable to diagnostics.

*** Note that vectors and matrices are ineligible!
DATA INITIALIZATION (CON'T.)

THE REASON THESE ITEMS ARE UNAVAILABLE TO DIAGNOSTICS IS THAT THEY ARE PUT IN A SPECIAL LITERAL AREA BY THE COMPILER -- SUCH ITEMS DO NOT OCCUPY STORAGE WHERE THEY ARE DECLARED.

FINALLY, USE OF CONSTANT ALLOWS A GREATER RANGE OF COMPILER OPTIMIZATION:

EXAMPLE 1
DECLARE PI SCALAR DOUBLE
    CONSTANT(3.1415926535);
DECLARE RAD_TO_DEG SCALAR
    DOUBLE CONSTANT(180/PI);
DECLARE SINS SCALAR
    CONSTANT(SIN(15/RAD_TO_DEG));

EXAMPLE 2
DECLARE V1 VECTOR CONSTANT(1, 1, 1);
DECLARE V2 VECTOR CONSTANT(V1);
ILLEGAL!
DATA INITIALIZATION (CON'T.)

EXAMPLE 3

DECLARE K1 INTEGER CONSTANT(3);
DECLARE S1 SCALAR CONSTANT(16.5);

;

W = SQRT(S1) ** K1;

THE EXPRESSION "SQRT(S1) ** K1" WILL BE EVALUATED AT COMPILE TIME. THE CODE WILL LOOK LIKE:

LE 0, \{COMPILER_CALCULATED\}
    \{CONSTANT\}
STE 0, W
DATA INITIALIZATION (CON'T.)

BOOLEAN INITIALIZATION

EQUIVALENT

\{ 
\begin{align*} 
\text{DECLARE BOOL BOOLEAN INITIAL(ON);} \\
\text{DECLARE BOOL BOOLEAN INITIAL(BIN'1');} \\
\text{DECLARE BOOL BOOLEAN INITIAL(TRUE);} \\
\text{DECLARE BOOL BOOLEAN INITIAL(OFF);} \\
\text{DECLARE BOOL BOOLEAN INITIAL(BIN'0');} \\
\text{DECLARE BOOL BOOLEAN INITIAL(FALSE);} 
\end{align*} 
\}

AND OF COURSE;

\text{DECLARE BOOL BOOLEAN constant(TRUE);} 
\text{(ALTHOUGH THIS HARDLY MAKES SENSE)}
DATA INITIALIZATION (CON'T.)

CHARACTER INITIALIZATION

(1) DECLARE C CHARACTER(3) INITIAL('');
(2) DECLARE C CHARACTER(12) INITIAL(CHAR(3)'ABCD');
(3) DECLARE C CHARACTER(255) INITIAL(CHAR(255)'A');

In (1) we get (on the 360)

<table>
<thead>
<tr>
<th>MAX</th>
<th>CUR</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

In (2) we get (on the 360)

| 12  | 12  | A   | B   | C   | D   | A   | B   | C   | D   |

I.E., A CHARACTER STRING REQUIRES 2 BYTES PLUS 1 BYTE PER CHARACTER.
A VECTOR OR MATRIX IS CONSIDERED TO BE A MULTI-VALUED DATA ITEM.

ARRAYS ARE OBVIOUSLY MULTI-VALUED.

IMPORTANT NOTE -- HAL/S MAKES LIFE EASY IF AN ENTIRE MULTI-VALUED
ITEM IS TO BE INITIALIZED TO A SINGLE VALUE:

```
DECLARE Q MATRIX(16, 16) INITIAL(0);
DECLARE R ARRAY(400) SCALAR DOUBLE INITIAL(6);
```

IF THE FOREGOING IS NOT SUITABLE THEN YOU MUST SUPPLY THE
REQUISITE NUMBER OF DATA ITEMS.

VECTOR INITIALIZATION
```
DECLARE X_VEC VECTOR CONSTANT(1, 0, 0);
DECLARE R_VEC VECTOR DOUBLE INITIAL(6, -3.5, 44.82);
```
DATA INITIALIZATION (CON'T.)

MATRIX INITIALIZATION

DECLARE I3 MATRIX CONSTANT(1, 0, 0, 0, 1, 0, 0, 0, 1);
DECLARE I4 MATRIX(4, 4) DOUBLE CONSTANT(1, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1, 0, 0, 0, 1);

ARRAY INITIALIZATION

FOR AN ARRAY, EACH ELEMENT IS INITIALIZED IN TURN:

DECLARE C ARRAY(6) CHARACTER(4) CONSTANT('ABCD', 'EFGH', 'IJKL', 'MN', 'OPQ', 'RST');
DECLARE M ARRAY(2) MATRIX(2,2) INITIAL(6, 1, 2, 3, 0, 1, 4, 5);

\[
\begin{pmatrix}
M_1 & M_2 \\
\end{pmatrix}
\]
DATA INITIALIZATION (CON'T.)

INITIALIZATION ORDER:

VECTOR -- BY INCREASING INDEX
MATRIX -- ROW-BY-ROW
ARRAY -- ELEMENT-BY-ELEMENT IN ORDER OF INCREASING INDEX,
         IF AN ELEMENT IS MULTI-VALUED IT IS INITIALIZED
         IN FULL BEFORE GOING TO THE NEXT ELEMENT.

MATRIX ORDER EXAMPLE:

\[ M = \begin{bmatrix}
  A_{11} & A_{12} & A_{13} \\
  A_{21} & A_{22} & A_{23} \\
  A_{31} & A_{32} & A_{33}
\end{bmatrix} \]

I.E., THE MATRIX IS STORED IN CORE AS
\[ [A_{11}; A_{12}; A_{13}; A_{21}; A_{22}; A_{23}; A_{31}; A_{32}; A_{33}] \]
REPLACE STATEMENT

THERE ARE TWO KINDS OF REPLACE STATEMENTS: SIMPLE AND PARAMETERIZED

SYNTAX

SIMPLE:

REPLACE IDENTIFIER BY "ANY TEXT";

MUST BE PRESENT

EXAMPLES: (LEGAL USAGES)

(1) REPLACE PREC BY "SINGLE";

DECLARE SCALAR PREC, A, B, C;

(2) REPLACE S BY "A + B + LOG(C)"

DECLARE SCALAR, A, B, C, D;

D = S;

(3) REPLACE DV BY "VECTOR DOUBLE INITIAL(0)"

DECLARE VEC1 DV,

VEC2 DV,

VEC3 DV;

(4) REPLACE N BY "4"

DECLARE V1 VECTOR(N),

M1 MATRIX(N, N),

M2 MATRIX(2, N);
REPLACE STATEMENTS (CON’T.)

(ILLEGAL USAGES)

(5) REPLACE 'SINGLE' BY "DOUBLE";
    RESERVED WORD (KEYWORD)

(6) REPLACE 'SIN' BY "SINH";
    RESERVED WORD (BUILT-IN FUNCTION)

(7) REPLACE '+' BY "/"; MISSING
    NOT AN IDENTIFIER

(8) REPLACE '16.2E3' BY "18.4E3";
    ARITHMETIC LITERAL (NOT AN IDENTIFIER)

REPLACE STATEMENTS RESULT IN SOURCE-LEVEL SUBSTITUTION. THE ACTUAL
SUBSTITUTIONS CAN BE SEEN IN THE LISTING IF CENT (¢) SIGNS ARE USED
AROUND THE REPLACE NAME, I.E.,

\[ D = \text{¢S¢}; \]

WILL RESULT IN

\[ D = A + B + \log(C); \]

IN THE LISTING.
DECLARE C21D_MCDS_EVENT EVENT LATCHED INITIAL FALSE
DECLARE C21D_MCDS_EVENT EVENT LATCHED INITIAL(2MEC(000,10)D,4#NAME(C21D_MCDS_NULL_EVT)).

DEL INPUT TABLE

STRUCTURE C21D_MCDS_EVT:
  C21D_MCDS_EVT_ARRAY INTEGER
  C21D_MCDS_EVT_SIZE INTEGER

DECLARE C21D_MCDS_EVT C21D_MCDS_EVT STRUCTURE(C21D_MCDS_BUCKET,1000000)
DECLARE C21D_MCDS_EVT C21D_MCDS_EVT STRUCTURE(C21D_MCDS_BUCKET,1000000)

DECLARE EVENT LATCHED:
  C21D_MCDS_EVTyling ARRAY ARRAY C21D_MCDS_BUCKET,1000000

DECLARE INTEGER SINGLE,C21D_MCDS_EVENT_INDEX INITIAL(0);

DEL LOAD TABLE

DECLARE C21D_MCDS_EVENT_INDEX INITIAL(INDEX1)

DEL BITE

STRUCTURE C21D_MCDS_BITE:
  C21D_MCDS_BITE_SIZE INTEGER
  C21D_MCDS_BITE_ARRAY ARRAY C21D_MCDS_BITE STRUCTURE(10)

DECLARE C21D_MCDS_BITE C21D_MCDS_BITE STRUCTURE(10)

DECLARE FOR PCs

DECLARE EXTERNAL FUNCTION FOR C21D_MCDS_MEM(11);
DECLARE C21V_A_TSIP SCALAR DOUBLE INITIAL (.1)
DECLARE C21V_A_DFCS_ENGAGE INTEGER INITIAL (0)
DECLARE C21V_A_TH_ET_STAT INITIAL (HEX '00000')

DECLARE C2IV_G_SLIM_LD INTEGER ;
DECLARE C21B_V.VALIC_CFS BOOLEAN;
DECLARE C21B_V.HDG BOOLEAN INITIAL('00');
DECLARE CUT SOFTWARE ACTIVE FLAG;

CLOSE C21_CCMPNN;
NAME = C21_INCOM, CINCODE = CO, NEW RVLAN, ETC.
REPLACE STATEMENTS (CON'T.)

NOTES:
(1) A REPLACE STATEMENT TAKES EFFECT ONLY AFTER IT APPEARS.
(2) A REPLACE STATEMENT ADHERES TO NAME-SCOPE RULES.
(3) A REPLACE CAN CAUSE TROUBLE BECAUSE IT CAN ONLY BE OVER-RIDDEN BY ANOTHER REPLACE.

EXAMPLES:
(1) DECLARE V1 VECTOR(N);  *** ERROR UNDECLARED IDENTIFIER
    REPLACE N BY "4";
    DECLARE V2 VECTOR(N);

(2) ALPHA: PROGRAM;
    REPLACE M BY "6";
    ... SINCE REPLACE IS AT PROGRAM LEVEL, IT WILL BE
    CLOSE ALPHA; EFFECTIVE EVERYWHERE
REPLACE STATEMENTS (CON'T.)

(3) POTENTIAL TROUBLE

GAMMA: EXTERNAL COMPOOL;

. . .

REPLACE I BY "8";

. . .

CLOSE GAMMA;

ALPHA: PROGRAM;

DECLARE I INTEGER *** ERROR

. . .

DO FOR I = 1 TO 10;

. . .

END;

. . .

CLOSE ALPHA;
REPLACE STATEMENTS (CON'T.)

REPLACE STATEMENTS CAN BE OVERRIDDEN.

EXAMPLE:

ALPHA: PROGRAM;
REPLACE X BY "6";
  ...;
WRITE(X) A, B, C;
  ...;
BETA: PROCEDURE;
REPLACE X BY "7";
  ...;
WRITE(X) A, B, C;
  ...;
CLOSE BETA;
  ...;
CLOSE ALPHA;
REPLACE STATEMENTS (CON'T.)

NOTE: WHENEVER REPLACE TEXT IS SUBSTITUTED FOR AN IDENTIFIER, THE IDENTIFIER IS UNDERLINED BY THE OUTPUT WRITER. (¢ SIGNS THAT CAUSE PRINTING OF THE ACTUAL TEXT DO NOT RESULT IN UNDERLINING.)

EXAMPLES:

(1) SOURCE
REPLACE Q BY "A + SIN(B)**2";

OUTPUT WRITER
M S = Q;

(2) SOURCE
S = ¢Q¢;

OUTPUT WRITER
E 2
M S = A + SIN(B) ;
HAL/S ALLOWS THREE DISTINCT TYPES OF SUBSCRIPTING: (ALL SUBSCRIPTING STARTS AT 1).

(1) COMPONENT -- APPLICABLE TO STRINGS (I.E., BIT STRINGS* AND CHARACTER STRINGS) AS WELL AS VECTORS AND MATRICES.

(2) ARRAY

(3) STRUCTURE (DEFERRED UNTIL LATER)

* SINCE BOOLEANS ARE IN EFFECT DEGENERATE BIT STRINGS, USERS MUST CONSIDER THIS WHEN SETTING UP SUBSCRIPTS FOR ARRAYS OF BOOLEANS (MORE ON THIS LATER).

IN 1-LINE SOURCE FORMAT, SUBSCRIPTING IS INDICATED BY A "$".

EXAMPLES: VECT$7 W$I MAT$(2,4) X$(I+2J+1)

NOTE THAT PARENTHESES ARE ONLY REQUIRED WHEN SUBSCRIPT IS MORE THAN ONE TOKEN!!!
THE OUTPUT WRITER OF PHASE 1, HOWEVER WILL ALWAYS PRINT SUBSCRIPTS (PARENTHESES REMOVED) ON AN "S" CARD:

EXAMPLE:

SOURCE: \[ I = Q^\{(2J + 1)\}; \]
LISTING: \[ M \quad I = Q \quad S \quad 2J + 1 \]

COMPONENT SUBSCRIPTING

CHARACTER STRINGS

\[ 1 \leq L \leq 255 \]

DECLARE STRING CHARACTER \((L)\);

- TO SELECT \( I \)TH CHARACTER FROM STRING:

\[ \text{STRING}^I \]

WHERE \( I \) IS AN INTEGER EXPRESSION AND

\[ 1 \leq I \leq L \]

EXAMPLE:

\[ \text{STRING}^\{(2I)\} \]
\[ \text{STRING}^3 \]
\[ \text{STRING}^\{(I^*3-1)\} \]
**SUBSCRIPTING (CON’T.)**

- **To select 'I' characters starting at 'J':**
  
  \[ \text{STRING$(I \text{ AT } J)$} \]

  Where \( I \) and \( J \) are integer expressions and
  
  \[
  1 \leq J \leq L \\
  0 \leq I \leq L - J + 1 \\
  0 \text{ will produce a null string.}
  \]

  **Example:**

  \[
  \text{STRING$(3 \text{ AT } K)$} \\
  \text{STRING$(I^2 \text{ AT } 2L)$}
  \]

- **To select a substring starting with the \( I \)th character and ending with the \( J \)th:**

  \[ \text{STRING$(I \text{ TO } J)$} \]

  Where \( I \) and \( J \) are integer expressions and
  
  \[
  I \leq J \\
  1 \leq I, J \leq L
  \]
SUBSCRIPTING (CON'T.)

EXAMPLES:
- STRING$(1 TO J)
- STRING$(3 TO 4)
- STRING$(1**2 TO 8)

MORE EXAMPLES

ASSUME:

DECLARE STRING CHARACTER(10) INITIAL('ABCDEFGHIJ');
DECLARE I INTEGER CONSTANT(2),
   J INTEGER CONSTANT(8),
   K INTEGER INITIAL(2);

(A) STRING$4 = 'D'
(B) STRING$0   ILLEGAL
(C) STRING$11 ILLEGAL
(D) STRING$I = 'B'
(E) STRING$(I**2) = 'D'
(F) STRING$(K**4) RUN-TIME ERROR
(G) STRING$(I**4) COMPIL-E-TIME ERROR
(H) STRING$(COS(0)+1) = 'B'
(I) STRING$(5 AT 3) = 'CDEFG'
(J) STRING$(I AT J) = 'HI'
(K) STRING$(J AT 3) = 'CDEFGHIJ'
SUBSCRIPTING (CON'T.)

(l) STRING$(5 TO 8) = 'EFGH'

(m) STRING$(4 TO 3) = 'ILLEGAL'

VECTOR

Elements (which are scalars of course) are indexed starting from 1

DECLARE V VECTOR(L); 2 ≤ L ≤ 64

- To select i^th scalar from vector:
  \[ V$I \quad (\text{a scalar}) \]
  where \( i \) is an integer expression and
  \[ i ≤ i ≤ L \]

- To select a sub-vector of length i starting from the j^th element:
  \[ V$(I AT J) \quad (\text{a vector}) \]
  where \( i \) is an integer literal and
  \[ 2 ≤ i ≤ L \]
  and \( j \) is an integer expression with
  \[ 1 ≤ j ≤ i - i + 1 \]
**SUBSCRIPTING (CON'T.)**

- To select a sub-vector (partition) starting from the *i*\(^{th}\) element and ending with the *j*\(^{th}\):
  
  \[
  V$\left(1 \text{ TO } J\right) 
  \]
  
  (A vector)

  Where *i*, *j* are integer literals and

  \[1 \leq i < j \leq L\]

**Q.** Why in the `.AT.` and `.TO.` subscripting forms, can the limits be integer expressions for character strings, but are required to be integer literals for vectors?

**A.** Because character strings in HAL/S are innately of dynamic length whereas vector lengths must be known absolutely.
A. EXAMPLES OF LEGAL VECTOR SUBSCRIPTING

LET \( V = \begin{bmatrix} 2.0 \\ 3.5 \\ -1.4 \\ 6.9 \end{bmatrix} \)

(1) \( V_1 = 2.0 \) (SCALAR)
(2) \( V_4 = 6.9 \) (SCALAR)
(3) \( V_{(2 \text{ AT } 2)} = \begin{bmatrix} 3.5 \\ -1.4 \end{bmatrix} \) (2-VECTOR)
(4) \( V_{(1 \text{ TO } 3)} = \begin{bmatrix} 3.5 \\ 2.0 \\ -1.4 \end{bmatrix} \) (3-VECTOR)

IF ADDITIONALLY WE HAVE

DECLARE I INTEGER CONSTANT(3);

THEN THE FOLLOWING ARE ALSO VALID:

(5) \( V_{(1 \text{ AT } 2)} = \begin{bmatrix} 3.5 \\ -1.4 \\ 6.9 \end{bmatrix} \) (3-VECTOR)
(6) \( V_{(1 \text{ TO } 1)} = \begin{bmatrix} 3.5 \\ 2.0 \\ -1.4 \end{bmatrix} \) (3-VECTOR)
SUBSCRIPTING (CON'T.)

B. EXAMPLES OF ILLEGAL VECTOR SUBSCRIPTING

SUPPOSE:

DECLARE V VECTOR(5) INITIAL(2,3,4,6,8);
DECLARE I INTEGER INITIAL(4);
DECLARE J INTEGER CONSTANT(5);
DECLARE K INTEGER INITIAL(1);

I.E.,

\[
V = \begin{bmatrix}
2 \\
3 \\
4 \\
6 \\
8 \\
\end{bmatrix}
\]

THEN THE FOLLOWING ARE ILLEGAL:

(1) \text{V$(0)$}
(2) \text{V$(6)$}
(3) \text{V$(J+1)$} \quad J+1 \geq \text{LENGTH}
(4) \text{V$(K \text{ AT } 1)$}
(5) \text{V$(I \text{ TO } K)$}

SUMMARY OF VECTOR COMPONENT SUBSCRIPTING:

(1) COMPONENT SUBSCRIPTING OF A VECTOR RESULTS IN A SCALAR OR A SMALLER VECTOR.

(II) IF SUBSCRIPTING RESULTS IN A VECTOR, ITS LENGTH MUST BE COMPUTABLE AT COMPILATE TIME!!!
SUBSCRIPTING (CON'T.)

MATRIX

(MOST OF THE SUBSCRIPTING RULES ARE GENERALIZATIONS OF THE VECTOR RULES.)

1. COMPONENT SUBSCRIPTING OF A MATRIX CAN RESULT IN A SCALAR, A VECTOR, OR A SUBMATRIX. IF A VECTOR OR MATRIX RESULTS, THE VECTOR LENGTH OR MATRIX ROW/COLUMN LENGTHS, RESPECTIVELY, MUST BE COMPUTABLE AT COMPILE TIME.

2. MATRIX SUBSCRIPTING INVOLVES TWO DIMENSIONS: ROW AND COLUMN. AGAIN, ELEMENTS ARE INDEXED FROM 1.

   • TO EXTRACT THE SCALAR ELEMENT IN THE I\(^{TH}\) ROW AND J\(^{TH}\) COLUMN OF THE M \(\times\) N MATRIX Q WHERE I, J ARE INTEGER EXPRESSIONS (I.E., NEED NOT BE KNOWN AT COMPILE TIME), AND 1 \(\leq\) I \(\leq\) M, 1 \(\leq\) J \(\leq\) N:

\[
Q(I, J)
\]
• TO SELECT THE I\textsuperscript{TH} ROW:
  \[ Q$(I, *) \] (AN N-VECTOR)

• TO SELECT THE J\textsuperscript{TH} COLUMN:
  \[ Q$(*, J) \] (AN M-VECTOR)

• TO SELECT A SUBMATRIX OF DIMENSIONS \( a \times b \):
  \[ Q$(a \text{ AT } I, b \text{ AT } J) \]

EXAMPLES

LET \( Q \) BE THE 4 \times 5 MATRIX:
\[
\begin{bmatrix}
6 & 3 & 9 & 0 & 1 \\
2 & 4 & 6 & 2 & 8 \\
1 & 0 & 0 & 6 & 3 \\
5 & 7 & 10 & 6 & 2
\end{bmatrix}
\]

THEN
(1) \( Q$(1,1) = 6 \) (SCALAR)
(2) \( Q$(4,5) = 2 \) (SCALAR)

\[
\begin{bmatrix}
2 \\
4 \\
6
\end{bmatrix}
\]

(3) \( Q$(2,* ) = \begin{bmatrix}
6 \\
2 \\
8
\end{bmatrix} \) (5-VECTOR)
AGAIN,

\[ Q = \begin{bmatrix}
6 & 3 & 9 & 0 & 1 \\
2 & 4 & 6 & 2 & 8 \\
1 & 0 & 0 & 6 & 3 \\
5 & 7 & 10 & 6 & 2
\end{bmatrix} \]

(4) \( Q(\ast, 3) = \begin{bmatrix}
9 \\
6 \\
0 \\
10
\end{bmatrix} \)

(5) \( Q(3 \text{ AT } 1, 3) = \begin{bmatrix}
9 \\
6 \\
0
\end{bmatrix} \)

(6) \( Q(2 \text{ TO } 3, 3 \text{ TO } 5) = \begin{bmatrix}
6 & 2 & 8 \\
0 & 6 & 3
\end{bmatrix} \)

(7) \( Q(\ast, 3 \text{ AT } 2) = \begin{bmatrix}
3 & 9 & 0 \\
4 & 6 & 2 \\
0 & 0 & 6 \\
7 & 10 & 6
\end{bmatrix} \)
PARTITIONING

EXAMPLE:

\[
\begin{array}{ccc}
H_{11} & H_{12} & H_{13} \\
H_{21} & H_{22} & H_{23} \\
H_{31} & H_{32} & H_{33} \\
H_{41} & H_{42} & H_{43} \\
H_{51} & H_{52} & H_{53} \\
H_{61} & H_{62} & H_{63} \\
\end{array}
\]

1) \( H_1 \) to 3, 1 to 3
2) \( H_{3,4} \)
3) \( H_{65} \)
4) \( H_5 \) to 6, 2 to 3
5) \( H_4, 1 \) to 3
ARRAY SUBSCRIPTING

• Arrays may be of 1, 2, or 3 dimensions (with a maximum range of 1 to 32767 in each dimension). Here we will only consider single dimension arrays.

• Array subscripting is easy for arrays of integers and scalars since no component subscripting is possible. For arrays of character strings, bit strings (and booleans), vectors, and matrices, life is more complicated.

• Subscripting an array can result in a single element or a sub-array of elements. In the latter case (as was true for vector and matrix component subscripting) the arrayness (i.e., ranges of all array dimensions) must be computable at compile time.
ARRAY SUBSCRIPTING (CON'T.)

LET "TABLE" BE AN ARRAY OF LENGTH L OF ANY DATA (INTEGER, SCALAR, BOOLEAN, CHARACTER, VECTOR, MATRIX, BIT STRING)

- TO SELECT THE I\textsuperscript{TH} ARRAY ELEMENT:
  \[
  \text{TABLE}$(1:) \quad \text{parentheses needed because "1" and ":" are 2 tokens}
  \]

WHERE I IS AN INTEGER EXPRESSION AND
\[1 \leq I \leq L\]

NOTE: IF TABLE IS AN ARRAY OF INTEGERS OR SCALARS SO THAT NO COMPONENT SUBSCRIPTING IS POSSIBLE, THEN THE COLON MAY BE OMITTED, THUS:
\[
\text{TABLE}\$1 \quad \text{suffices.}
\]

- TO SELECT A SUB-ARRAY OF LENGTH I STARTING AT THE J\textsuperscript{TH} ARRAY ELEMENT OF TABLE:
  \[
  \text{TABLE}$$(1 \text{ AT } J) \quad \text{colon optional for integer/scalar}
  \]

WHERE I IS AN INTEGER LITERAL WITH
\[1 \leq I \leq L\]
AND J IS AN INTEGER EXPRESSION WITH
\[1 \leq J \leq L - I + 1\]
ARRAY SUBSCRIPTING (CON'T.)

- To select a sub-array starting from the \(i^{th}\) array element and ending with the \(j^{th}\):
  
  \[
  \text{TABLE}$(i \text{ TO } j)$
  \]

  where \(i\) and \(j\) are both integer literals and

  \[
  1 \leq i < j \leq l
  \]

Example(1)

Declare \(s\) array(3) vector(2) initial(6, 9, 4, 2, 0, 8);

Component of array

\[
S$(1:) = \begin{bmatrix} 6 \\ 9 \end{bmatrix}
\]

(An unarrayed vector)

Partition of array

\[
S$(2 \text{ AT } 1:) = \begin{bmatrix} 6 \\ 9 \end{bmatrix}, \begin{bmatrix} 4 \\ 2 \end{bmatrix}
\]

Still an array

\[
S$(2 \text{ TO } 3:) = \begin{bmatrix} 4 \\ 2 \end{bmatrix}, \begin{bmatrix} 0 \\ 8 \end{bmatrix}
\]

Still an array

No component subscripting here.
ARRAY SUBSCRIPTING (CON'T.)

EXAMPLE (2)

```
DECLARE A ARRAY(4) INTEGER INITIAL(9, 0, -6, 3);
A$1 = 9
A$4 = 3
A$(1 TO 3) = (9, 0, -6)
A$(2 AT 2) = (0, -6)
```

EXAMPLE (3)

```
DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');
C$(2:) = 'IS'
C$(2 AT2:) = ('IS', 'HARD')
```

EXAMPLE (4)

```
DECLARE BOOL ARRAY(5) BOOLEAN INITIAL(TRUE, FALSE, ON, OFF, BIN'O');
BOOL$(4:) = OFF
BOOL$(3 AT 3:) = (ON, OFF, BIN'O')
BOOL$(1 TO 4:) = (TRUE, FALSE, ON, OFF)
```
ARRAY AND COMPONENT SUBSCRIPTING

GENERIC FORM:

TABLES (ARRAY: COMPONENT)

IN OTHER WORDS, THE "::" IS USED TO ISOLATE THE ARRAY SUBSCRIPTS FROM THE COMPONENT SUBSCRIPTS.

EXAMPLE 1

DECLARE S ARRAY(3) VECTOR(2) INITIAL(6, 9, 4, 2, 0, 8);

S$(1:2) = 9 (A SCALAR)
S$(2 AT 2:1) = (4,0) AN ARRAY OF 2 SCALARS

EXAMPLE 2

DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');

C$(2:2) = 'S'
C$(3:3) = 'R'
C$(2 AT 1:1 TO 2) = ('TH', 'IS')
C$(2 TO 3:2) = ('S', 'A')
ARRAY AND COMPONENT SUBSCRIPTINGS (CON’T.)

EXAMPLE 3

DECLARE M ARRAY(3) MATRIX INITIAL(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15
16,17,18,19,20,21,22,23,24,25,26,27);

I.E.,

\[
M = \begin{pmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{pmatrix}, \quad
\begin{pmatrix}
10 & 11 & 12 \\
13 & 14 & 15 \\
16 & 17 & 18
\end{pmatrix}, \quad
\begin{pmatrix}
19 & 20 & 21 \\
22 & 23 & 24 \\
25 & 26 & 27
\end{pmatrix}
\]

\[M_{(2:2,1)} = 13\]
\[M_{(3:3,3)} = 27\]
\[M_{(*:3,3)} = (9, 18, 27)\]

INDICATES DO FOR EACH ELEMENT OF ARRAY

\[M_{(2 AT 1:2 AT 2,2 AT 2)} = \begin{pmatrix}
5 & 6 \\
8 & 9
\end{pmatrix}, \begin{pmatrix}
14 & 15 \\
17 & 18
\end{pmatrix}\]
COMPONENT SUBSCRIPTING ONLY

IF THE DATA TO BE COMPONENT - SUBSCRIPTED (BIT AND CHARACTER STRING, VECTOR, OR MATRIX) IS ALSO ARRAYED, A TOKEN AMOUNT OF ARRAY SUBSCRIPTING MUST BE KEPT. IF TABLE IS AN ARRAYED DATA ITEM AND WE WISH TO PERFORM COMPONENT SUBSCRIPTING SIMULTANEOUSLY ON ALL ARRAY ELEMENTS THEN WE NEED THE FOLLOWING FORM:

```
TABLES(*) COMPONENT SUB)
```

WITH THIS FORM WE WILL PRODUCE A NEW ARRAY (WITH THE SAME NUMBER OF DIMENSIONS AND SAME RANGE IN EACH DIMENSION AS THE ORIGINAL) WITH EACH ELEMENT THEREOF BEING A COMPONENT-SUBSCRIPTED VERSION OF THE ORIGINAL ELEMENT.

EXAMPLE 1

DECLARE C ARRAY(3) CHARACTER(4) INITIAL('THIS', 'IS', 'HARD');
C$(#:2) = ('H', 'S', 'A')
EXAMPLE 2

LET M BE AN ARRAY 3 OF 3 x 4 MATRICES:

\[
M = \begin{pmatrix}
1 & 2 & 3 & 4 \\
5 & 6 & 7 & 8 \\
9 & 10 & 11 & 12
\end{pmatrix}, \begin{pmatrix}
13 & 14 & 15 & 16 \\
17 & 18 & 19 & 20 \\
21 & 22 & 23 & 24
\end{pmatrix}, \begin{pmatrix}
25 & 26 & 27 & 28 \\
29 & 30 & 31 & 32 \\
33 & 34 & 35 & 36
\end{pmatrix}
\]

THEN

\[M_{2:3,4} = 24\]

\[M_{2 \text{ at } 2:*,3} = \begin{pmatrix}
15 \\
19 \\
23
\end{pmatrix}, \begin{pmatrix}
27 \\
31 \\
35
\end{pmatrix}\]

\[M_{1 \text{ to } 3:2 \text{ to } 3,3 \text{ to } 4} = \begin{pmatrix}
7 & 8 \\
11 & 12
\end{pmatrix}, \begin{pmatrix}
19 & 20 \\
23 & 24
\end{pmatrix}, \begin{pmatrix}
31 & 32 \\
35 & 36
\end{pmatrix}\]

\[M_{*:2 \text{ to } 3,3 \text{ to } 4} = \text{ (the same)}\]

\[\begin{pmatrix}
7 & 8 \\
11 & 12
\end{pmatrix}, \begin{pmatrix}
19 & 20 \\
23 & 24
\end{pmatrix}, \begin{pmatrix}
31 & 32 \\
35 & 36
\end{pmatrix}\]
SUBSCRIPTING SUMMARY

RECOMMENDATION: EVEN WHERE OPTIONAL, I.E., ARRAY SUBSCRIPTING ON INTEGERS AND SCALARS, USE THE TRAILING "COLON".

- UNARRAYED DATA -

COMPONENT SUBSCRIPTING:

VAR$(7 AT 1)
VAR$3
VAR$(4 TO 5, 3 TO 4)

- ARRAYED DATA -

NO COMPONENT SUBSCRIPTING:

VAR$(7:)  ——— NEED PARENTHESES SINCE "7" AND "!" ARE 2 TOKENS.
VAR$(3 AT 2:)

ARRAY AND COMPONENT SUBSCRIPTING:

VAR$(7:3)
VAR$(4 AT 1: 3 TO 4, 1 TO 2)

COMPONENT SUBSCRIPTING ONLY: (ARRAYNESS UNCHANGED)

VAR$(*: 3 TO 4, 1 TO 2)
VAR$(*: 7)
MORE DATA INITIALIZATION

SUMMARY OF OLD MATERIAL:

1. TO INITIALIZE A SINGLE-VALUED DATA ITEM, SUPPLY 1 LITERAL VALUE IN THE INITIAL/CONSTANT LIST:
   DECLARE I INTEGER INITIAL(7);
   DECLARE S CONSTANT(9.3E-4);

2. TO INITIALIZE A MULTI-VALUED DATA ITEM, (ARRAY, VECTOR, MATRIX) WE HAVE 3 CHOICES:
   A. INITIALIZE EVERYTHING TO THE SAME VALUE: (1 ITEM IN LIST)
      DECLARE M MATRIX INITIAL(0);
      DECLARE A ARRAY(15000) INITIAL(6.5);
   B. INITIALIZE TO DIFFERENT VALUES: (NEED 1 LITERAL FOR EVERY ELEMENT)
      DECLARE M MATRIX INITIAL(1,0,0,0,1,0,0,0,1);
      DECLARE A ARRAY(15000) INITIAL(3.5, 6.5, 3.5, 6.5, 3.5,
                       6.5, 3.5, 6.5, 3.5, 6.5, ......, 6.5);
      15000 OF THEM!!
MORE DATA INITIALIZATION (CON'T.)

C. IF THE DATA IS ARAYED, SUPPLY EXACTLY ENOUGH LITERALS TO INITIALIZE A SINGLE ELEMENT OF THE ARRAY. IN THIS CASE ALL ELEMENTS OF THE ARRAY WILL BE INITIALIZED IDENTICALLY.

EXAMPLE:

INSTEAD OF

DECLARE MM ARRAY(3) MATRIX(2,2) INITIAL(1,0,0,1,1,0,0,1,1,0,0,1);

CODE

DECLARE MM ARRAY(3) MATRIX(2,2) INITIAL(1,0,0,1,0,0,1,0,0,1);

IN BOTH CASES

\[
MM = \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}, \begin{bmatrix} 1 & 0 \\ 0 & 1 \end{bmatrix}
\]
MORE DATA INITIALIZATION (CON'T.)

- SHORT CUTS -

1) USE OF REPETITION FACTOR #

EXAMPLES

DECLARE C ARRAY(6) CHARACTER(4) INITIAL(3#'ABC', 3#'DEF');
C = ('ABC', 'ABC', 'ABC', 'DEF', 'DEF', 'DEF')

DECLARE I4 MATRIX(4,4) INITIAL(1, 3#(0,0,0,1));
I4 = 
\[
\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 \\
\end{bmatrix}
\]

I.E., A SEQUENCE OF LITERALS CAN BE REPEATED.
MORE DATA INITIALIZATION (CON'T.)

- REPEATED GROUPS CAN EVEN BE NESTED

DECLARE 14 MATRIX(4,4) INITIAL(1, 3#(4#0,1));

\[ M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1 
\end{bmatrix} \]

DECLARE V ARRAY(3,2,2) INITIAL(1,2,3,2,3,1,2,3,2,3,1,2);

DECLARE V ARRAY(3,2,2) INITIAL(2#(1,2,3,2,3),1,2);

DECLARE V ARRAY(3,2,2) INITIAL(2#(1,2#(2,3)),1,2);
MORE DATA INITIALIZATION (CON'T.)

(2) PARTIAL INITIALIZATION

- SKIP OVER VALUES NOT TO BE INITIALIZED (JUST USE n#)

DECLARE M MATRIX(4,4) INITIAL(1, 3#(4#,1)),

\[ M = \begin{bmatrix}
1 & x & x & x \\
x & 1 & x & x \\
x & x & 1 & x \\
x & x & x & 1 \\
\end{bmatrix} \quad x \text{ NOT INITIALIZED} \]

- LEAVE REMAINDER OF LIST UNINITIALIZED (USE OF * SYMBOL)

DECLARE M MATRIX(4,4) INITIAL(1,2,3,*),

\[ M = \begin{bmatrix}
1 & 2 & 3 & x \\
x & x & x & x \\
x & x & x & x \\
x & x & x & x \\
\end{bmatrix} \]
AUTOMATIC/STATIC INITIALIZATION

TO DESCRIBE THE WAY IN WHICH DATA INITIALIZATION IS EFFECTED, HAL/S
HAS 2 INITIALIZATION ATTRIBUTE KEYWORDS THAT ARE OPPOSITE IN MEANING:
STATIC AND AUTOMATIC.

MOST DATA IS OF THE STATIC TYPE AND THIS IS THE DEFAULT. STATIC DATA
IS INITIALIZED WHEN THE DATA MODULES ARE BROUGHT INTO CORE (I.E., THE
DATA IS ALREADY EXISTENT IN THE LOAD MODULES). COMPOOLS ALWAYS CONSIST
OF STATIC DATA.

THE KEYWORD STATIC CAN BE SPECIFIED BY THE USER, BUT THIS IS NOT RECOMMENDED
SINCE IT CLUTTERS THE LISTING:

{ DECLARE I INTEGER INITIAL(5) STATIC;
  DECLARE I INTEGER STATIC INITIAL(5);
  DECLARE I INTEGER INITIAL(5);
}

ALL EQUIVALENT! NOTE THAT STATIC CAN PRECEDE OR FOLLOW THE INITIAL LIST.
AUTOMATIC/STATIC INITIALIZATION (CON'T.)

AUTOMATIC DATA GENERALLY RESULTS IN EXECUTABLE CODE IN THE PROLOGUE OF CODE BLOCKS TO INITIALIZE THE DATA EACH TIME THE BLOCK IS ENTERED. FOR MULTI-VALUED DATA THIS CAN BE VERY EXPENSIVE -- AND AUTOMATIC, IN GENERAL, SHOULD BE SPECIFIED ONLY WHEN IT IS REALLY NEEDED. THIS WILL BE DISCUSSED IN MORE DETAIL LATER WHEN REENTRANCY IS CONSIDERED.

THE FOLLOWING ARE EQUIVALENT:

DECLARE I INTEGER AUTOMATIC INITIAL(5);
DECLARE J INTEGER INITIAL(5) AUTOMATIC;

RESTRICTIONS ON USE OF STATIC/AUTOMATIC

• DATA INITIALIZED CONSTANT MAY NOT POSSESS EITHER THE STATIC OR AUTOMATIC KEYWORD.

• COMPOOL DATA MAY NOT HAVE EITHER A STATIC OR AN AUTOMATIC SPECIFICATION.
AUTOMATIC/STATIC INITIALIZATION (CON'T.)

EXAMPLES

1 A: COMPOOL;
   DECLARE I INTEGER INITIAL(4) STATIC; —— ILLEGAL
   . . .
   CLOSE A;

2 B: PROGRAM;
   DECLARE M MATRIX INITIAL(0) STATIC; —— (DEFAULT) RECOMMEND NEVER SPECIFYING STATIC
   . . .
   CLOSE B;

   - COMSUB EXAMPLE -

3 C: PROCEDURE;
   DECLARE I INTEGER INITIAL(10);
   . . .
   I = 4;
   CLOSE C;

   C: PROCEDURE;
   DECLARE I INTEGER INITIAL(10) AUTOMATIC;
   . . .
   I = 10 ALWAYS UPON ENTRY
   I = 4;
   . . .
   CLOSE C;

   (PROLOG OF COMSUB WILL CONTAIN CODE, I.E.:
   LA RX, 10
   ST RX, 1)
INTEGER/SCALAR CONVERSIONS

INTERPRETATION

1. In HAL/S expressions, arbitrary mixtures of integers and scalars (both of either single or double precision) can occur.

2. The following conversions are cheap (i.e., done via inline code)
   - Single integer → double integer
   - Single scalar → double scalar

   All other conversions require calls to library routines.

3. Explicit conversions are neither more nor less efficient than implicit ones.

4. Conversions are performed on the right side of an = sign without regard for what is on the left side. Only upon assignment, is the left side taken into account.
INTEGER/SCALAR CONVERSIONS (CON'T.)

INTERPRETATION (CON'T.)

5) WHEN ONLY INTEGERS APPEAR IN AN EXPRESSION, ALL INTEGERS ARE CONVERTED TO THE PRECISION OF THE MOST PRECISE INTEGER.

EXCEPTION: IN \textit{I/J} ALL INTEGERS ARE CONVERTED TO SCALARS.

6) WHEN ONLY SCALARS APPEAR IN AN EXPRESSION, ALL SCALARS ARE CONVERTED TO THE PRECISION OF THE MOST PRECISE SCALAR.

7) IF INTEGERS AND SCALARS ARE MIXED, ALL INTEGERS ARE CONVERTED TO SCALARS OF THE REQUISITE PRECISION.
** EXPRESSIONS **

** EXPRESSION ** = A MEANINGFUL COMBINATION OF OPERATORS AND OPERANDS THAT RESULTS IN SOMETHING BELONGING TO A LEGAL HAL/S DATA TYPE -- THIS DEFINES THE TYPE OF THE EXPRESSION.

** ARITHMETIC OPERATIONS **

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>**</td>
<td>EXPONENTIATION</td>
</tr>
<tr>
<td></td>
<td>INVERSION</td>
</tr>
<tr>
<td></td>
<td>TRANSPOSITION</td>
</tr>
<tr>
<td>(BLANK)</td>
<td>MULTIPLICATION</td>
</tr>
<tr>
<td></td>
<td>(VECTOR OUTER PRODUCT)</td>
</tr>
<tr>
<td></td>
<td>(MATRIX MULTIPLICATION)</td>
</tr>
<tr>
<td>+</td>
<td>VECTOR CROSS PRODUCT</td>
</tr>
<tr>
<td>-</td>
<td>VECTOR DOT PRODUCT</td>
</tr>
<tr>
<td>/</td>
<td>DIVISION</td>
</tr>
<tr>
<td>+</td>
<td>ADDITION</td>
</tr>
<tr>
<td>-</td>
<td>SUBTRACTION (NEGATION -- UNARY OP)</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

| (BLANK)            |                                        |
| 2                 | I                                       |
| (VECTOR OUTER PRODUCT) | \( \vec{V} \cdot \vec{W} \)       |
| (MATRIX MULTIPLICATION)   | \( M \cdot N \)                        |
| \( \vec{V} \)       | \( \vec{W} \)                           |
| \( A - B \)        |                                        |
| - B               |                                        |
NEGATION
- MATRIX (ALL ELEMENTS)
- VECTOR (ALL ELEMENTS)
- SCALAR
- INTEGER

ADDITION AND SUBTRACTION
- MATRIX ± MATRIX (MUST BE CONFORMABLE)
- VECTOR ± VECTOR (MUST BE CONFORMABLE)
- SCALAR ± SCALAR
- SCALAR ± INTEGER*
- INTEGER ± INTEGER

* RESULT IS SCALAR. THE INTEGER IS CONVERTED TO A SCALAR OF REQUISITE PRECISION.

NOTE: LEFT AND RIGHT-HAND SIDES CAN BE SUBSCRIPTED VARIABLES OR MORE COMPLEX EXPRESSIONS PROVIDED THEY ARE OF THE CORRECT TYPE.
EXPRESSIONS (CON'T.)

DECLARE M ARRAY(3) MATRIX INITIAL(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20,21,22,23,24,25,26,27);

\[
M = \begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}, \begin{bmatrix}
10 & 11 & 12 \\
13 & 14 & 15 \\
16 & 17 & 18
\end{bmatrix}, \begin{bmatrix}
19 & 20 & 21 \\
22 & 23 & 24 \\
25 & 26 & 27
\end{bmatrix}
\]

DECLARE V VECTOR INITIAL(1.5, 2.5, 3.5);
DECLARE S INITIAL(50.3);
DECLARE I INTEGER : INITIAL(5);

\[
\begin{align*}
I + 1 & = 6 \quad \text{(INTEGER RESULT)} \\
I + 0.3 & = 5.3 \quad \text{(SCALAR RESULT)} \\
S + 4 & = 54.3 \quad \text{(SCALAR RESULT)} \\
I + S & = 55.3 \quad \text{(SCALAR RESULT)} \\
S + V & \text{ ILLEGAL} \quad \text{(TYPE MISMATCH)} \\
S + V$2 & = 52.8 \quad \text{(SCALAR RESULT)} \\
V + M$2 & \text{ ILLEGAL} \quad \text{(TYPE MISMATCH)}
\end{align*}
\]

\[
V + M$(2:*$2) = \begin{bmatrix}
1.5 \\
2.5 \\
3.5
\end{bmatrix} + \begin{bmatrix}
11 \\
14 \\
17
\end{bmatrix} = \begin{bmatrix}
12.5 \\
16.5 \\
20.5
\end{bmatrix}
\quad \text{(VECTOR RESULT)}
\]
**EXPRESSONS (CON'T.)**

**DIVISION**

<table>
<thead>
<tr>
<th>Matrix/Scalar</th>
<th>OR</th>
<th>Matrix/Integer*</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vector/Scalar</td>
<td>OR</td>
<td>Vector/Integer*</td>
</tr>
<tr>
<td>Scalar/Scalar</td>
<td>OR</td>
<td>Scalar/Integer*</td>
</tr>
<tr>
<td>Integer*/Scalar</td>
<td>OR</td>
<td>Integer*/Integer*</td>
</tr>
</tbody>
</table>

* INTEGER WILL BE CONVERTED TO A SCALAR OF REQUISITE PRECISION.

**NOTE:** IF USERS WANT A TRUE INTEGER DIVISION, E.G., IN THE FORTRAN SENSE, THE DIV FUNCTION SHOULD BE USED:

\[ I = \text{DIV}(J,4); \]

IF \( J = 10 \), \( I \) WILL BE 2.

**EXAMPLES**

\( 1/3 \) IS 0.333333 (A SCALAR -- NOT AN INTEGER 0)

\[
\begin{bmatrix}
10 \\
20 \\
30
\end{bmatrix} \quad \text{THEN} \quad \begin{bmatrix}
5 \\
10 \\
15
\end{bmatrix}
\]
**DOT PRODUCT**  \[ \mathbf{v} \cdot \mathbf{w} \]

Standard mathematical definition with the following equivalence:
\[ \mathbf{v}^T \mathbf{w} = \mathbf{v} \cdot \mathbf{w} \]

Clearly \( \mathbf{v} \) and \( \mathbf{w} \) must have identical lengths. The result is a scalar and of same precision as most precise vector.

**CROSS PRODUCT**  \[ \mathbf{v} \times \mathbf{w} \]

Standard mathematical definition. Both \( \mathbf{v} \) and \( \mathbf{w} \) must be 3-vectors and the result is a 3-vector.

**Note:** HAL/S does not allow the form \( \mathbf{v}^T \) (i.e., the transpose of a vector). As will be seen later, context is used to determine meaning.
MULTIPLICATION

A × B

MULTIPLICATION IS INDICATED BY LOGICAL ADJACENCY, I.E., A BLANK. THE BLANK IS NOT NEEDED PRIOR TO A PARENTHESIZED EXPRESSION, BUT IT IS MANDATORY IN ALL BUT A FEW SPECIAL CASES.

EXAMPLES:

\[ 3(i + j) \]

NO BLANK NEEDED

\[ 31 \]

COMPARE

BLANK NOT NEEDED BUT IS DESIRABLE

\[ \{ 3e \} \]

WOULD LOOK LIKE ILLEGAL ARITHMETIC LITERALS

\[ \{ 3b \} \]

\[ \{ 3h \} \]

\[ m, n \]

BLANK NEEDED
MULTIPLICATION ASSUMES 5 FORMS:

1. INTEGER* SCALAR
   SCALAR INTEGER*
   INTEGER INTEGER
   SCALAR SCALAR

   * INTEGER WILL BE CONVERTED TO A SCALAR OF THE REQUISITE PRECISION.

2. INTEGER* VECTOR
   SCALAR VECTOR
   VECTOR INTEGER*
   VECTOR SCALAR
   INTEGER* MATRIX
   SCALAR MATRIX
   MATRIX INTEGER*
   MATRIX SCALAR

   { ELEMENT-BY-ELEMENT MULTIPLICATION

   * INTEGER WILL BE CONVERTED TO A SCALAR OF THE REQUISITE PRECISION.
EXPRESSIONS (CON'T.)

3) VECTOR VECTOR
defines the dyadic product (vector outer product)

\[ \vec{V} \vec{W} \]

is equivalent to the mathematical form \( V W^T \)

Example:

\[ V = \begin{bmatrix} 1 \\ 2 \\ 3 \\ 4 \end{bmatrix} \quad W = \begin{bmatrix} 1 \\ 3 \\ 5 \end{bmatrix} \]

\[ \begin{bmatrix} 1 & 3 & 5 \\ 2 & 6 & 10 \\ 3 & 9 & 15 \\ 4 & 12 & 20 \end{bmatrix} \]

4) MATRIX MATRIX
defines standard matrix multiplication, i.e.,

\[ M \cdot N \]

where \( M \) is a \( M \times N \) matrix and \( N \) is a \( N \times P \) matrix. again, note that \( M \) and \( N \) could be subscripted expressions, e.g., partitions of larger matrices.
EXPRESSIONS (CON'T.)

VECTOR  MATRIX
MATRIX  VECTOR

DEFINES STANDARD VECTOR-MATRIX MULTIPLICATION.

A. IF \( \vec{V} \times M \) THEN IF \( V \) HAS LENGTH \( L \), \( M \) MUST HAVE DIMENSIONS \( L \times P \).
   THE RESULT IS A VECTOR OF LENGTH \( P \), EQUIVALENT MATHEMATICALLY TO
   \[
   \vec{V}^T \times M = [\text{row vector}] \times [\text{matrix}]
   \]

B. IF \( M \times \vec{V} \) THEN IF \( V \) HAS LENGTH \( L \), \( M \) MUST HAVE DIMENSIONS \( M \times L \).
   THE RESULT IS A VECTOR OF LENGTH \( M \), EQUIVALENT MATHEMATICALLY TO
   \[
   M \times \vec{V} = [\text{matrix}] \times [\text{column vector}]
   \]
EXPONENTIATION (INTEGERS AND SCALARS)

1. INTEGER**INTEGER
2. INTEGER**SCALAR
3. SCALAR**INTEGER
4. SCALAR**SCALAR

In case 1, if the exponent is a non-negative integer, the result of the expression is an integer. In case 2 - 4, the result is always a scalar.

EXAMPLES

If I is an integer with I = 5 then

\[ I^{\text{0.2}} = 25 \] (integer result)

\[ I^{\text{-1.0}} = 0.2 \] (scalar result)

\[ 2^{\text{0.5}} = \sqrt{2} \] (scalar result)

Note: An exponent of .5 (or 1/2) is recognized as a special case and is entirely equivalent to using the sqrt built-in function.

\[ S^{\text{0.5}} = \text{sqrt}(S) \]
EXPONENTIATION (SQUARE MATRIX TO AN INTEGRAL POWER)

\[ M^{**1} \]

\( M \) IS A SQUARE MATRIX (\( M \times M \))
\( I \) IS AN INTEGER LITERAL

<table>
<thead>
<tr>
<th>( I )</th>
<th>RESULT</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>RESULT</td>
</tr>
<tr>
<td>( \leq -2 )</td>
<td>REPEATED PRODUCT OF INVERSE</td>
</tr>
<tr>
<td>-1</td>
<td>MATRIX INVERSE</td>
</tr>
<tr>
<td>0</td>
<td>UNIT MATRIX</td>
</tr>
<tr>
<td>1</td>
<td>NO-OP</td>
</tr>
<tr>
<td>( \geq 2 )</td>
<td>REPEATED PRODUCT</td>
</tr>
</tbody>
</table>

NOTE THAT THE VALUE OF \( I \) MUST BE KNOWN AT Compile-Time.

\( M^{**(-1)} \) IS EQUIVALENT TO USING THE MATRIX INVERSION FUNCTION:
\( \text{INVERSE}(M) \)

\( M^{**0} \) IS A FAST WAY OF BUILDING AN \( M \times M \) UNIT MATRIX DYNAMICALLY.
**Expressions (con't.)**

\[
M^{*(-2)} = (M^{*(-1)})^2
\]

\[
M^{*3} = M \cdot M \cdot M
\]

\[
M^{*0} = \begin{bmatrix}
1 & 0 \\
1 & 1 \\
0 & 1
\end{bmatrix}
\]

**Example**

If a matrix \(M\) has characteristic equation

\[A\lambda^2 + B\lambda + C = 0\]

we could code the following in HAL/S:

\[A M^{*2} + B M + C M^{*0}\]

The result will be an \(N \times N\) zero matrix.
EXPONENTIATION (TRANSPOSE OF A MATRIX)

\[ M^{**T} \]

This is exactly equivalent to using the built-in function \textsc{transpose}

\[ M^{**T} = \text{TRANSPOSE}(M) \]

If \( M \) is an \( m \times n \) matrix, \( M^T \) will be an \( n \times m \) matrix.

\textit{Note: Vectors cannot be transposed. \( V^{**T} \) is illegal.}

\textit{Note: Even if the user has defined \( T \) to be a variable, \( M^{**T} \) means transpose -- it is not the same "T"!}
COMMENTARY ON VECTOR/MATRIX OPS

RULE: IF A VECTOR TRANSPOSE (I.E., ROW VECTOR) APPEARS IN A MATHEMATICAL STATEMENT, LEAVE THE TRANSPOSE OPERATION OFF WHEN CODING THE HAL/S STATEMENTS.

<table>
<thead>
<tr>
<th>DOT PRODUCT</th>
<th>MATH</th>
<th>HAL/S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mathbf{v}^T \mathbf{w}$</td>
<td>$\mathbf{v} \cdot \mathbf{w}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CROSS PRODUCT</th>
<th>MATH</th>
<th>HAL/S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mathbf{v} \times \mathbf{w}$</td>
<td>$\mathbf{v} \times \mathbf{w}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>OUTER PRODUCT</th>
<th>MATH</th>
<th>HAL/S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mathbf{v} \mathbf{w}^T$</td>
<td>$\mathbf{v} \mathbf{w}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>MATRIX/VECTOR PRODUCT</th>
<th>MATH</th>
<th>HAL/S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mathbf{M} \mathbf{v}$</td>
<td>$\mathbf{M} \mathbf{v}$</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VECTOR/MATRIX PRODUCT</th>
<th>MATH</th>
<th>HAL/S</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$\mathbf{v}^T \mathbf{M}$</td>
<td>$\mathbf{v} \mathbf{M}$</td>
</tr>
</tbody>
</table>

NOTE: THE MATRIX TRANSPOSE OPERATION IN HAL/S IS EXPENSIVE; AVOID IT IF YOU CAN.
COMMENTARY ON VECTOR/MATRIX OPS (CON'T.)

EXAMPLES

1. MATH: \( \bar{X} = M^T \bar{Y} \)
   
   HAL/S: \( X = M**T Y \)
   
   BUT THIS CAN BE IMPROVED UPON! IN THE MATH CASE WE COULD WRITE
   
   \( \bar{X}^T = (M^T \bar{Y})^T = \bar{Y}^T M \)
   
   IN HAL/S THIS WOULD BECOME
   
   \( X = Y M \) (elimination of a matrix transpose)
   
   SINCE VECTOR TRANSPOSE IS ILLEGAL (AND NOT NEEDED!).
   
   ACTUALLY, THE LATEST RELEASE OF THE COMPILER (WITH THE GLOBAL OPTIMIZATION
   OF PHASE 1.5) MAKES THIS SUBSTITUTION AUTOMATICALLY!

2. MATH: \( \bar{X} = M^T N^T P \bar{Y} + Q^T \bar{R} \)
   
   HAL/S: \( X = (M**T)(N**T) PY + (Q**T) R \)
   
   BUT BETTER IS:
   
   HAL/S: \( X = Y(P**T) N M + R Q \)
MATRIX/VECTOR OPTIMIZATION

1 H↑ PROGRAM:
2 H↑ DECLARE MATRIX,
2 H↑ M INITIAL(1),
2 H↑ N INITIAL(2),
2 H↑ P INITIAL(3),
2 H↑ Q INITIAL(4))
3 H↑ DECLARE VECTOR,
3 H↑ X,
3 H↑ Y INITIAL(1, 2, 3),
3 H↑ R INITIAL(4, 5, 6);
Cf
E↑ - *T&T* - *T-
4 H↑ X = M N P Y + Q R:

<table>
<thead>
<tr>
<th>LA</th>
<th>M</th>
<th>N</th>
<th>P</th>
<th>Q</th>
<th>R</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>3.72(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>4.156(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.76(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.184(0.12)</td>
<td>MY653</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>4.36(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.88(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.176(0.12)</td>
<td>VM653</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>4.0(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.76(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.176(0.12)</td>
<td>VM653</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>3.168(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>4.108(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.88(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.176(0.12)</td>
<td>VM653</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>3.76(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>4.88(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.100(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.112(0.12)</td>
<td>VY253</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>3.100(0.13)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LA</td>
<td>2.144(0.10)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BAL</td>
<td>14.48(0.12)</td>
<td>VY153</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**MATRIX/VECTOR OPTIMIZATION (CON'T.)**

\[ E^t \quad - \quad T^t \quad - \quad X = Y \quad P \quad N \quad M \quad N \quad R \quad Q \]

<table>
<thead>
<tr>
<th>LA</th>
<th>3,100(0,13)</th>
</tr>
</thead>
<tbody>
<tr>
<td>LA</td>
<td>2,144(0,10)</td>
</tr>
<tr>
<td>BAL</td>
<td>14,48(0,12)</td>
</tr>
</tbody>
</table>

#4 \[ \text{CSE} = (P \quad Y) \quad N \quad M \quad R \quad Q \]

\[ \bar{X} = \text{CSE} \]

#6 \[ \bar{X} = \text{CSE} \]
CHARACTER OPERATIONS

CATENATION || OR CAT

CHAR || INTEGER*
INTEGER || CHAR
INTEGER || INTEGER*
CHAR || CHAR
CHAR || SCALAR*
SCALAR || CHAR
SCALAR || SCALAR*
SCALAR || SCALAR*
INTEGER || SCALAR*
SCALAR || INTEGER*

* INTEGERS AND SCALARS ARE IMPLICITLY CONVERTED TO CHARACTER STRINGS:

EXAMPLE: WILL BE CONVERTED TO CHARACTER
WRITE(6) 'THRUST OF ENGINE ' || 'IS' || '40095' || 'POUNDS';
BOOLEAN OPERATIONS

1  \& = AND  
2  \mid = OR  
3  \neg = NOT

(Logical Intersection)
(Logical Conjunction)
(Logical Complement)

ASSUME:

DECLARE B1 BOOLEAN INITIAL(TRUE);  
DECLARE B2 BOOLEAN INITIAL(TRUE);  
DECLARE B3 BOOLEAN INITIAL(FALSE);  
DECLARE B4 BOOLEAN INITIAL(FALSE);

\neg B1 = FALSE  
B1 \mid B2 = TRUE  
B1 \mid B3 = TRUE  
B3 \mid B1 = TRUE  
B3 \mid B4 = FALSE
EXPRESSIONS (CON'T.)

INTERSECTION (&)

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 &amp; B2</td>
<td>TRUE</td>
</tr>
<tr>
<td>B1 &amp; B3</td>
<td>FALSE</td>
</tr>
<tr>
<td>B3 &amp; B1</td>
<td>FALSE</td>
</tr>
<tr>
<td>B3 &amp; B4</td>
<td>FALSE</td>
</tr>
</tbody>
</table>

BINARY OPERATION

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>T&amp;T</td>
<td>T</td>
</tr>
<tr>
<td>T&amp;F</td>
<td>F</td>
</tr>
<tr>
<td>F&amp;T</td>
<td>F</td>
</tr>
<tr>
<td>F&amp;F</td>
<td>F</td>
</tr>
</tbody>
</table>

OTHER EXAMPLES

<table>
<thead>
<tr>
<th>Expression</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>B1 &amp; FALSE</td>
<td>FALSE</td>
</tr>
<tr>
<td>B3</td>
<td>TRUE</td>
</tr>
</tbody>
</table>

¬B1 | B2 & B3 | ¬B4 | TRUE

FALSE | FALSE | TRUE
ARITHMETIC AND CHARACTER PRECEDENCE

- ** 1 EXponentiation
- Φ 2 Multiplication
- * 3 CROSS PRODUCT
- • 4 DOT PRODUCT
- / 5 DIVISION
- [+ 6 ADDITION
- [− 6 SUBTRAction, NEGATION
- || 7 CHARACTER CATENATION

- SEQUENCES OF OPERATIONS OF THE SAME PRECEDENCE ARE EVALUATED FROM LEFT TO RIGHT -- EXCEPT FOR ** AND /, WHICH ARE EVALUATED FROM RIGHT TO LEFT,
  I.E.,
  \[ \frac{A}{B/C/D} = \frac{(AC)}{(BD)} \quad \frac{A}{B(C/D)} = A/(BD/C) = AC/BD \]

- SEQUENCES OF MULTIPLICATIONS ARE SOMETIMES REORDERED TO MINIMIZE THE NUMBER OF ELEMENTAL PRODUCTS INVOLVED.
DIVIDE EXAMPLE

1 NT: PROGRAM:
2 NT: DECLARE INITIAL(1);
3 NT: S1, S2, S3, S4, S5, S6, S7, S8, S9, S10, S11, S12, S13, S14, S15, S16, S17, S18, S19,
4 NT: S20;
5 NT: DECLARE T;
6 NT: T = S1 S2 S3 S4 S5 S6 S7 S8 S9 S10 S11 S12 S13 S14 S15 S16 S17 S18 S19 S20;

000002A
000009 7805 0002
00000A 600D 0006
00000B 6015 000A
00000C 6016 000E
00000D 6029 0012
00000E 602D 0016
00000F 6035 001A
000100 603D 001E
000101 6045 0022
000102 604D 0026
000103 7A19 002C
000104 6231 0018
000105 6231 0019
000106 6239 001C
000107 6231 001A
000108 6241 0020
000109 6209 0004
00010A 6249 002C
00010B 6229 0014
00010C 6251 0028
00010D 7C03
00010E 64E2
00010F 3E53 002A

ST#4 EQU *
#8TEST T CSECT ESDID= 0001
LE 0,2(1)
ME 0,6(1)
ME 0,10(1)
ME 0,14(1)
ME 0,18(1)
ME 0,22(1)
ME 0,26(1)
ME 0,30(1)
ME 0,34(1)
ME 0,38(1)
LE 2,12(1)
ME 2,24(1)
ME 2,28(1)
ME 2,32(1)
ME 2,4(1)
ME 2,36(1)
ME 2,28(1)
ME 2,40(1)
LER 4,0
MER 4,2
STE 4,42(1)
DIVIDE EXAMPLE (CON’T.)

C↑
6 M↑ CLOSE TESTI)

ST#5  EQU *
DER 0.2
STE 0.42(1)

ST#6  EQU *
EQU *

LBL#2  EQU *
SYC 44(1)
H’21’
MATRIX INVERTER (4x4)

INVERTER:PROGRAM;
REPLACE PREC BY "SINGLE";
DECLARE MATRX(4,4) PREC,Q,Q1,Q2;
DECLARE I MATRX(4,4) PREC-CONSTANT(1,3)(0,0,0,0,1));
DECLARE MATRX(2,2) PREC,C,S,T,M,N;
INVERT:FUNCTION(INMAT) MATRX(2,2) PREC;
DECLARE MATRX(2,2) PREC,INMAT;
RETURN MATRXS @FREC,2,2)(INMAT$(2,2),-INMAT$(1,2),
- INMAT$(2,1), INMAT$(1,1)) / INMAT$(1,1) INMAT$(2,2)
-1 INMAT$(1,2) INMAT$(2,1));
CLOSE INVERT;
ON ERRORS (10151)-RETURN;
DO WHILE TRUE;
READ(5) Q;
Q1=Q**(-1);
S=INVERT(Q$(1 TO 2,1 TO 2));
T=S Q$(1 TO 2,3 TO 4);
G=Q$(3 TO 4,1 TO 2);
N=INVERT(Q$(3 TO 4,3 TO 4)-C T);
M=N C S;
Q2$(1 TO 2,1 TO 2)=S-T M;
Q2$(1 TO 2,3 TO 4)=- T N;
Q2$(3 TO 4,1 TO 2)=M;
Q2$(3 TO 4,3 TO 4)=N;
MATRIX INVERTER (4x4) (CON’T.)

```
WRITE(6) 'Q = ',C,SKIP(2);
WRITE(6) 'DET(Q) = ',DET(Q),SKIP(2);
WRITE(6) 'CL = ',C1,SKIP(2);
WRITE(6) 'DET(CL) = ',DET(CL),SKIP(2);
WRITE(6) 'Q2 = ',C2,SKIP(2);
WRITE(6) 'DET(Q2) = ',DET(Q2),SKIP(2);
WRITE(6) 'I - C - C1 = ',I-Q*Q1,SKIP(2);
WRITE(6) 'I - Q*Q2 = ',I-Q*Q2,SKIP(2);
END;
CLOSE;
```
HAL/S COMPILATION

1 M | PROGRAM:
2 M | REPLACE PREC BY "SINGLE";
3 M | DECLARE MATRIX(4, 4) PREC Q1, Q1, Q1, Q2;
4 M | DECLARE MATRIX(4, 4) PREC CONSTANT(1, 3, 1, 0, 0, 0, 0, 0, 1);
5 M | DECLARE MATRIX(2, 2) PREC C, S, T, M, N;

6 M | INVERT:
7 M | FUNCTION(IN_MAT) MATRIX(2, 2) PREC;
8 M | DECLARE MATRIX(2, 2) PREC IN_MAT;
9 M | RETURN MATRIX;
10 M | close INVERT;
MATRIX INVERTER (4x4)

10 M1 ON ERROR 1015
11 M1 RETURN
12 M1 DO WHILE TRUE
13 M1 READ(S1, Q1)
14 M1 Q1 = Q1 - 1
15 M1 S = INVERT(Q1 TO 2, 1 TO 2)
16 M1 T = S 0 TO 2, 3 TO 4
17 M1 C = Q3 TO 4, 1 TO 2
18 M1 N = INVERT(C3 TO 4, 3 TO 4 - C T 1)
19 M1 M = -N C S1
20 M1 Q2 1 TO 2, 1 TO 2
21 M1 Q2 1 TO 2, 3 TO 4
22 M1 Q2 3 TO 4, 1 TO 2
23 M1 Q2 3 TO 4, 3 TO 4
24 M1 WRITE(6) - C = S - Q1, SKIP(2)
25 M1 WRITE(6) - DET(Q) = ' , DET(Q), SKIP(2)
MATRIX INVERTER (4x4)

25 F
WRITE(6) 'Q1 = %1, Q1, SKIP(2);  

27 E
WRITE(6) 'DET(Q1) = %1, DET(Q1), SKIP(2);  

28 EM
WRITE(6) 'Q2 = %1, Q2, SKIP(2);  

29 E
WRITE(5) 'DET(Q2) = %1, DET(Q2), SKIP(2);  

30 EM
WRITE(6) '1 - Q1 = %1, 1 - Q1, SKIP(2);  

31 E
WRITE(6) '1 - Q2 = %1, 1 - Q2, SKIP(2);  

32 EM
END

33 EM
CLOSE;
**MATRIX INVERTER (4x4)**

**1ST MATRIX**

\[
\begin{align*}
\text{HAL} & = 360.33 \quad \text{START TIME} = 1127103.55 \quad \text{DA} = 175/271 \\
0 & = 4.0000000 \times 10^0 \\
\text{DET} & = 2.083 \times 10^4
\end{align*}
\]

**J1**

\[
\begin{align*}
3.0499007 & \times 10^4 \\
4.696326 & \times 10^0 & - 0.3085872 & \times 10^0 \\
-1.831437 & \times 10^1 & 6.1600062 & \times 10^1 \\
-2.4165513 & \times 10^1 & 8.3166076 & \times 10^1 \\
\text{DET(J1)} & = -4.7980782 \times 10^{-1}
\end{align*}
\]

**J2**

\[
\begin{align*}
3.0499011 & \times 10^4 \\
4.696326 & \times 10^0 & - 0.3085506 & \times 10^0 \\
-1.831438 & \times 10^1 & 6.1600072 & \times 10^1 \\
-2.4165503 & \times 10^1 & 8.3166062 & \times 10^1 \\
\text{DET(J2)} & = -4.7993131 \times 10^{-1}
\end{align*}
\]

**J - Q1**

\[
\begin{align*}
0 & \\
-0.3085872 & \times 10^0 & 1.4177550 & \times 10^0 & 3.5282852 & \times 10^{-7} & 6.71157316 & \times 10^{-7} \\
-2.4165513 & \times 10^1 & 8.3166076 & \times 10^1 & 3.1603893 & \times 10^0 & 1.69653306 & \times 10^0 \\
\end{align*}
\]

**J - Q2**

\[
\begin{align*}
2.8610229 & \times 10^4 \\
1.8973259 & \times 10^0 & - 3.6114972 & \times 10^0 & 1.7114770 & \times 10^{-7} & 5.50890114 & \times 10^{-7} \\
-1.9337113 & \times 10^1 & 2.3740047 & \times 10^1 & 2.1928996 & \times 10^{-8} & 1.9238928 & \times 10^{-8}
\end{align*}
\]
### Matrix Inverter (4x4)

#### 2nd Matrix

<table>
<thead>
<tr>
<th>0</th>
<th>5.000000E+00</th>
<th>7.000000E+00</th>
<th>6.000000E+00</th>
<th>5.000000E+00</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7.000000E+00</td>
<td>1.000000E+01</td>
<td>8.000000E+00</td>
<td>7.000000E+00</td>
</tr>
<tr>
<td>2</td>
<td>6.000000E+00</td>
<td>1.000000E+00</td>
<td>1.000000E+00</td>
<td>9.000000E+00</td>
</tr>
<tr>
<td>3</td>
<td>5.000000E+00</td>
<td>7.000000E+00</td>
<td>9.000000E+00</td>
<td>1.000000E+00</td>
</tr>
</tbody>
</table>

**Det(Q1) = 1.000017E+00**

\[
\begin{align*}
Q_1 &= \begin{pmatrix} 6.799163E+01 & -4.059493E+01 & -1.699794E+01 & 9.998764E+00 \\ -4.059493E+01 & 2.459487E+01 & 9.998764E+00 & -5.999243E+00 \\ -1.699794E+01 & 9.998764E+00 & 4.999493E+00 & -2.999693E+00 \\ 9.998764E+00 & -5.999246E+00 & -2.999695E+00 & 1.999815E+00 \end{pmatrix} \\
\end{align*}
\]

**Det(Q1) = 1.000045E+00**

\[
\begin{align*}
Q_2 &= \begin{pmatrix} 6.800000E+00 & -4.100000E+00 & -1.700000E+00 & -1.000000E+00 \\ -4.100000E+00 & 7.500000E+00 & 1.000000E+01 & -6.000000E+00 \\ -1.700000E+00 & 7.500000E+00 & 5.000000E+00 & -3.000000E+00 \\ -1.000000E+00 & -6.000000E+00 & -3.000000E+00 & 2.000000E+00 \end{pmatrix} \\
\end{align*}
\]

**Det(Q2) = 1.000056E+00**

\[
\begin{align*}
\end{align*}
\]

\[
\begin{align*}
1 - Q_2 &= \begin{pmatrix} 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \\ 0.0 & 0.0 & 0.0 & 0.0 \end{pmatrix} \\
\end{align*}
\]
EXPRESSIONS (CON'T.)

PRECEDENCE EXAMPLE:

(from Programmer's Guide)

RESULT OF STEP' 1 1 1 IS' S1 + S2**2 - V1, V2/2/2

1 2 9 4 3 8 5 7 6

Also,

\[ \overline{V, \overline{\overline{W} \cdot X}} \cdot = \overline{V, (\overline{W} \cdot X)} \cdot \quad \text{(A SCALAR)} \]

\[ V \cdot \overline{W} \cdot X \cdot = (V \cdot \overline{W}) \cdot X \cdot \]

\[ 2**3**2 = 2**(3**2) = 2**9 = 512 \]

\[ \text{NOT!} \quad (2**3)**2 = (8)**2 = 64 \]

\[ 4/3/10/3 = 4/3/(10/3) = 4/(9/10) = 40/9 \]

\[ \text{NOT!} \quad (4/3)/10/3 = (4/30)/3 = 4/90 \]
BOOLEAN PRECEDENCE

\[
\begin{array}{c|c|c}
\text{HI} & \text{LO} \\
\hline
\text{\texttt{NOT}} & 1 & \text{COMPLEMENT} \\
& 2 & \text{INTERSECTION} \\
& 3 & \text{CONJUNCTION}
\end{array}
\]

* Sequences of operands of the same precedence are evaluated from left to right.

**Examples**

1. IF \(B_1 \rightarrow B_2 \lor B_3 \rightarrow B_4\) THEN DO;
   
   EQUIVALENT TO:
   
   IF \((B_1) \lor (\neg B_2) \lor (B_3 \lor (\neg B_4))\) THEN DO;

2. IF \(B_1 \land B_2 \land B_3 \land B_4 \land B_5 \land B_6\)...
   
   EQUIVALENT TO:
   
   IF \((B_1 \land B_2) \land (B_3 \land B_4) \land (\neg B_5 \land \neg B_6))\)....
OVERIDING PRECEDENCE ORDER

- PRECEDENCE ORDER CAN ALWAYS BE ALTERED BY USE OF PARENTHESES.

EXAMPLES

1. IF (B1\(\neg\)(B2\|B3))\&\&B4 THEN DO;
2. IF B1\&(B2\|B3)\&(B4\|\neg B5)\&\&B6 ...
3. (A**B)**C
4. (A/B)/(C/D)
EXPLICIT CONVERSIONS

A. VECTOR CONVERSION

VECTORS CAN BE DYNAMICALLY CONSTRUCTED VIA THE CONSTRUCTION

\texttt{VECTOR$(\ell)$ (exp$_1$, exp$_2$, ..., exp$_\ell$)}

TO CREATE AN $\ell$-VECTOR. IF THE DEFAULT IS DESIRED THE LENGTH
SPECIFICATION CAN BE OMITTED, I.E.,

\texttt{VECTOR(exp$_1$, exp$_2$, exp$_3$)}

IF \texttt{exp} IS AN ARRAY OF $\ell$ INTEGERS OR SCALARS, WE CAN SAY:

\texttt{VECTOR$(\ell)$ (exp)}

- ALL EXPRESSIONS (exp$_1$) MUST BE OF INTEGER OR SCALAR TYPE.
  (AGGREGATES OF THESE TYPES MAY ALSO BE USED.)

- THE RESULT OF THE VECTOR CONVERSION FUNCTION IS A SINGLE-PRECISION
  VECTOR. IF A DOUBLE-PRECISION VECTOR IS NEEDED USE THE FORM:

\texttt{VECTOR$(\texttt{DOUBLE,} \ell)$ (exp$_1$, exp$_2$, ...)}

CLEARLY \hspace{1cm} 2 \leq \ell \leq 64
EXPLICIT CONVERSIONS (CON'T.)

VECTOR EXAMPLES

1. \( \text{VECTOR}(1,0,0) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \) SINGLE PRECISION (3-VECTOR)

2. \( \text{VECTOR\$}(\text{DOUBLE})(1,0,0) = \begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix} \) DOUBLE PRECISION (3-VECTOR)

3. \( \text{VECTOR\$}(\text{DOUBLE}, 4)(3, 2*2, 8-6, \sin(0)) = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 0 \end{bmatrix} \) SINGLE PRECISION (4-VECTOR)

4. \( \text{VECTOR\$}(\text{DOUBLE}, 4)(3, 2*2, 8-6, \sin(0)) = \begin{bmatrix} 3 \\ 4 \\ 2 \\ 0 \end{bmatrix} \) DOUBLE PRECISION (4-VECTOR)

5. DECLARE Q ARRAY(12) INITIAL(1,2,3,4,5,6,7,8,9,10,11,12)

6. \( \text{VECTOR\$}(\text{DOUBLE}, 5)(Q\$(5 \text{ AT } 6)) = \begin{bmatrix} 6 \\ 7 \\ 8 \\ 9 \\ 10 \end{bmatrix} \)
EXPLICIT CONVERSIONS (CON'T.)

B. MATRIX CONVERSIONS

Matrices can also be constructed dynamically. Use the construction:

\[
\text{MATRIX}_x^c(r, c) (\exp_1, \exp_2, \ldots \exp_{rc})
\]

To create an \( r \times c \) single precision matrix,

\[
2 \leq r \leq 64 \\
2 \leq c \leq 64 \\
r \text{ and } c \text{ default to a } 3 \times 3 \text{ matrix, e.g.,} \\
\text{MATRIX}(\exp_1, \exp_2, \ldots \exp_9)
\]

will create a \( 3 \times 3 \) matrix (single precision).

If \exp is an array of \( rc \) integers or scalars, we can say

\[
\text{MATRIX}_x^c(r, c)(\exp)
\]

* All expressions (\exp) must be of integer or scalar type.
* If a double-precision matrix is needed, use the form:

\[
\text{MATRIX}_x^c(@DOUBLE, r, c)(\exp_1, \ldots \exp_{rc})
\]
EXPLICIT CONVERSIONS (CON'T.)

- MATRICES ARE ASSEMBLED ROW-BY-ROW FROM THE LIST (JUST AS IS DONE IN INITIAL LISTS IN DECLARE STATEMENTS)

**MATRIX EXAMPLES**

1. \[ \text{MATRIX}(1,0,0,0,1,0,0,0,0,1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \] A 3x3 SINGLE-PRECISION MATRIX

   \[ \text{MATRIX}(@\text{DOUBLE})(1,0,0,0,1,0,0,0,0,1) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \] A 3x3 DOUBLE-PRECISION MATRIX

2. \[ \text{MATRIX}((2,2)(4,5,7,9)) = \begin{bmatrix} 4 & 5 \\ 7 & 9 \end{bmatrix} \] A 2x2 SINGLE-PRECISION MATRIX

   \[ \text{MATRIX}(@\text{DOUBLE},2,2)(4,5,7,9) = \begin{bmatrix} 4 & 5 \\ 7 & 9 \end{bmatrix} \] A 2x2 DOUBLE-PRECISION MATRIX
EXPLICIT CONVERSIONS (CON'T.)

MATRIX EXAMPLES

3. DECLARE Q ARRAY(50) SCALAR DOUBLE INITIAL(10#1, 10#2, 10#3, 10#4, 10#5);

\[
\begin{bmatrix}
1 & 1 & 1 & 1 & 1 \\
2 & 2 & 2 & 2 & 2 \\
2 & 2 & 2 & 2 & 2 \\
3 & 3 & 3 & 3 & 3
\end{bmatrix}
\]

4. MATRIX(3#(1/2, SQRT(2)/2, SQRT(3)/2))

\[
\begin{bmatrix}
1/2 & 1/2 \sqrt{2} & 1/2 \sqrt{3} \\
1/2 & 1/2 \sqrt{2} & 1/2 \sqrt{3} \\
1/2 & 1/2 \sqrt{2} & 1/2 \sqrt{3}
\end{bmatrix}
\]

Note that the repetition symbol '#', previously introduced in initial lists, is also applicable here.
ASSIGNMENTS

There are 3 classes of assignments depending on whether the left-hand side of the statement is:

1. Arithmetic (matrix, vector, integer, or scalar)
2. Character
3. Bit/Boolean

General form: \( L = R \)

Where the receiver \( L \) is a (possibly subscripted) data item and \( R \) is either a data item or an expression.

Arithmetic Assignments: We will consider left-hand sides of matrices, vectors, integers, and scalars in turn.

Matrix

\[
\begin{align*}
\text{Matrix} &= \text{Matrix} \\
\text{Matrix} &= 0 \quad \text{(creates a null matrix)}
\end{align*}
\]

Note: Matrix = 6 is invalid!

Both left and right matrices must match in row and column dimensions. Precisions, however, need not match!
ASSIGNMENTS (CON'T.)

EXAMPLES:

M1 IS 3x3
\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9 \\
\end{bmatrix}
\]

M2 IS 2x2
\[
\begin{bmatrix}
20 & 40 \\
60 & 80 \\
\end{bmatrix}
\]

M3 IS 2x3
\[
\begin{bmatrix}
-1 & -2 & -3 \\
-4 & -5 & -6 \\
\end{bmatrix}
\]

M1 = 0 RESULTS IN
\[
\begin{bmatrix}
0 & 0 & 0 \\
0 & 0 & 0 \\
0 & 0 & 0 \\
\end{bmatrix}
\]

M1 = 6 ILLEGAL

M1 = MATRIX(2,2,2,3,3,3,4,4,4)
RESULTS IN
\[
\begin{bmatrix}
2 & 2 & 2 \\
3 & 3 & 3 \\
4 & 4 & 4 \\
\end{bmatrix}
\]

M2 = M3 ILLEGAL

M2 = -M3$(*, 2 AT 1)$
RESULTS IN
\[
\begin{bmatrix}
+1 & +2 \\
+4 & +5 \\
\end{bmatrix}
\]

M1$(2 AT 2, *) = M3$ RESULTS IN
\[
\begin{bmatrix}
1 & 2 & 3 \\
-1 & -2 & -3 \\
-4 & -5 & -6 \\
\end{bmatrix}
\]
ASSIGNMENTS (CON'T.)

VECTOR

VECTOR = VECTOR
VECTOR = 0 (CREATE A ZERO VECTOR)

NOTE: VECTOR = 6 IS INVALID!

BOTH LEFT AND RIGHT VECTORS MUST MATCH IN LENGTHS. PRECISIONS, HOWEVER, NEED NOT MATCH!

EXAMPLES:

\[
\begin{align*}
V_1 & = \begin{bmatrix} -5 \\ 10 \\ 15 \end{bmatrix} & V_2 & = \begin{bmatrix} 2 \\ 4 \\ 6 \end{bmatrix} & V_3 & = \begin{bmatrix} 100 \\ 200 \end{bmatrix} \\
V_1 & = 0 & RESULTS IN & V_1 = \begin{bmatrix} 0 \\ 0 \\ 0 \end{bmatrix} \\
V_1 & = 6 & ILLEGAL \\
V_1 & = V_2 & ILLEGAL \\
V_1 & = V_2(3 \text{ AT } 2) & RESULTS IN & \begin{bmatrix} 4 \\ 6 \\ 8 \end{bmatrix} \\
V_1 & = V_3 & ILLEGAL \\
V_1 & = V_3 & ILLEGAL \\
V_1 & = V_3 & RESULTS IN \begin{bmatrix} -5 \\ 100 \\ -200 \end{bmatrix}
\end{align*}
\]
ASSIGNMENTS (CON'T.)

INTEGER/SCALAR

THIS HAS ALREADY BEEN COVERED IN SOME DETAIL. WE WILL MERELY POINT
OUT THE FOLLOWING:

1. INTEGER = INTEGER
2. INTEGER = SCALAR
3. SCALAR = INTEGER
4. SCALAR = SCALAR

IN TYPES 2 AND 3 A CHANGE OF DATA WILL BE MADE. SCALARS ARE
CONVERTED TO INTEGERS BY ROUNDING.

EXAMPLES:

LET I BE AN INTEGER;
S BE A SCALAR,
M A 3x3 MATRIX.

THEN

I = 3
I = 11.2
-I = 11.9
S = 16.4
M*(3,*) = 5
M*(3,3) = 5

RESULTS IN I = 3
RESULTS IN I = 11
RESULTS IN I = 12
RESULTS IN S = 16.4
ILLEGAL
RESULTS IN

\[
\begin{bmatrix}
X & X & X \\
X & X & X \\
X & X & 5
\end{bmatrix}
\]
BOOLEAN ASSIGNMENTS

BOOLEAN = BOOLEAN EXPRESSION

EXAMPLES:

DECLARE BOOLEAN, B1, B2, B3, B4, B5;

B1 = TRUE;
B2 = OFF;
B3 = BIN'1';
B4 = BIN'0';
B5 = B1;
B5 = B1 & B2;
B5 = NOT B3;
B5 = B2 OR B3;
B5 = B1 AND B3;
B5 = NOT B1;

(SAME AS B1 = ON)
(SAME AS B2 = FALSE)
(SAME AS B3 = TRUE)
(SAME AS B4 = FALSE)
B5 = TRUE
B5 = FALSE
B5 = TRUE
B5 = TRUE
B5 = TRUE
B5 = TRUE
B5 = FALSE
MULTIPLE ASSIGNMENTS

MULTIPLE ASSIGNMENTS ARE RECOMMENDED BECAUSE THEY REDUCE THE NUMBER OF SEPARATE STATEMENTS OCCURRING IN A PROGRAM, AND THUS ENHANCE READABILITY.

FORM:

\[ L_1, L_2, L_3, \ldots - R \]

REQUISITIONS: EACH \( L_i = R \) MUST BE A LEGAL TYPE OF ASSIGNMENT.

NOTE: THE EXACT ORDER IN WHICH THE ASSIGNMENTS ARE MADE IS NOT EASILY PREDICTED.

EXAMPLES

LET \( M_1 \) BE A 2x2 MATRIX, \( M_2 \) A 3x3 MATRIX, \( S_1 \) AND \( S_2 \) SCALARS, AND \( I_1 \) AND \( I_2 \) INTEGERS.

\[ \text{MULTIPLE ASSIGNMENT} \]

\[ 1 \text{ IF } S_1 = 6 \text{ THEN } M_1, M_2, S_1, S_2, I_1, I_2 = 0; \]
ASSIGNMENTS (CON'T.)

CONTRAST THIS WITH:

IF $S_1 = 6$ THEN DO,

\begin{align*}
M_1 &= 0, \\
M_2 &= 0, \\
S_1 &= 0, \\
S_2 &= 0, \\
I_1 &= 0, \\
I_2 &= 0,
\end{align*}

END;

1. $M_1, S_1 = 6$ IS **ILLEGAL** BECAUSE $M_1 = 6$ IS ILLEGAL.

2. $M_1, S_1 = 0$ IS OK.

3. POTENTIAL HAZARD:

\[ M_1(I_1, I_2), I_1 = I_2; \]

Assume $I_1 = 2, I_2 = 3$ BEFORE ASSIGNMENT, COMPILER IS FREE TO PICK BEST ORDER ON A MULTIPLE ASSIGNMENT. EITHER

\[ M_1(2, 3) \text{ OR } M(3, 3) \]

WILL BE ASSIGNED IN THIS CASE.
CONDITIONALS

SIMPLE IF STATEMENT

IF exp THEN \[stmt\] ;

WHERE exp IS EITHER A BOOLEAN EXPRESSION (i.e., evaluated as true or false) OR A RELATIONAL EXPRESSION (i.e., \(a = 0\)). The key point is that exp must be something that can be evaluated as true or false.

NOTES:
1. \[stmt\] is executed only if exp evaluates to true.
2. \[stmt\] can have a statement label.
3. If exp is false, \[stmt\] is bypassed.
CONDITIONALS (CON'T.)

EXAMPLES (SIMPLE IF)

ASSUME B1, B2, B3 ARE BOOLEANS.
S1, S2, S3 ARE SCALARS.

1. IF B1 \& B2 THEN
   \{ S1, S2 = 0 \}
   \} NOTE THAT THE "IF" PART AND "TRUE" PART
   \} ARE 2 STMTS.

2. IF \neg B3 THEN
   B1 = B2;

3. IF B1 \& B2 THEN
   IF S1 = 6 THEN
     IF \neg B3 \| B2 THEN
       S1 = S2;
   \} NOTE: IF STATEMENTS CAN BE NESTED.

4. IF S1 < 4 \| S2 > 6 THEN
   DO;
   S1 = \sin(S2)**2;
   S2 = S2 + S3;
   END;

ENTIRE STATEMENT
GROUP IS TRUE PART
AUGMENTED IF STATEMENT

This is like a simple IF except that it has an ELSE clause that is executed only if the "exp" is false.

An augmented IF can be placed within nested simple IFs, but an augmented IF can never nest inside another (without a statement group acting as an insulator).

EXAMPLES (AUGMENTED IF)

Assume B1, B2, B3 are booleans,
S1, S2, S3 are scalars,
I1, I2, I3 are integers.

1) IF B1 THEN
   S1 = S2**2 + S3**2;
ELSE S1 = SIN(S2 S3);
CONDITIONALS (CON'T.)

(2) IF \( S1 < S2 \) THEN
  \[ \text{ZERO IT: } S1 = 0; \]
  \[ \text{ELSE } S1 = S2^*2; \]
  \[\{ \text{TRUE PART CAN HAVE A LABEL} \}\]

(3) IF \( B1 \) THEN
  \[ B2 = \text{FALSE}; \]
  \[ \text{ELSE TROUBLE: } B2 = \text{TRUE}; \]
  \[\{ \text{FALSE PART CAN HAVE A LABEL, ALSO} \}\]

(4) IF \( B1 \& B2 \& B3 \) THEN
  IF \( S1 < 5 \& S1 > 9 \) THEN
    IF \( S1 = 0 \& S2 = 4 \) THEN
      \( I1 = 6; \)
    ELSE \( I1 = 10; \)
  \[\{ \text{SIMPLE IFs} \}\]
  \[\{ \text{AUGMENTED IF} \}\]

\text{- NOTE: DANGLING ELSEs ARE NOT A PROBLEM BECAUSE ELSE GOES WITH THE INNERMOST IF.}
CONDITIONALS (CON’T.)

(5) IF B1 | B2 THEN
   B3 = FALSE;
ELSE IF B2 & B3 THEN
   B3 = TRUE;
ELSE IF S1 = 0 THEN
   S1 = 1;

(6) IF S1 = 4 THEN
   S2 = 1;
ELSE IF S1 = 6 THEN
   S2 = 2;
ELSE IF S1 = 8 THEN
   S2 = 3;

——— CONTROL JUMPS HERE AFTER COMPLETION OF ANY TRUE PART.

ANY LABELS PUT ON TRUE PART OR FALSE PART (ELSE) MAY NOT BE BRANCHED
TO FROM ANYWHERE OUTSIDE OF THE IF STATEMENT.
CONDITIONALS (CON'T.)

RELATIONAL EXPRESSIONS

RELATIONAL EXPRESSIONS CAN BE USED IN CONDITIONAL STATEMENTS (E.G., IFs) AS CAN BOOLEAN EXPRESSIONS. A RELATIONAL EXPRESSION IS SOME SORT OF A COMPARISON THAT WILL EVALUATE TO A TRUE OR A FALSE CONCLUSION.

RELATIONALS ARE EXPRESSIONS LINKED BY COMPARATIVE OPERATORS: >, <, <=, >=, =, /=. RELATIONALS, IN TURN, MAY BE COMBINED USING THE 3 BOOLEAN OPERATORS: &&, |, AND . RELATIONALS, HOWEVER, ARE NOT QUITE BOOLEANS.

COMPARATIVE OPERATORS ARE LOOSELY GROUPED INTO 2 CLASSES:

CLASS I: >, <, <=, >=, <, >

CLASS II: =, !=
CONDITIONALS (CON'T.)

CLASS II OPERATORS (=, \neq) ARE THE ONLY ONES THAT CAN BE USED WITH: VECTOR, MATRIX, BOOLEAN, AND BIT STRING DATA TYPES.

- VECTOR AND MATRIX COMPARSES ARE PERFORMED ELEMENT-BY-ELEMENT.
- VECTORS AND MATRICES MUST HAVE THE SAME "SHAPES", I.E., VECTORS MUST BE OF IDENTICAL LENGTHS, AND MATRICES MUST HAVE THE SAME NUMBER OF ROWS AND COLUMNS.

EXAMPLES:

\[ \begin{align*}
\text{LET } V_1 &= \begin{bmatrix} 1 \\ 4 \\ 6 \end{bmatrix},
V_2 &= \begin{bmatrix} 4 \\ 6 \end{bmatrix},
M_1 &= \begin{bmatrix} 1 & 2 \\ 3 & 5 \end{bmatrix},
M_2 &= \begin{bmatrix} 1 & 4 & 6 \\ 2 & 1 & 3 \end{bmatrix} \\
(1) & \ V_1 = V_2 \quad \text{ILLEGAL (VECTORS \# IN LENGTH)} \\
(2) & \ M_1 = M_2 \quad \text{ILLEGAL (MATRICES HAVE DIFFERENT COLUMN CT)} \\
(3) & \ V_1 = M_2$(1,*) \quad \text{TRUE} \\
(4) & \ V_1$(2 AT 2) = V_2 \quad \text{TRUE} \\
(5) & \ V_2 = M_1$(*, 1) \quad \text{FALSE}
\end{align*} \]
CONDITIONALS (CON'T.)

FOR INTEGERS, SCALARS, AND CHARACTER STRINGS, BOTH CLASS I AND
CLASS II CONDITIONAL OPERATORS CAN BE USED.

- AS WE WOULD EXPECT, IN AN INTEGER/SCALAR COMPARISON, THE
  INTEGERS ARE FIRST CONVERTED TO SCALARS.

- CHARACTER STRING COMPARISONS ALLOW US TO SORT STRINGS ALPHABETICALLY.

SUPPOSE

\[
\begin{align*}
I_1 &= 5 & I_2 &= -3 & I_3 &= 0 \\
S_1 &= -80 & S_2 &= 6.5 & S_3 &= 10.3 \\
C_1 &= 'Z' & C_2 &= 'ABC' & C_3 &= 'B'
\end{align*}
\]

THEN

\[
\begin{align*}
I_1 > I_2 &= \text{TRUE} \\
I_1 < I_3 &= \text{FALSE} \\
I_1 = 5 &= \text{TRUE} \\
C_1 < C_2 &= \text{FALSE} \\
C_2 > C_3 &= \text{TRUE} \\
I_1 < S_2 &= \text{TRUE} \\
I_3 < C_1 &= \text{ILLEGAL} \\
(I_3 - I_2) < I_1 &= \text{TRUE} \\
I_1 * 2 < -S_1 &= \text{TRUE} \\
C_2 < C_3 &= \text{TRUE}
\end{align*}
\]
CONDITIONALS (CON'T.)

- COMPARATIVE OPERATIONS CAN BE COMBINED (AS IF THEY WERE BOOLEAN OPERANDS), USING THE BOOLEAN OPERATORS \&; |; AND-\.

STILL, COMPARATIVE OPERATIONS ARE NOT BOOLEANS AND THEY CANNOT BE MIXED WITH BOOLEANS IN A CONDITIONAL STATEMENT.

EXAMPLES

ASSUME: \( B_1, B_2, B_3 \) BOOLEANS,
\( I_1, I_2, I_3 \) INTEGERS,
\( S_1, S_2, S_3 \) SCALARS,
\( V_1, V_2, V_3 \) VECTORS.

LEGAL

IF \( B_1 \neg B_2 \neg B_3 \) THEN ...
IF \( V_1 \neq V_2 S_1 \leq S_2 \) THEN ...
IF \( I_1 + I_2 \leq 0 S_1 \leq S_2 \times 2 \) THEN ...
IF \( V_1 - V_2 = V_3 \& S_1 \leq S_2 + 2 \) THEN ...

ILLEGAL

IF \( B_1 \neg B_2 S_1 \leq S_2 \) THEN ...
IF \( \neg B_2 S_1 = S_2 \) THEN ...
IF \( V_1 = V_2 \& B_3 \) THEN ...

4 - 9
CONDITIONALS (CON'T.)

BUT WE CAN GET OUT OF TROUBLE BY TURNING THE BOOLEAN EXPRESSIONS INTO RELATIONAL EXPRESSIONS:

IF \((B_1 \lor B_2) = \text{TRUE} \lor S_1 < S_2\) THEN ...
IF \(B_2 = \text{FALSE} \lor S_1 = S_2\) THEN ...
IF \(V_1 = V_2 \& B_3 = \text{TRUE}\) THEN ...

PRECEDENCE

1. ARITHMETIC OPERATIONS (E.G., /, *, , +, -, **, MULT., (BLANK)).
2. COMPARATIVE OPERATIONS (E.G., <, >, =, \!, \!, \!, \!, \!, \!, \!, \!, \!),
3. \(! (\neg)\)
4. AND (\&)
5. OR (|)
EXAMPLES (PRECEDENCE IN RELATIONAL EXPRESSION)

IF S1 + S2 = 0 | \neg (S3 > 0) \land \neg (S4 < 0 | S5 > 0) ... 

1 \rightarrow 2 \rightarrow 10 \rightarrow 4 \rightarrow 3 \rightarrow 9 \rightarrow 8 \rightarrow 5 \rightarrow 7 \rightarrow 6

ARROWS INDICATE ORDER OF EXECUTION OF OPERATIONS

IF V = \neg W \land S1 = S2 \land (B1 \land B2) = TRUE ... 

2 \rightarrow 1 \rightarrow 5 \rightarrow 4 \rightarrow 3 \rightarrow 8 \rightarrow 6 \rightarrow 7
LABELS AND BRANCHES

LABELS ARE NAMES CHOSEN BY THE PROGRAMMER AND ATTACHED TO EXECUTABLE STATEMENTS. WE CAN DISTINGUISH BETWEEN BLOCK LABELS (MANDATORY LABELS ON BLOCKS).

ALPHA: PROGRAM;
BETA: COMPOOL;
GAMMA: PROCEDURE;
DELTA: FUNCTION;
EPSILON: TASK;
ZETA: UPDATE;

AND STATEMENT LABELS. ONLY THE LATTER MAY BE BRANCHED TO.

- A STATEMENT MAY HAVE 0, 1, OR MORE LABELS -- THERE IS NO CORE OR CPU PENALTY FOR SUCH LABELS AND THEY MAY BE USEFUL FOR DIAGNOSTIC PURPOSES (AS WILL BE SEEN).
CONDITIONALS (CON'T.)

EXAMPLES

(1) SET_X: \( X = \sin(Y) \);

(2) SET_X: BOMB_OUT: \( X = \sqrt{-1} \);

(3) BIT_LOOP: DO FOR I = 1 TO 10000;

(4) DEPART: CLOSE ALPHA;

(5) IF A = 3 THEN
   CALL_PROC: CALL UPDATER\(A\);
   ELSE DONE: GO TO END_ALL;

(6) CHECK_FAIL: IF Z = 0 THEN
   CALL FLASH_DISPLAY\(6\);

GO TO STATEMENT

GENERIC FORM:

   \text{GO TO } \textit{label};

THIS DOES THE OBVIOUS -- CONTROL IS TRANSFERRED TO THE STATEMENT WITH LABEL \textit{label} (PROVIDED STRINGENT CONDITIONS ARE MET).
CONDITIONALS (CON'T.)

ALTHOUGH STRUCTURED PROGRAMMING PRACTICE DISCOURAGES USE OF "GO TO" STATEMENTS, THEY ARE SOMETIMES PREFERABLE TO A DAY OF WORK TO ELIMINATE ONE. THE FOLLOWING EXAMPLE, HOWEVER, SHOWS HOW EASY HAL/S MAKES THE ELIMINATION OF GO TO'S.

EXAMPLE

IF VALUE < LOW_LIMIT THEN GO TO LOW;
IF VALUE > HI_LIMIT THEN GO TO HI;
IN_LIMITS: GOOD_FLAG = TRUE;
OUT_VAL = VALUE;
GO TO FINISH;
LOW: GOOD_FLAG = FALSE;
OUT_VAL = LOW_LIMIT;
GO TO FINISH;
HI: GOOD_FLAG = FALSE;
OUT_VAL = HI_LIMIT;
FINISH:
CONDITIONALS (CON'T.)

BECOMES

IF VALUE < LOW_LIMIT THEN
  DO;
    GOOD_FLAG = FALSE;
    OUT_VAL = LOW_LIMIT;
  END;
ELSE IF VALUE > HI_LIMIT THEN
  DO;
    GOOD_FLAG = FALSE;
    OUT_VAL = HI_LIMIT;
  END;
ELSE DO;
  GOOD_FLAG = TRUE;
  OUT_VAL = VALUE;
END;
STATEMENT GROUPS

A STATEMENT GROUP is a set of HAL/S statements that are considered as a unit for the purposes of conditional or repetitive execution. Statement groups begin with a "DO" statement, close with an "END" statement, and can contain other statement groups (or even code blocks) nested within. The "DO" statement is considered to be an executable statement (although code is not always generated for it) and as such can possess a statement label. The "DO" statement has the form:

```
DO (control)
```

(Control) is optional and will be described next. If (Control) is omitted the statement group DO ... END is executed once.
STATEMENT GROUPS (CON'T.)

THE "END" STATEMENT IS ALSO CONSIDERED TO BE AN EXECUTABLE STATEMENT (AGAIN, IT MIGHT NOT ALWAYS RESULT IN CODE) AND MAY BE LABELLED. THE GENERIC FORM OF THE "END" STATEMENT IS:

```
END label
```

`label` IS OPTIONAL, BUT IF IT IS PRESENT IT MUST MATCH THE LABEL ON THE CORRESPONDING "DO" STATEMENT. THUS,

```
OPTIONAL — — — — — — MUST MATCH IF PRESENT

CYCLE: DO FOR I = 1 TO 5;

END_CYCLE: END CYCLE;
```

USE OF THESE OPTIONAL LABELS IS RECOMMENDED FOR THE FOLLOWING REASONS:
(1) ALLOWS CROSS-CHECKING BY THE COMPILER THAT ENSURES EVERY "DO" IS PROPERLY CLOSED.
(2) A LABEL PROVIDES A FIXED REFERENCE POINT FOR DIAGNOSTICS.
(3) MEANINGFUL LABELS INCREASE PROGRAM READABILITY.
STATEMENT GROUPS (CON'T.)

DO WHILE

DO WHILE (condition);

....

END;

• THE STATEMENT GROUP IS REPETITIVELY EXECUTED AS LONG AS
  THE (condition) REMAINS TRUE.

• IF (condition) IS FALSE, THEN THE STATEMENT GROUP IS NOT EXECUTED
  AT ALL.

• (condition) IS A RELATIONAL OR A BOOLEAN EXPRESSION (NOT A
  MIXTURE OF COURSE) AND IS EVALUATED PRIOR TO EACH CYCLIC
  EXECUTION OF THE STATEMENT GROUP.

• WHEN (condition) BECOMES FALSE, CYCLIC EXECUTION HALTS AND
  CONTROL IS PASSED TO THE STATEMENT FOLLOWING THE "END" STATEMENT.

EXAMPLE

I = 50;
DO WHILE I > 0;
  TABLE$1 = I**2;
  I = I - 2;
END;
DO UNTIL

DO UNTIL (condition) ;

. . .

END;

• THE STATEMENT GROUP IS EXECUTED CYCLICALLY UNTIL (condition)

• THE TEST OF (condition) IS MADE AT THE END OF THE CYCLES
  SO THE STATEMENT GROUP WILL ALWAYS BE EXECUTED AT LEAST
  ONCE.

• AS WAS THE CASE WITH DO...WHILE, (condition) IS A RELATIONAL OR A
  BOOLEAN EXPRESSION.

LOOPS THAT WILL CYCLE FOREVER:

(A)· DO WHILE (TRUE) ;

. . .

END;

(B)· DO UNTIL (FALSE) ;

. . .

END;
STATEMENT GROUPS (CON'T.)

DO FOR

THERE ARE 2 BASIC FORMS: ITERATIVE AND DISCRETE:

ITERATIVE

DO FOR (loop var) = (initial) TO (final) BY (increment);

- (loop var) IS AN UNARRAYED INTEGER OR SCALAR. IT IS THE
CONTROL VARIABLE (THE VARIABLE COULD BELONG TO AN ARRAY,
BUT IF SO ITS ARRAYNESS MUST BE SUBSCRIPTED AWAY).

- (initial), (final), AND (increment) ARE INTEGER OR SCALAR
EXPRESSIONS.

- ON THE FIRST CYCLE, (loop var) WILL HAVE THE VALUE (initial).

- ON EACH SUCCEEDING CYCLE, (loop var) WILL BE AUGMENTED BY
(increment).

- CYCLIC EXECUTION CONTINUES UNTIL (loop var) LIES OUTSIDE THE
RANGE BOUNDED BY (initial) AND (final).
STATEMENT GROUPS (CON'T.)

NOTE: ① INCREMENT IS ASSUMED TO BE +1 IF NOT SPECIFIED, I.E., IN THIS CASE THE "BY (increment)" IS OPTIONAL.
② (initial), (final), AND (increment) ARE EVALUATED ONCE (AND THEIR VALUES SAVED) PRIOR TO INITIATION OF THE LOOP, I.E., VALUES CANNOT BE CHANGED WITHIN THE LOOP.

EXAMPLES

① DO FOR I = 1 TO 20;
   ...
   END;
EQUIVALENT
   DO FOR I = 1 TO 20 BY 1;
   ...
   END;

② DO FOR I = 2J TO (J**2-1) BY 5;
   ...
   J = J - 1;
   CANNOT AFFECT LOOP TERMINATION.
   ...
   END;


STATEMENT GROUPS (CONT.)

DISCRETE

DO FOR \((\text{loop var}) = \exp_1, \exp_2, \ldots, \exp_n\)

- \((\text{loop var})\) IS AN UNARRAYED INTEGER OR SCALAR -- THE CONTROL VARIABLE.
- THE LOOP WILL BE EXECUTED \(n\) TIMES, WITH \((\text{loop var}) = \exp_1\)
  ON THE \(i^{th}\) EXECUTION.

**NOTE:** EACH EXPRESSION \(\exp_1\) IS EVALUATED JUST PRIOR TO USING IT, I.E., ON THE \(i^{th}\) EXECUTION.

EXAMPLES

1) DO FOR \(I = 1, 3, 5, 9, \ldots\)
   END;

2) DO FOR \(S = \sin(30), \sin(45), \sin(60), \ldots\)
   END;

3) \(J = 1\)
   DO FOR \(I = J, J+2, J+4, J-6, \ldots\)
   \(J = J(J+1)\),
   \(\ldots\)
   END;

\(I = 1\)  1\(^{st}\) CYCLE
4  2\(^{nd}\) CYCLE
10  3\(^{rd}\) CYCLE
36  4\(^{th}\) CYCLE
STATEMENT GROUPS (CON'T.)

Both the iterative and discrete "DO FOR" statements may have an additional "WHILE" or "UNTIL" clause acting as a further execution qualifier:

DO FOR I = 1 TO J WHILE I < 50;

. . .

END;

DO FOR I = 1 TO 10000 UNTIL J = 0;

. . .

IF A > 0 THEN J = J - 1;

. . .

END;

DO FOR I = 1, 3, 7, 9 WHILE B;

. . .

IF SIN(Q) < .4 THEN B = FALSE;

. . .

END;
STATEMENT GROUPS (CON'T.)

DO CASE -- USED TO SELECT ONE STATEMENT TO BE EXECUTED BASED ON A CALCULATION (ROUGHLY EQUIVALENT TO A FORTRAN COMPUTED GOTO).

FORM:

DO CASE (exp);

- \( (\text{exp}) \) IS AN INTEGER OR SCALAR EXPRESSION (SCALAR WILL BE CONVERTED TO AN INTEGER BY ROUNDING).
- IF \( (\text{exp}) \) EVALUATES TO 1, 2, 3, ..., \( K \) THEN THE 1\(^{\text{ST}}\), 2\(^{\text{ND}}\), 3\(^{\text{RD}}\), ..., \( K^{\text{TH}} \) STATEMENT (OR STATEMENT GROUP) FOLLOWING THE "DO CASE" WILL BE EXECUTED.
- IN THIS SIMPLE FORM OF THE "DO CASE" NO ERROR CHECKING IS PERFORMED, I.E., UNPREDICTABLE AND DISASTROUS BEHAVIOR WILL RESULT IF \( (\text{exp}) \) EVALUATES TO \( < 0 \) OR \( > K \).
A DO CASE is powerful because a specific case can range from a simple statement to a complex "DO...END" group containing an arbitrary number of statements.

**Example**

```plaintext
DO CASE j;
  J = 1;
  DO;
    J = 2;
    K = J**2 - 4;
  END;
  J = 3;
  DO;
    CALL ALPHA;
    END;
  ALPHA: PROCEDURE;
  CLOSE ALPHA;
END;
```

```
CASE 1
CASE 2
```

```
DO FOR K = 1, 3, 5;
  END;
J = 5;
DO;
  END;
  END;
```

```
CASE 4 (ANY DO,...END GROUP CAN HAVE NESTED BLOCKS)
```

```
CASE 5
CASE 6
```

END OF DO CASE;
STATEMENT GROUPS (CON'T.)

ERROR CHECKING ON THE CASE VARIABLE IS PERFORMED IF AN "ELSE" CLAUSE IS ADDED TO THE "DO CASE".

\[
\text{DO CASE (exp); ELSE } \text{stmt} ; \\
\text{ (case 1)} \\
\text{ (case 2)} \\
\ldots \\
\text{ (case } K) \\
\text{END};
\]

IF (exp) EVALUATES TO \(< O \text{ OR } > K\) THEN \text{stmt} IS EXECUTED (WHICH COULD OF COURSE SIGNAL AN ERROR CONDITION).

- AN ELSE CLAUSE SHOULD ALWAYS BE PROGRAMMED IF THERE IS A POSSIBILITY THAT THE CASE VARIABLE COULD EVER MISBEHAVE!

NOTE: A DO CASE MAY HAVE A MAXIMUM OF 256 CASES.
STATEMENT GROUPS (CON'T.)

BRANCHING (GO TO, EXIT, REPEAT)

A STATEMENT GROUP, I.E. DO...END, DO WHILE, DO UNTIL, DO FOR, DO CASE, MAY BE BRANCHED OUT OF VIA A GO TO, BUT SUCH A GROUP MAY NOT BE BRANCHED INTO. BRANCHING WITHIN, OF COURSE, IS LEGAL.

LEGAL

1. \[
\begin{align*}
&\text{DO,} \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{I = 1}_j \\
&\text{\ldots} \\
&\text{END}_j \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{K = 1}_j \\
&\text{\ldots} \\
&\text{END}_j \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{\ldots} \\
&\text{END}_j \\
\end{align*}
\]

WITHIN

2. \[
\begin{align*}
&\text{DO,} \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{END}_j \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{K = 1}_j \\
&\text{\ldots} \\
&\text{END}_j \\
\end{align*}
\]

TO THE END

3. \[
\begin{align*}
&\text{DO,} \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{\ldots} \\
&\text{GO TO X}_j \\
&\text{\ldots} \\
&\text{X}_i \\
&\text{\ldots} \\
&\text{END}_j \\
\end{align*}
\]

TO THE OUTSIDE
STATEMENT GROUPS (CON'T.)

LEGAL

4) DO;
   DO;
   DO;
   GO TO X;
   END;
X: I = I;
   END;

OK BECAUSE ALREADY
IN THE OUTER DO!

ILLEGAL

1) GO TO X;
   DO;
   DO;
   GO TO X;
   END;
X: I = I;
   END;

CANNOT BRANCH INTO

2) DO;
   GO TO X;
   DO;
   GO TO X;
   END;
X: I = I;
   END;

CANNOT BRANCH INTO
(EVEN IF NESTED)
STATEMENT GROUPS (CON'T.)

SINCE BRANCHING WITHIN (OR OUT OF) "DO" GROUPS IS INVARIBLY
NECESSARY, AND SINCE "GO TOs" ARE UNDESIRABLE IN STRUCTURED
PROGRAMMING, HAL/S OFFERS TWO ALTERNATE CONSTRUCTIONS:

EXIT & REPEAT

EXIT

FORM 1: EXIT;
• CAUSES IMMEDIATE BRANCH OUT OF THE INNERMOST
STATEMENT GROUP IN WHICH IT IS ENCLOSED.
• EXECUTION IS DIRECTED TO THE FIRST STATEMENT
FOLLOWING THE "END" OF THE GROUP BRANCHED OUT OF.

FORM 2: EXIT label;
• CAUSES BRANCH OUT OF THE ENCLOSING STATEMENT
GROUP THAT POSSESS THE LABEL label ON ITS
"DO" STATEMENT.
• EXECUTION IS DIRECTED TO THE FIRST STATEMENT
FOLLOWING THE "END" STATEMENT OF THAT GROUP.
STATEMENT GROUPS (CON'T.)

OUTER:  DO WHILE J < 0;
  J = J - 2;
  Q = SIN(X)**J;

MIDDLE:  DO FOR I = 1 TO 10;
  A$I = A$I - 5;
  DO FOR J = 1 TO 20;
    B$J = B$J - 1;
    IF I = 5 THEN EXIT;
    ELSE IF Q = 0 THEN
      EXIT OUTER;
  END;
END;

END;

THIS SHOWS THE ADVANTAGES IN LABELING DO...END GROUPS. NOTE THAT THE EXIT STATEMENT ACTS LIKE A GO TO, BUT IS A MORE SECURE (AND STRUCTURED) CONSTRUCTION.
STATEMENT GROUPS (CON'T.)

REPEAT (MUST BE ENCLOSED WITHIN DO FOR, DO WHILE, OR DO UNTIL GROUP).

FORM 1: REPEAT;

- MUST BE ENCLOSED WITHIN A DO FOR, DO WHILE, OR DO UNTIL GROUP (I.E., A REPETITIVE GROUP).

- CAUSES IMMEDIATE BRANCH TO THE BEGINNING OF THE INNERMOST REPETITIVE GROUP.

FORM 2: REPEAT label;

- MUST BE ENCLOSED WITHIN A REPETITIVE GROUP THAT HAS LABEL label ON ITS "DO" STATEMENT.

- CAUSES IMMEDIATE BRANCH TO THE BEGINNING OF SAID REPETITIVE GROUP.
STATEMENT GROUPS (CON'T.)

EXAMPLES

1. REPEAT
   DO WHILE I < 10
      . . .
   IF A = 0 THEN
      DO;
      IF K = 1 THEN REPEAT;
      . . .
      END;
   END;
   . . .
   END;
   DO FOR I = 1 TO 100;
      . . .
      IF J > 50 THEN REPEAT;
      . . .
      END;
   END;

NON-REPETITIVE GROUP
STATEMENT GROUP (CON'T.)

\(\text{REPEAT label}\)

ONE: DO FOR I = 1 TO 10;
    J = I;
    DO WHILE J > 0;
        J = J - 1;
        X$J = X$J + J;
        IF X$J = 25 THEN REPEAT;
        IF X$J = 0 THEN REPEAT ONE;
    END;
END;
Z = 0;
PROCEDURES AND FUNCTIONS

PROCEDURES AND FUNCTIONS ARE THE TWO MOST COMMON CODE BLOCKS. IF PROCEDURES/FUNCTIONS ARE COMPILED SEPARATELY THEY ARE CALLED COMSUBs (COMMON SUBROUTINES). OTHERWISE, THEY ARE NESTED IN A PROGRAM.

- THE MAXIMUM NESTING DEPTH OF CODE BLOCKS IS 16.
- THE MAXIMUM NUMBER OF BLOCKS (CODE AND DATA) IN A COMPILATION IS 256 (INCLUDING EXTERNAL TEMPLATES).
- A PROGRAM BLOCK CANNOT NEST WITHIN A PROGRAM BLOCK.

THIS IS NOT NESTING

P1: EXTERNAL PROGRAM;
CLOSE P1;
D VERSION 6
P2: PROGRAM
DECLARE NN NAME PROGRAM
    INITIAL(NAME(P1));
CLOSE P2;

THIS IS

P1: PROGRAM;

P2: PROGRAM
DECLARE NN NAME PROGRAM
    INITIAL(NAME(P1));
CLOSE P2;
CLOSE P1;
THE MAXIMUM NEST DEPTH HERE IS 4.

PARAMETERS

PROGRAM BLOCKS CANNOT HAVE PARAMETERS -- ONLY PROCEDURE AND FUNCTION BLOCKS MAY. (THE OTHER BLOCKS NOT COVERED SO FAR -- UPDATE BLOCKS AND TASK BLOCKS -- ARE ALSO PARAMETERLESS.)

PARAMETERS ARE EITHER INPUT OR ASSIGN PARAMETERS. OF THESE, ONLY A PROCEDURE CAN HAVE AN ASSIGN PARAMETER.
PROCEDURES AND FUNCTIONS (CON'T.)

- INPUT PARAMETERS ARE USED TO PASS DATA INTO A PROCEDURE OR FUNCTION (THEIR VALUES CANNOT BE CHANGED BY THE PROCEDURE/FUNCTION).

- ASSIGN PARAMETERS CAN BE USED BOTH TO PASS DATA IN AND OUT OF A PROCEDURE BLOCK. A FUNCTION DOES NOT HAVE ASSIGN PARAMETERS; FUNCTION RESULTS ARE PASSED OUT VIA A RETURN STATEMENT.

AGAIN, IT IS RECOMMENDED THAT ALL BLOCKS (BUT ESPECIALLY FUNCTIONS AND TASKS) BE DEFINED PRIOR TO THEIR POINTS OF INVOCATION.

PROCEDURE BLOCK

(labeled)

PROCEDURE(\(l_1, l_2, \ldots l_n\))

ASSIGN(\(a_1, a_2, \ldots a_m\))

MANDATORY IDENTIFIER

(OPTIONAL) ASSIGN PARAMETERS

(OPTIONAL) INPUT PARAMETERS
FUNCTION BLOCK

\textit{label}: FUNCTION(\textit{i}_1, \textit{i}_2, \ldots \textit{i}_n) \textit{attributes};

MANDATORY (OPTIONAL) INPUT PARAMETERS

THE TYPE AND PRECISION OF THE
DATA RETURNED BY THE FUNCTION


PROCEDURE AND FUNCTION BLOCKS MUST HAVE A \textit{CLOSE} STATEMENT AS THE FINAL STATEMENT. THE \textit{CLOSE} STATEMENT IS AN EXECUTABLE STATEMENT OF THE FORM:

\begin{verbatim}
   CLOSE;
\end{verbatim}

OR

\begin{verbatim}
   CLOSE \textit{label};
\end{verbatim}

IF (\textit{label}) IS PRESENT, IT MUST MATCH THE LABEL ON THE PROCEDURE/FUNCTION BLOCK HEADER.
PROCEDURES AND FUNCTIONS (CON’T.)

A procedure is exited either by falling into (or branching to) the close statement, or by use of a return statement. A function must exit via a return (exp) statement, i.e., not only a return, but one that returns a value of the proper data type and precision (implicit conversions allowed).

- Parameter declarations should always precede local data declarations. In the case of COMSUBS this is required.

Input parameters - passed by “value” in the case of integers and scalars. All other data passed by “name” (i.e., reference).

Assign parameters - always passed by “name”.

The concept of a stack, which will be discussed in detail later, is a key part of the machinery for parameter passage.
PROCEDURES AND FUNCTIONS (CON'T.)

EXAMPLE 1

INPUT

ASSIGN

PARMS

PARMS

ALPHA: PROCEDURE(I, S1, V) ASSIGN(S2, W);

DECLARE INTEGER, I;
DECLARE SCALAR, S1, S2 DOUBLE;
DECLARE V VECTOR, W MATRIX;
DECLARE VECTOR, X, Y;
DECLARE SCALAR, T, U;
DECLARE MATRIX, A, B, C;

DECLARE SCALAR, T, U;

DECLARE VECTOR, X, Y;

DECLARATIONS

DECLARATIONS

PARAMETER

DECLARATIONS

LOCAL DATA

DECLARATIONS

AN_ASSIGN PARM CAN BE USED AS
INPUT AS WELL AS OUTPUT.

IF S2 = 0 THEN
  DO;
    S2 = 0;
    \(\hat{W} = S1 \hat{V} + i^2 \hat{X} \hat{Y}\);
  END;
ELSE
  DO;
    S2 = 1;
    \(\hat{W} = S1 \hat{V} \hat{X} + i^2 \hat{V} \hat{Y}\);
  END;
IF I = 0 THEN
  \(\hat{W} = \hat{W} + \hat{A} \hat{B} + \hat{I} \hat{C}\);
CLOSE ALPHA; —— CAUSES A RETURN TO THE CALLER.
PROCEDURES AND FUNCTIONS (CON'T.)

EXAMPLE 2

INPUT PARM

FUNCTION WILL RETURN A 3x3 DOUBLE PRECISION MATRIX

BETA: FUNCTION(M) MATRIX DOUBLE;

DECLARE M MATRIX DOUBLE;

DECLARE M1 MATRIX DOUBLE
INITIAL(3#1, 3#2, 3#3);

DECLARE M2 MATRIX DOUBLE
INITIAL(3#2, 3#1, 3#3);

RETURN M + M1 + M2;

CLOSE BETA;

Will never be executed because of return stmt.

NOTE: A run-time error will occur if the close statement is ever reached in a function.

Procedures are entered via a call statement that may or may not have input or assign arguments -- depending on whether the procedure has input or assign parameters.

ASSIGN ARGUMENTS

CALL ALPHA(J, X, V1) ASSIGN(T, H);

INPUT ARGUMENTS
FUNCTIONS ARE ENTERED BY INVOCATION, I.E., BY USING THE NAME OF THE FUNCTION AS THOUGH IT WERE A DATA TYPE. IF THE FUNCTION HAS INPUT PARAMETERS, THEN THE INVOCATION MUST SPECIFY CORRESPONDING INPUT ARGUMENTS.

FUNCTION BETA TAKES MATRIX M2 AS AN INPUT ARGUMENT AND RETURNS A NEW MATRIX THAT IS THEN ASSIGNED TO M3.

IF THE FUNCTION HAS NO PARAMETERS, THEN THE PARENTHEZIZED LIST IS OMITTED.

MORE FREEDOM (E.G., MISMATCH OF DATA TYPES AND/OR PRECISION) IS ALLOWED FOR INPUT ARGUMENTS THAN IS ALLOWED FOR ASSIGN ARGUMENTS.
PROCEDURES AND FUNCTIONS (CON'T.)

INPUT PARAMETERS

REMEMBER THAT PARAMETERS ARE DECLARED DATA ITEMS, BUT ARGUMENTS (INPUT, AT LEAST) CAN BE EXPRESSIONS:

\[ \text{ALPHA: PROCEDURE}(I, S); \]
\[ \text{DECLARE I INTEGER, S SCALAR;} \]
\[ \text{...} \]
\[ \text{CLOSE ALPHA;} \]
\[ \text{CALL ALPHA}(K^3 + 3 + \sin(7), \log(\cosh(4)) + 3.14159); \]

VALUE WILL BE COMPUTED AND PHYSICALLY ASSIGNED TO I.

VALUE WILL BE COMPUTED AND PHYSICALLY ASSIGNED TO S.

IF INPUT ARGUMENTS ARE EXPRESSIONS THAT RESULT IN NON-INTEGER OR SCALAR RESULTS, HAL/S PUTS THE EVALUATED EXPRESSION IN A TEMPORARY LOCATION AND PASSES THE ADDRESS OF THIS TEMPORARY LOCATION TO THE PROCEDURE.
PROCEDURES AND FUNCTIONS (CON'T.)

IMPLICIT CONVERSIONS (INPUT PARAMETERS)

[NO IMPLICIT CONVERSIONS LEGAL FOR ASSIGN PARAMETERS]

<table>
<thead>
<tr>
<th>INPUT PARM</th>
<th>INPUT ARG</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALAR SINGLE</td>
<td>SCALAR SINGLE</td>
</tr>
<tr>
<td>SCALAR DOUBLE</td>
<td>SCALAR DOUBLE</td>
</tr>
<tr>
<td>INTEGER SINGLE</td>
<td>INTEGER SINGLE</td>
</tr>
<tr>
<td>INTEGER DOUBLE</td>
<td>INTEGER DOUBLE</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>VECTOR SINGLE</td>
<td>VECTOR SINGLE</td>
</tr>
<tr>
<td>VECTOR DOUBLE</td>
<td>VECTOR DOUBLE</td>
</tr>
<tr>
<td>(LENGTHS MUST MATCH, OF COURSE)</td>
<td></td>
</tr>
<tr>
<td>MATRIX SINGLE</td>
<td>MATRIX SINGLE</td>
</tr>
<tr>
<td>MATRIX DOUBLE</td>
<td>MATRIX DOUBLE</td>
</tr>
<tr>
<td>(ROW/COL SIZES MUST MATCH)</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td>BOOLEAN</td>
</tr>
</tbody>
</table>

NOTE THAT THE LEGAL CONVERSIONS FOR INPUT ARG → INPUT PARM ARE THE SAME AS FOR ASSIGNMENT STATEMENTS.
ASSIGN PARAMETERS

- **ASSIGN ARGUMENTS** MUST BE HAL/S DATA ITEMS -- THEY CANNOT BE EXPRESSIONS.

- **ASSIGN ARGUMENTS** MUST MATCH THE CORRESPONDING PARAMETER IN TYPE AND PRECISION.

- MATRIX/VECTOR DIMENSIONS MUST MATCH EXACTLY.

- IF THE ASSIGN ARGUMENT IS AN ARRAY IT MUST EXACTLY MATCH THE NUMBER OF DIMENSIONS (AND RANGES) OF THE ASSIGN PARAMETER.

- AN ASSIGN ARGUMENT OF VECTOR OR MATRIX TYPE CAN BE COMPONENT SUBSCRIPTED -- PROVIDED THE EFFECT IS TO REDUCE THE VECTOR/MATRIX TO A SINGLE SCALAR.

- AN ARRAY CAN BE SUBSCRIPTED -- BUT ONLY IF ALL ARRAYNESS IS SUBSCRIPTED AWAY, I.E., THE ARRAY IS REDUCED TO A SINGLE ELEMENT.
Examples

(from programmer's guide)

DECLARE M1 MATRIX(3,3),
M2 MATRIX(3,3) DOUBLE,
M3 MATRIX(4,4),
S SCALAR,
I INTEGER,
ID INTEGER DOUBLE.

FUNCTION

ONE: PROCEDURE
CLOSE;

TWO: PROCEDURE (A,B) ASSIGN (C);
DECLARE A MATRIX(3,3);
DECLARE B INTEGER;
DECLARE C INTEGER;

DECLARE M1 MATRIX(3,3),
M2 MATRIX(3,3) DOUBLE,
M3 MATRIX(4,4),
S SCALAR,
I INTEGER,
ID INTEGER DOUBLE;

CLOSE;
LEGAL:

\[ S = S + \text{ONE}; \]

\[ S = S + M \]

\[ \text{ONE} \]

\[ M_2 = \text{TWO}(M_2, S) + M_2; \]

\[ \text{M2 is converted to single precision during transmission.} \]

\[ M_2 = \text{TWO}(M_2, 1); \]

\[ \text{I is converted to scalar type during transmission.} \]

ILLEGAL:

\[ M_2 = \text{TWO}(M_3, 1.5); \]

\[ \text{row and column dimensions of } M_3 \text{ do not match those of parameter } A. \]

\[ M_2 = \text{TWO}(M_1, \text{'ARGUMENT'}[I]); \]

\[ \text{transmission of character type argument to scalar parameter } B \text{ incurs an illegal type conversion.} \]
PROCEDURES AND FUNCTIONS (CON'T.)

EXAMPLES
(FROM PROGRAMMER'S GUIDE)

ONE: FUNCTION INTEGER;
DECLARE A MATRIX (4,4);
CLOSE;

TWO: FUNCTION(A,B) MATRIX(4,4) DOUBLE;
DECLARE A MATRIX(4,4);
DECLARE B SCALAR;
CLOSE;

DECLARE M1 MATRIX(4,4),
M2 MATRIX(4,4) DOUBLE,
M3 MATRIX(3,3),
S SCALAR,
I INTEGER;
CALL ONE;
CALL ONE(I)
- illegal: ONE possesses no parameters.

CALL TWO(M2, S+1) ASSIGN(I)
- values may be passed in and out of TWO through I.
  - type conversion required here.
  - precision conversion required here.

CALL TWO(M3, ID) ASSIGN(S)
- type conversion illegal for assign arguments.
  - precision conversion required.
  - dimension mismatch: parameter is a 3 x 3 matrix.

CALL TWO(M1, I) ASSIGN(I)
- appearance in both places is legal.
PROCEDURES AND FUNCTIONS (CON'T.)

PROCEDURE QUIRKS

THIS IS ILLEGAL:

```
ALPHA: PROCEDURE(A) ASSIGN(A);
   DECLARE A;
   . . .
   CLOSE ALPHA;
```

BUT THIS IS OK:

```
ALPHA: PROCEDURE(A) ASSIGN(B);
   DECLARE A, B;
   . . .
   CLOSE ALPHA;
   CALL ALPHA(Q) ASSIGN(Q);
```

IN THIS LATTER CASE, NO HIDDEN DANGERS EXIST IF Q IS AN INTEGER OR SCALAR OR A BIT (BOOLEAN) -- SINCE Q WILL BE PASSED BY VALUE ON THE INPUT SIDE AND ASSIGNED VIA REFERENCE ON THE ASSIGN SIDE. FOR OTHER DATA, HOWEVER, MODIFICATION OF THE ASSIGN PARAMETER CAN RESULT IN MODIFICATION OF THE INPUT PARAMETER!
PROCEDURES AND FUNCTIONS (CON'T.)

This problem will not occur if a "temporary" copy had to be made for the input argument, e.g., because a precision conversion was necessary.

Example:

```
ALPHA: PROCEDURE(V) ASSIGN(W);
    DECLARE VECTOR DOUBLE, V, W;
    W = 2V;
    X = V + VECTOR$(gDOUBLE)(1,1,2);
    
    CLOSE ALPHA;

TROUBLE
    DECLARE VECTOR DOUBLE, U;
    
    CALL ALPHA(U) ASSIGN(U);
```

This would not be a problem if, within Alpha, V were declared vector single instead!
FUNCTION RETURN

A FUNCTION IS EXITED VIA

   RETURN (exp);

WHERE (exp) IS AN EXPRESSION WHICH MATCHES THE DECLARED TYPE OF
THE FUNCTION -- EXCEPT THAT THE STANDARD IMPLICIT CONVERSIONS ARE
ALLOWED:

   SCALAR   ←→   INTEGER
   INTEGER   ←→   SCALAR
   CHARACTER   ←→   INTEGER
   SCALAR

NOTE THAT PRECISIONS NEED NOT MATCH SINCE HAL/S WILL GENERATE CODE
NECESSARY TO ACCOMPLISH PRECISION MATCHING.

NOTE: REMEMBER THAT WE CANNOT SAY

   BETA: FUNCTION(I, J)(ARRAY(6)) SCALAR;

I.E., NO ARRAY DECLARATION IS POSSIBLE. THEREFORE, A FUNCTION
CAN NEVER RETURN AN ARRAY!
BETA: FUNCTION(I, M, V) SCALAR;

DECLARE I INTEGER,
    M MATRIX DOUBLE,
    V VECTOR;

RETURN 1**2 + 6;  \[\text{WILL BE CONVERTED TO A SINGLE PRECISION SCALAR}\]

RETURN M$(3,2);  \[\text{WILL BE CONVERTED FROM DOUBLE TO SINGLE}\]

RETURN 'I = ' || I;

\[\text{ERROR -- CHARACTER STRINGS DO NOT CONVERT TO SCALAR IMPLICITLY.}\]

CLOSE BETA;
PROCEDURES AND FUNCTIONS (CON'T.)

CHARACTER STRINGS AS PARAMETERS:

All input and assign parameters of character type must be declared as:

CHARACTER(*)

I.E., specifying an indefinite length. This avoids truncation problems due to dynamically varying sizes of strings.

EXAMPLES

1. ALPHA: PROCEDURE(C1) ASSIGN(C2);
   DECLARE CHARACTER(*), C1, C2;

2. ALPHA: PROCEDURE ASSIGN(C);
   DECLARE C CHARACTER(7), ← ILLEGAL

3. BETA: FUNCTION CHARACTER(8);
4. BETA: FUNCTION CHARACTER(*), ← ILLEGAL

Note that functions of character type must specify their maximum length just as a standard declare has to!
PROCEDURES AND FUNCTIONS (CON'T.)

ARRAYS AS PARAMETERS:

2- AND 3-DIMENSIONAL ARRAY ARGUMENTS/PARAMETERS MUST ALWAYS MATCH EXACTLY IN DIMENSION AND RANGE:

ALPHA: PROCEDURE(B) ASSIGN(C);
DECLARE B ARRAY(2,4) INTEGER,
   C ARRAY(3,4,5);

   ...
CLOSE ALPHA;
DECLARE U ARRAY(8) INTEGER,
   V ARRAY(2,4) SCALAR,
   W ARRAY(3,4,5);
CALL ALPHA(U) ASSIGN(W);
   ILLEGAL: U IS 1-D, B IS 2-D
CALL ALPHA(V) ASSIGN(W);
   OK: ARRAY TEMPORARY WILL BE CREATED (EXPENSIVE)

CALL ALPHA(W$(2 AT 1, *, 3)) ASSIGN(W);
   EVEN THIS WILL WORK SINCE AN ARRAY (2,4)
   WILL RESULT -- WHICH WILL THEN REQUIRE CONVERSION TO INTEGER.
PROCEDURES AND FUNCTIONS (CON’T.)

ONE DIMENSIONAL ARRAYS ARE MORE FLEXIBLE -- IF THE PARAMETER IS DECLARED AS:

\[ \text{ARRAY}(n) \]

WHERE

\[ 2 \leq n \leq 32767 \]

THEN THE CORRESPONDING ARGUMENT MUST ALSO BE \text{ARRAY}(n), I.E., MUST MATCH EXACTLY AS IN THE 2-D AND 3-D CASES. IF:

\[ \text{ARRAY}(*) \]

IS SPECIFIED, HOWEVER, THEN THE ARGUMENTS CAN BE 1-D ARRAYS OF ANY LEGAL LENGTH. THIS ALLOWS PROCEDURES TO OPERATE ON ARRAYS OF VARIABLE SIZE.

EXAMPLES:

1. \text{ALPHA}: \text{PROCEDURE}(A);
   \text{DECLARE A ARRAY}(6) \text{ INTEGER};
   \ldots
   \text{CLOSE ALPHA};

   ALPHA CAN ONLY BE CALLED WITH AN ARRAY(6) INPUT ARGUMENT:

   \text{DECLARE B ARRAY}(6) \text{ INTEGER};
   \text{CALL ALPH(A)(B);}
PROCEDURES AND FUNCTIONS (CON'T.)

(2) \textsc{Alpha}: \textsc{procedure(a)};

\textsc{Declare a array(*) integer};
\ldots
\textsc{Close a};
\ldots
\textsc{Declare c array(6) integer,}
\textsc{d array(30000) integer double,}
\textsc{e array(4) scalar};
\textsc{call a(c)};
\text{ \{ THIS IS ALL LEGAL! \}}
\text{ call a(d)};
\text{ \{ \}}
\text{ call a(e)};
\text{ \}}

\text{Within the procedure, the actual length of the array(*) items can be found by means of the built-in function \textsc{size(a)}}
PROCEDURES AND FUNCTIONS

EXAMPLE

ALPHA: PROCEDURE(A) ASSIGN(B);
DECLARE ARRAY(*) SCALAR, A, B;
DECLARE I INTEGER;

DO FOR I = 1 TO SIZE(A);
    B$I = SINH(A$I);
END;

B = SINH(A), AN EXAMPLE OF AN ARRAYED ASSIGNMENT (WILL BE DISCUSSED LATER)

CLOSE ALPHA;

IMPORTANT POINT: THE COMPILER CAN ALSO OBTAIN THE SIZES OF ARRAY(*) AT RUN-TIME.
HAL/S typically supports the following set of built-in functions. Minor variations may arise between implementations.

### ARITHMETIC FUNCTIONS

- Arguments may be INTEGER or SCALAR types
- In functions with one argument, result type matches argument type (except as specifically noted)
- In functions with two arguments, unless specifically specified, result type is scalar if either or both arguments are scalar; otherwise the result type is integer.
- Arrayed arguments cause multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match.

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABS(a)</td>
<td>$</td>
</tr>
<tr>
<td>CEILING(a)</td>
<td>Smallest integer $\geq a$</td>
</tr>
<tr>
<td>DIV(a,B)</td>
<td>Integer division $a/B$ (arguments rounded to integers)</td>
</tr>
<tr>
<td>FLOOR(a)</td>
<td>Largest integer $\leq a$</td>
</tr>
<tr>
<td>MIDVAL(a,B,Y)</td>
<td>The value of the argument which is algebraically between the other two. If two or more arguments are equal, the multiple value is returned. Result is always scalar.</td>
</tr>
<tr>
<td>MOD(a,B)</td>
<td>$a \mod B$</td>
</tr>
<tr>
<td>ODD(a)</td>
<td>TRUE 1 if $a$ odd { result is FALSE 0 if $a$ even \ BOOLEAN</td>
</tr>
<tr>
<td>REMAINDER(a,B)</td>
<td>Signed remainder of integer division $a/B$ (argument rounded to integer)</td>
</tr>
</tbody>
</table>

C-1

TERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 861-1840
HAL/S-FC COMPILER SYSTEM
SPECIFICATION
IR-95-5
1 March 1976

INTERMETRICS APPROVAL

Daniel J. Lickly
Head, HAL Language Compiler Department

Dr. F. H. Martin
Shuttle Program Manager

INTERMETRICS
DMDVAL

HAL/S-FC LIBRARY ROUTINE DESCRIPTION

Source Member Name: DMDVAL

Size of Code Area: 20 Bw

Stack Requirement: 18 Bw

Data CSECT Size: 0 Bw

Intrinsic Procedure

Other Library Modules Referenced: None

ENTRY POINT DESCRIPTIONS

Primary Entry Name: DMDVAL

Function: Finds mid value of three double precision scalar arguments.

Invoked by:

Compiler emitted code for HAL/S construct of the form:

MIDVAL(A,B,C) where A,B,C are double precision scalars.

Other Library Modules:

Execution Time (microseconds): 41.4

Input Arguments:

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision</th>
<th>How Passed</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>DP</td>
<td>F0</td>
<td></td>
</tr>
<tr>
<td>scalar</td>
<td>DP</td>
<td>F2</td>
<td></td>
</tr>
<tr>
<td>scalar</td>
<td>DP</td>
<td>F4</td>
<td></td>
</tr>
</tbody>
</table>

Output Results:

<table>
<thead>
<tr>
<th>Type</th>
<th>Precision</th>
<th>How Passed</th>
<th>Units</th>
</tr>
</thead>
<tbody>
<tr>
<td>scalar</td>
<td>DP</td>
<td>F0</td>
<td></td>
</tr>
</tbody>
</table>

Errors Detected:

Error #: Cause Fixup

Comments:

Registers Unsafe Across Call: F0,F1,F2,F3,F4,F5.

Algorithm:

IF A = B THEN RETURN A;
IF A < B THEN DO;
    IF B <= C THEN RETURN B;
    ELSE IF A <= C THEN RETURN C;
    ELSE RETURN A;
END;
ELSE DO;
    IF C <= B THEN RETURN B;
    ELSE IF C <= A THEN RETURN C;
    ELSE RETURN A;
END;

INTERMETRICS INCORPORATED • 701 CONCORD AVENUE • CAMBRIDGE, MASSACHUSETTS 02138 • (617) 661-1840
<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ROUND(a)</td>
<td>nearest integral value to a</td>
</tr>
</tbody>
</table>
| SIGN(a)       | +1 a > 0  
                 -1 a < 0  |
| SIGNUM(a)     | +1 a > 0  
                 0 a = 0   
                 -1 a < 0  |
| TRUNCATE(a)   | largest integer < |a| times SIGNUM (integer(a)) |
ALGEBRAIC FUNCTIONS

- arguments may be integer or scalar types - conversion to scalar occurs with integer arguments
- result type is always scalar
- arrayed arguments cause multiple invocations of the function, one for each array element
- angular values are supplied or delivered in radians.

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARCCOS((a))</td>
<td>(\cos^{-1} a), (</td>
</tr>
<tr>
<td>ARCCOSH((a))</td>
<td>(\cosh^{-1} a), (a &gt; 1)</td>
</tr>
<tr>
<td>ARCSIN((a))</td>
<td>(\sin^{-1} a), (</td>
</tr>
<tr>
<td>ARCSINH((a))</td>
<td>(\sinh^{-1} a)</td>
</tr>
<tr>
<td>ARCTAN2((a, b))</td>
<td>(-\pi &lt; \tan^{-1}(a/b) \leq \pi) Proper Quadrant if: (k \sin \theta = a), (k \cos \theta = b), (k &gt; 0)</td>
</tr>
<tr>
<td>ARCTAN((a))</td>
<td>(\tan^{-1} a)</td>
</tr>
<tr>
<td>ARCTANH((a))</td>
<td>(\tanh^{-1} a), (</td>
</tr>
<tr>
<td>COS((a))</td>
<td>(\cos a)</td>
</tr>
<tr>
<td>COSH((a))</td>
<td>(\cosh a)</td>
</tr>
<tr>
<td>EXP((a))</td>
<td>(e^a)</td>
</tr>
<tr>
<td>LOG((a))</td>
<td>(\log_e a), (a &gt; 0)</td>
</tr>
<tr>
<td>SIN((a))</td>
<td>(\sin a)</td>
</tr>
<tr>
<td>SINH((a))</td>
<td>(\sinh a)</td>
</tr>
<tr>
<td>SQRT((a))</td>
<td>(\sqrt{a}), (a \geq 0)</td>
</tr>
<tr>
<td>TAN((a))</td>
<td>(\tan a)</td>
</tr>
<tr>
<td>TANH((a))</td>
<td>(\tanh a)</td>
</tr>
</tbody>
</table>
**VECTOR-MATRIX FUNCTIONS**

- arguments are vector or matrix types as indicated
- result types are as implied by mathematical operation
- arrayed arguments cause multiple invocations of the function, one for each array element

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ABVAL(a)</td>
<td>length of vector a</td>
</tr>
<tr>
<td>DET(a)</td>
<td>determinant of square matrix a</td>
</tr>
<tr>
<td>INVERSE(a)</td>
<td>inverse of nonsingular square matrix a</td>
</tr>
<tr>
<td>TRACE(a)</td>
<td>sum of diagonal elements of square matrix a</td>
</tr>
<tr>
<td>TRANSPOSE(a)</td>
<td>transpose of matrix a</td>
</tr>
<tr>
<td>UNIT(a)</td>
<td>unit vector in same direction as vector a</td>
</tr>
</tbody>
</table>
### MISCELLANEOUS FUNCTIONS

- Arguments are as indicated; if none are indicated, the function has no arguments.
- Result type is as indicated.

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Result Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOCKTIME</td>
<td>scalar</td>
<td>returns time of day</td>
</tr>
<tr>
<td>DATE</td>
<td>integer</td>
<td>returns date (implementation dependent format)</td>
</tr>
<tr>
<td>ERRGRP</td>
<td>integer</td>
<td>returns group number of last error detected, or zero</td>
</tr>
<tr>
<td>ERRNUM</td>
<td>integer</td>
<td>returns number of last error detected, or zero</td>
</tr>
<tr>
<td>PRIORI</td>
<td>integer</td>
<td>returns priority of process calling function</td>
</tr>
<tr>
<td>RANDOM</td>
<td>scalar</td>
<td>returns random number from rectangular distribution over range 0-1</td>
</tr>
<tr>
<td>RANDOMG</td>
<td>scalar</td>
<td>returns random number from Gaussian distribution mean zero, variance one.</td>
</tr>
<tr>
<td>RUNTIME</td>
<td>scalar</td>
<td>returns Real Time Executive clock time (Section 8.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>NEXTIME (label)</th>
<th>scalar</th>
<th>&lt;label&gt; is the name of a program or task. The value returned is determined as follows:</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>a) If the specified process was scheduled with the REPEAT EVERY option and has begun at least one cycle of execution, then the value is the time the next cycle will begin.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>b) If the specified process was scheduled with the IN or AT phrase, and has not yet begun execution, then the value is the time it will begin execution.</td>
</tr>
<tr>
<td></td>
<td></td>
<td>c) Otherwise, the value is equal to the current time (RUNTIME function).</td>
</tr>
<tr>
<td>Name, Arguments</td>
<td>Result Type</td>
<td>Comments</td>
</tr>
<tr>
<td>-----------------</td>
<td>-------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| **SHL(a, b)**   | Same as a   | a may be integer or bit type. b must be integer type.  
If a is integer type, the result is an integer whose internal binary representation is that of a shifted left by b bit locations. The signed nature of the integer a is taken into account in an implementation dependent manner which depends upon the number system and word size of the target computer.  
If a is bit type, the result is a bit string containing the value of a shifted left by b bit locations. a is treated as an unsigned logical quantity. The size of the result is implementation dependent.  
Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match. |
| **SHR(a, b)**   | Same as a   | a may be integer or bit type. b must be integer type.  
Results are as defined for the SHL function except that all shifting occurs to the right.  
Arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match. |
# CHARACTER FUNCTIONS

- First argument is character type - second argument is as indicated (any argument indicated as character type may also be integer or scalar, whereupon conversion to character type is implicitly assumed)
- Result type is as indicated
- Arrayed arguments produce multiple invocations of the function, one for each array element - arraynesses of arrayed arguments must match

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Result Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>INDEX(α, β)</td>
<td>integer</td>
<td>β is character type - if string β appears in string α, index pointing to the first character of β is returned; otherwise zero is returned</td>
</tr>
<tr>
<td>LENGTH(α)</td>
<td>integer</td>
<td>Returns length of character string</td>
</tr>
<tr>
<td>LJUST(α, β)</td>
<td>character</td>
<td>β is integer type - string α is expanded to length β by padding on the right with blanks β &gt; length(α)</td>
</tr>
<tr>
<td>RJUST(α, β)</td>
<td>character</td>
<td>β is integer type - string α is expanded to length β by padding on the left with blanks β &gt; length(α)</td>
</tr>
<tr>
<td>TRIM(α)</td>
<td>character</td>
<td>Leading and trailing blanks are stripped from α</td>
</tr>
</tbody>
</table>
BIT FUNCTIONS

- arguments are bit type
- result is bit type
- arrayed arguments produce multiple invocations of the function, one for each array element - arrayness of arrayed arguments must match

<table>
<thead>
<tr>
<th>Name, Arguments</th>
<th>Result Type</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>XOR(a,b)</td>
<td>bit</td>
<td>Result is Exclusive OR of a and b. Length of result is length of longer argument. Shorter argument is left padded with binary zeros to length of longer argument.</td>
</tr>
</tbody>
</table>

ARRAY FUNCTIONS

- arguments are n-dimensional arrays where n is arbitrary
- arguments are integer or scalar type
- result type matches argument type and is unarrayed

<table>
<thead>
<tr>
<th>Name, Parameters</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAX(a)</td>
<td>maximum of all elements of a</td>
</tr>
<tr>
<td>MIN(a)</td>
<td>minimum of all elements of a</td>
</tr>
<tr>
<td>PROD(a)</td>
<td>product of all elements of a</td>
</tr>
<tr>
<td>SUM(a)</td>
<td>sum of all elements of a</td>
</tr>
<tr>
<td>Name, Argument</td>
<td>Comments</td>
</tr>
<tr>
<td>----------------</td>
<td>----------</td>
</tr>
</tbody>
</table>
| SIZE(a)        | One of the following must hold:  
|                | • \( a \) is an unsubscripted arrayed variable with a one-dimensional array specification - function returns length of array.  
|                | • \( a \) is an unsubscripted major structure with a multiple array specification - function returns number of elements.  
|                | • \( a \) is an unsubscripted structure terminal with a one-dimensional array specification - function returns length of array.  
|                | Result is of integer type |
BIT STRINGS

REMEMBER THAT BOOLEAN = BIT(1)

(A) BIT STRING LITERALS

- BIN(n)'bbb...b' OR BIN'b...b'
- OCT(n)'ooo...o' OR OCT'o...o'
- HEX(n)'hhh...h' OR HEX'h...h'
- DEC(n)'ddd...d' OR DEC'd...d'

- THE REPETITION COUNT (n) IS OPTIONAL
- THE RESULTING BIT LITERAL MUST HAVE BETWEEN 1 AND 32 BITS.

EXAMPLES

<table>
<thead>
<tr>
<th>BIT LITERAL</th>
<th>LENGTH</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN'1011'</td>
<td>4</td>
</tr>
<tr>
<td>BIN(3)'101'</td>
<td>9</td>
</tr>
<tr>
<td>HEX(2)'FFE'</td>
<td>24</td>
</tr>
<tr>
<td>OCT'714'</td>
<td>9</td>
</tr>
<tr>
<td>DEC'25'</td>
<td>5</td>
</tr>
</tbody>
</table>
BIT DECLARATION

DECLARE name BIT(n),

WHERE \[1 \leq n \leq 32\]

AGAIN, NOTE THAT:

DECLARE B BIT(1),

AND

DECLARE B BOOLEAN,

ARE EQUIVALENT. BIT DECLARATIONS CAN BE COMBINED WITH OTHER DECLARED DATA IN COMPOUND AND FACTORED DECLARES, E.G.,

DECLARE BIT(4), B1, B2, B3,
DECLARE I INTEGER, B1 BOOLEAN,
   B2 BIT(32), S SCALAR DOUBLE,
   B3 BIT(6);
BIT STRINGS (CON'T.)

BIT STRING INITIALIZATION

BIT STRINGS ARE INITIALIZED VIA AN INITIAL/CONSTANT LIST CONTAINING BIT STRING LITERALS:

DECLARE B1 BIT(16) INITIAL(HEX'33F'),
    B2 BIT(1) CONSTANT(FALSE),
    B3 BIT(8) CONSTANT(BIN'11100111'),
    B4 ARRAY(3) BIT(2) INITIAL(BIN'10', BIN'00', BIN'11');

LITERALS ARE PADDED ON THE LEFT WITH ZEROS OR TRUNCATED ON THE LEFT AS REQUIRED.

DECLARE B1 BIT(8) INITIAL(BIN'1'),
    B2 BIT(4) INITIAL(HEX'FF');

RESULTS IN:

B1 = 00000001
B2 = 1111 (4 BITS LOST)
BIT STRINGS (CON’T.)

(1) BIT STRING SUBSCRIPTING

- COMPONENT SUBSCRIPTING -

(UNARRAYED BITS)

* TO SELECT THE I\textsuperscript{TH} BIT FROM A BIT STRING (RESULT IS A BIT(I)):
  
  \[
  \text{BIT\_STRING}\$I
  \]

  \[
  \text{WHERE}
  \]

  \[
  1 \leq I \leq L \quad (L = \text{LENGTH OF BIT\_STRING})
  \]

  'I' CAN BE VARIABLE, I.E., AN EXPRESSION.

* TO SELECT I BITS FROM A BIT STRING STARTING AT THE J\textsuperscript{TH} (RESULT IS A BIT(I)):

  \[
  \text{BIT\_STRING}\$(I \text{ AT J})
  \]

  \[
  \text{WHERE}
  \]

  \[
  1 \leq I \leq L \quad \text{AND IS KNOWN AT COMPILE TIME}
  \]

  AND \[
  1 \leq J \leq L - I + 1 \quad \text{AND MAY BE VARIABLE (AN EXPRESSION)}.
  \]
BIT STRINGS (CON'T.)

- To select a substring starting with the $I^\text{th}$ bit and
  ending with the $J^\text{th}$,

  \[ \text{BIT\_STRING}$(I \text{ TO } J)$ \]

Where

$I$ and $J$ are known at compile time, and

$1 \leq I < J \leq 32$

Note (1): The essence of the rules for subscripting is that

the resultant bit string must have a length known

at compile time (and of course lie within $1$ and $32$).

Note (2): Since booleans are equivalent to \text{BIT}(1), they may

be component subscripted -- although this is senseless.

This explains, however, the fact that array subscripts

on boolean arrays must have a trailing colon:

\[ \text{BOOL\_TABLE}$(1:) \]
BIT STRINGS (CON'T.)

ARRAYED BIT STRINGS

BIT STRINGS (AND BOOLEANS) MAY, OF COURSE, BE ORGANIZED INTO ARRAYS OF 1, 2, OR 3 DIMENSIONS.

ASSUME:

B IS AN ARRAY(4) OF BIT(5) STRINGS:

\[ B = (10110_2, 00000_2, 01100_2, 11111_2) \]

THEN

\[ B(2:) = 00000_2 \quad \text{A BIT(5)} \]
\[ B(2:3) = 0_2 \quad \text{A BIT(1)} \]
\[ B(1:2 AT 3) = 11_2 \quad \text{A BIT(2)} \]
\[ B(2 AT 2: 3 AT 3) = (000_2, 100_2) \quad \text{ARRAY(2) BIT(3)} \]
\[ B(*:1) = (1_2, 0_2, 0_2, 1_2) \quad \text{ARRAY(4) BIT(1)} \]
BIT STRINGS (CON’T.)

LEGAL OPERATIONS: (SAME AS BOOLEANS WITH THE ADDITION OF CatenATION).

& \text{ INTERSECTION}
\text{AND}

\text{CONJUNCTION}

\text{OR}

\text{COMPLEMENT}

\text{NOT}

\text{CATENATION}

\text{COMPLEMENT} - INVERTS THE LOGICAL VALUE OF EVERY BIT IN THE STRING.

\text{IF } B = 10110111 \\
\text{THEN } \neg B = 01001000
BIT STRINGS (CON'T.)

CONJUNCTION -- OR's ALL CORRESPONDING BITS TOGETHER
(BINARY OPERATION).

IF $B_1 = 11101011$
$B_2 = 10011100$

THEN $B_1 \lor B_2 = 11111111$

NOTE: IF THE TWO BIT STRINGS ARE OF UNEQUAL LENGTH, THE
SHORTER IS PADDED ON THE LEFT WITH ZEROS BEFORE ORING.
THE RESULTANT BIT STRING HAS THE LENGTH OF THE LONGER
STRING.

IF $B_1 = 101$
$B_2 = 111000$

$B_1 \lor B_2 = 111101$ (A BIT(6))
BIT STRINGS (CONT.)

INTERSECTION - AND's ALL CORRESPONDING BITS TOGETHER (BINARY OPERATION).

IF $B_1 = 1101001$
$B_2 = 0110010$
THEN $B_1 \text{ AND } B_2 = 0100000$

NOTE: IF THE TWO BIT STRINGS ARE OF UNEQUAL LENGTH, THE SHORTER IS PADDED ON THE LEFT WITH ZEROS BEFORE ANDING. THE RESULTANT BIT STRING HAS THE LENGTH OF THE LONGER STRING.

IF $B_1 = 101$
$B_2 = 110110$
THEN $B_1 \text{ AND } B_2 = 000100$ (BIT(6))
BIT STRINGS (CON'T.)

CATENATION - Two bit strings (Booleans) can be chained (catenated) together to form a single (longer) bit string. The second string is appended to the end of the first. If the resultant bit string exceeds 32, then the leftmost excess bits are truncated.

IF \( B_1 = \text{TRUE} = \text{BIN'1} \)
\( B_2 = 10110 \)
THEN \( B_1 \ CAT \ B_2 = 110110 \) BIT(6)

IF \( B_1 = 11011 \)
\( B_2 = 10101 \)
THEN \( B_1||B_2 = 1101110101 \) BIT(10)

IF \( B_1 = 10101010101010101 \) BIT(17)
\( B_2 = 11011011011011011 \) BIT(18)
\( B_1||B_2 = 0101101010101010101010101010101010 \)
THROWN AWAY

\( B_1||B_2 = 010101010101010110110110110110110 \) A BIT(32)
BIT STRINGS (CON'T.)

F PRECEDENCE

\[ \begin{array}{c|c|c|c|c}
\text{HI} & \neg, \text{NOT} & \text{COMPLEMENT} \\
\text{LO} & \text{CAT} & \text{CATENATION} \\
& \text{AND} & \text{INTERSECTION} \\
& \text{OR} & \text{CONJUNCTION} \\
\end{array} \]

SEQUENCES OF OPERATIONS OF THE SAME PRECEDENCE ARE EVALUATED LEFT TO RIGHT.

EXAMPLE:

IF \((B1 | \neg B2 | B3 \& B4 | B5 | B6) = \text{ON}\)

I.E., EQUIVALENT TO:

IF \(((B1 | \neg B2) | (B3 \& B4) | (B5 | B6)) = \text{ON}\)

(WHICH IS EASIER TO READ)
BIT STRINGS (CON'T.)

(6) BIT STRING ASSIGNMENTS

L_STRING = R_STRING

- NEITHER, EITHER, OR BOTH MAY BE SUBSCRIPTED.
- IF L_STRING > R_STRING IN LENGTH, THEN R_STRING IS LEFT PADDED WITH ZEROES PRIOR TO ASSIGNMENT
- IF L_STRING < R_STRING IN LENGTH, THEN R_STRING IS TRUNCATED FROM THE LEFT AS NEEDED.

EXAMPLES:

LET B1 BE A BIT(8)
    B2 BE A BIT(6) = 110110
THEN B1 = B2 RESULTS IN
    B1 = 00110110

IF B3 IS A BIT(10) = 1010111101
THEN B1 = B3 RESULTS IN
    B1 = 10111101
BIT STRINGS (CON'T.)

IF $B_1 = 11001110$, BIT(8)
    $B_2 = 110110$, BIT(6)

THEN

A) $B_1$(3 AT 2) = $B_2$ RESULTS
    IN $B_1 = 11101110$
      I.E.,
      \[
      \begin{array}{c}
      110110, \\
      11001110 \rightarrow 11101110
      \end{array}
      \]

B) $B_1$(5 AT 3) = $B_2$(2 TO 4) RESULTS
    IN $B_1 = 11001010$
      I.E.,
      \[
      \begin{array}{c}
      110110, \\
      00101 \quad \text{LEFT PADDED} \\
      11001110 \rightarrow 11001010
      \end{array}
      \]

C) $B_1$(3) = $B_2$ RESULTS
    IN $B_1 = 11001110$
      I.E.,
      \[
      \begin{array}{c}
      110110 \quad \text{LEFT TRUNCATED} \\
      11001110 \rightarrow 11001110
      \end{array}
      \]
BIT STRINGS (CON'T.)

BIT STRINGS IN CONDITIONAL STATEMENTS

- IF BIT STRINGS ARE COMPONENT SUBSCRIPTED DOWN TO A SINGLE BIT, THEN THEY MAY BE MIXED WITH BOOLEANS TO FORM BOOLEAN EXPRESSIONS.

EXAMPLES:

\[
\begin{align*}
B1 & \text{ BIT(4)} = 1101 \\
B2 & \text{ BIT(8)} = 10110100 \\
B3 & \text{ BOOLEAN = TRUE} \\
B4 & \text{ BOOLEAN = FALSE}
\end{align*}
\]

IF \( B1 \) THEN \( X = 0 \); \hspace{1cm} \text{ILLEGAL -- B1 IS NOT A BOOLEAN.}

IF \( B1 \& 1 \) THEN \( X = 0 \); \hspace{1cm} \text{TRUE WILL BE EXECUTED}

IF \( (B1 \& B2) \) THEN \( X = 0 \); \hspace{1cm} \text{ILLEGAL -- B1 IS NOT A BOOLEAN.}

IF \( (B1 \& 2 \& B2 \& 3 \& B4) \) THEN \( X = 0 \);

\text{TRUE FALSE WILL BE EXECUTED}
BIT STRINGS (CON'T.)

- BIT STRINGS MAY BE COMPARED FOR = OR = (CLASS II OPERATORS) TO FORM RELATIONAL EXPRESSIONS.

E.G.,

DO WHILE B1 = B2;
    IF B2 = BIN'101' THEN DO;

IF THE BIT OPERANDS ARE OF UNEQUAL LENGTH THEN, OF COURSE, THE SHORTER IS LEFT-PADDED WITH THE REQUISITE NUMBER OF ZEROS PRIOR TO THE COMPARISON.

EXAMPLES:

B1 = 101
B2 = 11
B3 = 1100101
B4 = 00101

THEN B1 = B4 IS TRUE
B1 = B2 IS FALSE
B1 = B2 IS TRUE
B3 = B4 IS FALSE
B3$(3 TO 7) = B4 IS TRUE
(B3 || B4) = B3 IS TRUE
BIT STRING COMPARISONS

TRUE = ON = 0000 ... 0001
FALSE = OFF = 0000 ... 0000

SUPPOSE:

DECLARE B1 BIT(8) INITIAL(BIN'11011101');
DECLARE B2 BIT(4) INITIAL(BIN'1010');

(A) IF B1 THEN DO; (ILLEGAL)
(B) IF B1 = TRUE THEN DO;
     -- FALSE BECAUSE '11011101' ≠ '00000001'
(C) IF B2 = FALSE THEN DO;
     -- FALSE BECAUSE '1010' ≠ '0000'
(D) IF B1$8 THEN DO;
     -- TRUE
(E) IF B2$(2 AT 2) = TRUE THEN DO;
     -- TRUE BECAUSE '01' = '01'
(F) IF B1$(3 AT 4) = BIN'111' THEN DO;
     -- TRUE
BIT STRINGS (CON'T.)

SUMMARY

A CONDITIONAL EXPRESSION, E.G.

IF (cond exp) THEN ...

DO WHILE (cond exp);

DO UNTIL (cond exp);

IS MADE UP OF COMBINED RELATIONAL EXPRESSIONS, E.G.

(A < 0) & (B = BIN'101')1S < 2

OR COMBINED BOOLEAN EXPRESSIONS, E.G.

(B1||(B2 & B3))&B21 B3

WHERE B1, ... B3 ARE BOOLEANS

NOTE: A BIT STRING THAT IS SUBSCRIPTED DOWN TO 1
BIT CAN BE USED AS A BOOLEAN.

BUT, A CONDITIONAL EXPRESSION CANNOT BE A MIXTURE:

IF (B1 & B2) | (B = B) ... IS ILLEGAL!!

RELATIONAL EXPRESSION

BIT EXPRESSION
CONDITIONAL EXPRESSIONS (CON'T.)

CONDITIONAL EXPRESSION IS EITHER A RELATIONAL EXPRESSION OR A BOOLEAN EXPRESSION:

RELATIONAL: (A<3) | (M = N) & (S = T**2)

BOOLEAN: B1 & (T2|B3) & B4|B5
KEY POINT: A BIT(1) IS EXACTLY EQUIVALENT TO A BOOLEAN AND MAY BE USED IN A BOOLEAN EXPRESSION;

EXAMPLE 1: (MIXING BIT(1) AND BOOLEAN)
DECLARE BA BIT(1), BB BOOLEAN,
      BC BOOLEAN;
IF BA|(¬BB&BC) THEN ...

EXAMPLE 2: (MAKING A BIT(1) BY SUBSCRIPTING)
DECLARE QBITS BIT(8), BB BOOLEAN,
      BC BOOLEAN;
IF QBITS$3|(BB&BC) THEN ...

EXAMPLE 3: (ILLEGAL USAGES)
IF QBITS THEN ... (THIS WILL NOT WORK EVEN THOUGH THE PROGRAMMER'S GUIDE IMPLIES THAT IT WILL EVALUATE TO TRUE IF THE LOWEST BIT IS A 1!!)

EXAMPLE 4: (FIXUP OF EXAMPLE 3)
IF QBITS = TRUE ... (NOW IT IS A RELATIONAL EXP)
CONDITIONAL EXPRESSIONS (CONT'D.)

1 Mt BOOL:
1 Mt PROGRAM:
2 Mt DECLARE BOOLEAN INITIAL(FALSE),
2Mt BA, BB, BC, BD, BE, BF;
3 Mt DECLARE INITIAL(BIN'0'),
3 Mt B1 BIT(1),
3 Mt B2 BIT(2),
3 Mt B3 BIT(3),
3 Mt B4 BIT(4),
3 Mt B5 BIT(5),
3 Mt B6 BIT(6),
3 Mt B7 BIT(7),
3 Mt B8 BIT(8),
3 Mt B9 BIT(9),
3 Mt B10 BIT(10),
3 Mt B11 BIT(11),
3 Mt B12 BIT(12),
3 Mt B13 BIT(13),
3 Mt B14 BIT(14),
3 Mt B15 BIT(15),
3 Mt B16 BIT(16),
3 Mt B17 BIT(17),
4 Mt DECLARE Q0 BIT(5) INITIAL(BIN'11110101010101010101'),
5 Mt IF BA & BB | BC & (BD | BE) THEN
6 Mt DO;
7 Mt END;

5-16.4
CONDITIONAL EXPRESSIONS (CONT'D.)

IF \( B_1 = B_2 \) \& \( B_3 \neq B_4 \) \& \( B_5 = B_6 \) THEN

DO;

END;

IF \( B_A = B_B \) \& \( B_C = B_D \) \& \( B_A \neq B_E \) THEN

DO;

END;

IF \( B_A \neq B_B \) \& \( B_C \neq B_D \) \& \( B_6 \neq B_8 \) THEN

DO;

END;

...IF \( B_A \) THEN

DO;

END;

IF \( B_4 \) THEN

ERROR #1 OF SEVERITY 1.

BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN
23 MT
S T
24 MT
DO;
HABL/S FC- 9.21
TMT
25 MT
END;

26 MT
S T
27 MT
DO;
28 MT
END;
29 MT
IF (B6 + B7) THEN
30 MT
DO;
31 MT
END;
32 MT
IF (B3 = TRUE) + (B4 = FALSE) + (B1 = BIN'1011') THEN
33 MT
DO;
34 MT
END;
35 MT
IF B5 = TRUE THEN
36 MT
DO;
37 MT
END;
38 MT
IF B10 THEN
39 MT
DO;
END;
40 MT
END;

If (BA + TRUE) + (BB = FALSE) + (B1 = BIN'01011') THEN
DO;
END;

**ERROR #2 OF SEVERITY 4.**

**BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN**

**LAST ERROR WAS DETECTED AT STATEMENT 28.**

**ERROR #3 OF SEVERITY 1.**

**BIT EXPRESSION IN IF CLAUSE MUST BE BOOLEAN**

**LAST ERROR WAS DETECTED AT STATEMENT 29.**
BIT STRINGS (CON'T.)

1 BIT STRING ARGUMENTS & PARAMETERS

- BIT STRINGS MAY BE **INPUT PARAMETERS (ARGUMENTS)**
  OF PROCEDURES AND FUNCTIONS AND **ASSIGN PARAMETERS**
  (ARGUMENTS) OF PROCEDURES.

**EXAMPLE:**

FLAGS: PROCEDURE(B1) ASSIGN(B2);

DECLARE B1 BIT(16),
    B2 BIT(8);

CLOSE FLAGS;

**RULES (INPUT PARAMETERS)**

- BOTH PARAMETER AND ARGUMENT MUST BE OF BIT TYPE
  (NO IMPLICIT CONVERSIONS).

- THE ARGUMENT IS PADDED/TRUNCATED ON THE LEFT AS
  NECESSARY TO FIT THE INPUT PARAMETER.
RULES: (ASSIGN PARAMETERS)

- THE ASSIGN ARGUMENT MUST BE A DECLARED BIT DATA ITEM.
- ARGUMENT AND PARAMETER LENGTHS MUST MATCH EXACTLY!
- NO SUBSCRIPTING (COMPONENT) IS ALLOWED ON THE ARGUMENT.

Examples:

Let the following data be declared:

```
DECLARE D1 BIT(16),
    B2 BIT(3);
```

and let the following procedure be defined:

```
SWITCHES: PROCEDURE(D2) ASSIGN(D1);
    DECLARE D1 BIT(3),
    D2 BIT(8);
    ...
    CLOSE SWITCHES;
```

Both legal and illegal invocations of this procedure are shown below:

- `CALL SWITCHES(D1|BIN'1001') ASSIGN(D2);`
  - this 16-bit quantity truncated to 9 bits on passage
- `CALL SWITCHES(D2) ASSIGN(D1);`
  - illegal - length mismatch
  - this 3-bit quantity padded to 8 bits on passage
- `CALL SWITCHES(BIN'1') ASSIGN(FALSE);`
  - illegal - not a declared bit string data item.
BIT STRINGS (CON'T.)

(1) BIT STRING FUNCTIONS

\[ \text{FUNCTION}(i_1, i_2, \ldots) \text{ BIT}(n); \]

OPTIONAL INPUT \[ 1 \leq n \leq 32 \]
PARAMETERS

AN INVOCATION OF SUCH A BIT FUNCTION BEHAVES LIKE A BIT\(^n\) DATA ITEM IN ALL CONTEXTS!

EXAMPLES:

(1) \( F: \text{FUNCTION}(B) \text{ BIT}(3); \)

DECLARE B BIT(8);

\[
\begin{align*}
\text{RETURN } \text{B}; & \quad \text{LEFTMOST 5 BITS TRUNCATED} \\
\text{RETURN } \text{B}$4; & \quad \text{RESULT Padded (on left) to 3 BITS} \\
\text{RETURN } 6; & \quad \text{ILLEGAL! BIT STRING QUANTITY NOT BEING RETURNED.}
\end{align*}
\]

CLOSE F;
BIT STRING (CON'T.)

(2) G: FUNCTION(B) BIT(6);

\[ G: \text{FUNCTION}(B) \text{ BIT(6)}; \]

\[ \ldots \]

\[ \text{CLOSE G}; \]

\[ \ldots \]

IF B1 | (B2 & G(B3)) | G(B4) = BIN'101101' THEN DO;

\[ \text{returns a BIT(6)} \]

---

**XOR FUNCTION**

_XOR_ **IS A RECENTLY ADDED BUILT-IN FUNCTION THAT PERFORMS AN EXCLUSIVE OR ON TWO INPUT BIT STRINGS AND RETURNS THE RESULT AS A BIT STRING.**

**SYNTAX:**

\[ \text{BIT\_STRING} = \text{XOR(BIT\_STRING1, BIT\_STRING2)}; \]

\[ \text{OR} \]

\[ \text{IF B1} | \text{XOR(B2, B3)} | (B4 \& B5) = \text{BIN}'011010' \text{ THEN ...} \]

**NOTE:** IF THE TWO INPUTTED BIT STRINGS ARE UNEQUAL IN LENGTH, THE SHORTER IS LEFT-PADD ED WITH ZEROES. THE RESULT IS A BIT STRING EQUAL IN LENGTH TO THE LONGEST INPUTTED STRING.
HAL/S allows two types of data organizations: arrays & structures.

- **Arrays**: 1-, 2-, or 3-dimensional patterns of homogeneous data items.
  
  **Example**:
  
  DECLARE AX ARRAY(3,2,100) INTEGER INITIAL(5);
  
  Each array dimension can range from 2 to 32,767.

- **Structures**: Hierarchical (tree-like) collections of homogeneous or heterogeneous data items. A structure is more general than an array since it can contain arrays (and even other structures).

  A structure must reference a structure template (think of it as a dsect in 360 terminology) that was previously defined.
A structure template is a tree:

- **START**
- **STOP**

- Level 0
  - □ "root"
  - □ "fork"
  - □ "leaf"

- Level 1
- "branch"

- Level 2

- Level 3

- Level 4
STRUCTURES (CON'T.)

NOTE: THE PROGRAMMER'S GUIDE MAY BE CONSULTED FOR AN ABSTRACT DESCRIPTION OF STRUCTURE TEMPLATES. HERE WE WILL DEVELOP THE CONCEPT USING CONCRETE EXAMPLES.

A) STRUCTURE TEMPLATE:

\[ \text{Define} \quad \text{STRUCTURE} \quad \text{name} = \text{node}_1, \text{node}_2, \text{node}_3, \ldots \text{node}_n \; ; \]

WHERE:

\[ \text{node}_i \equiv \text{name} \]

(MINOR STRUCTURES OR FORKS)

\[ \text{node}_i \equiv \text{name} \; \text{attributes} \]

(STRUCTURE TERMINAL)

NOTE: THE STRUCTURE LEVEL 'n' MUST SATISFY \(1 \leq n \leq 5\)

CURRENT LIMITATION
EXAMPLE 1

STRUCTURE QQ:

1: A INTEGER,
1: B BOOLEAN,
1: M MATRIX(4,4) DOUBLE,
1: C BIT(6),

NOTE THE SEMI-COLON!

4 STRUCTURE TERMINALS ALL AT LEVEL 1.

EXAMPLE 2

STRUCTURE RR:

1: Z1, A MINOR STRUCTURE HAS NO ATTRIBUTES
2: I INTEGER,
2: J INTEGER,

1: Z2,
2: I INTEGER,
2: J INTEGER,

2 MINOR STRUCTURES
OR FORKS
Example 3

STRUCTURE SS:

1 Z1, ————> F
2 S1, ————> T
2 S2, ————> T
2 ZZ1, ————> F
3 ZZ2, ————> F
4 ZZ3, ————> F
5 I INTEGER, ————> T

2 S3, ————> T
1 ZZ4, ————> F
2 S4, ————> T
2 S5 DOUBLE

T TERMINAL
F FORK (MINOR STRUCTURE)

NOTE (1): FORK (MINOR STRUCTURE) IS EASILY SEEN BY OBSERVING THAT THE LEVEL # OF THE NEXT NODE GOES UP BY 1!
STRUCTURES (CONT.)

AS A TREE, EXAMPLE 3 TAKES THE FORM:

ROOT OR TEMPLATE NAME

LEVEL

O = FORK (MINOR STRUCTURE)  O = LEAF (TERMINAL)
STRUCTURES (CON'T.)

NOTE (2): IF A TERMINAL HAS NO ATTRIBUTES IT IS CONSIDERED TO BE A SCALAR (JUST LIKE DECLARES).

NOTE (3): IF AN INTEGER, SCALAR, VECTOR, OR MATRIX TERMINAL HAS NO PRECISION SPECIFICATION, IT IS ASSUMED TO BE SINGLE (JUST LIKE DECLARES).

THE FOLLOWING DATA TYPES ARE LEGAL IN A STRUCTURE TEMPLATE:

<table>
<thead>
<tr>
<th>INTEGER</th>
<th>BOOLEAN</th>
</tr>
</thead>
<tbody>
<tr>
<td>SCALAR</td>
<td>BIT</td>
</tr>
<tr>
<td>VECTOR</td>
<td>CHARACTER</td>
</tr>
<tr>
<td>MATRIX</td>
<td>STRUCTURE</td>
</tr>
</tbody>
</table>

ALSO, NAME VARIABLES OF THESE TYPES ARE LEGAL, PLUS:

NAME EVENT
NAME PROGRAM
NAME TASK

WILL BE DISCUSSED LATER.
STRUCTURES (CON'T.)

RESTRICTIONS:

(1) TEMPLATE NAMES MUST BE UNIQUE WITHIN THE NAME SCOPE.

(2) NAMES WITHIN THE TEMPLATE DO NOT NECESSARILY HAVE TO BE UNIQUE.

(3) INITIAL/CONSTANT LISTS CANNOT BE PRESENT IN A TEMPLATE.

(4) STATIC/AUTOMATIC CANNOT BE SPECIFIED IN A TEMPLATE.

(5) CHARACTER(*), ARRAY(*), AND STRUCTURE(*) ARE ALSO PRECLUDED.

(6) A STRUCTURE TERMINAL CANNOT HAVE COPIES.

STRUCTURE TEMPLATES ARE ESSENTIALLY DECLARE STATEMENTS -- AS A RESULT THEY MUST BE PLACED WITHIN A DECLARE GROUP (AND PRIOR, OF COURSE, TO ANY STRUCTURE DECLARATIONS REQUIRING THE TEMPLATE).
STRUCTURES (CON’T.)

B) STRUCTURE DECLARATION:

SIMPLE STRUCTURE

DECLARE name a-STRUCTURE;

MULTI-COPY STRUCTURE

DECLARE name α-STRUCTURE(N);

WHERE name IS THE STRUCTURE (OR MAJOR STRUCTURE) NAME AND α IS THE NAME OF A PREVIOUSLY-DEFINED TEMPLATE. N IS THE NUMBER OF COPIES DESIRED.

2 ≤ N ≤ 32,767

EXAMPLE

STRUCTURE Q:

1 QA SCALAR, or just 1 QA

1 QB INTEGER,

1 QC BOOLEAN; HYPHEN

DECLARE Q Q-STRUCTURE(3);
DECLARE R Q-STRUCTURE;
DECLARE I INTEGER, S SCALAR, W Q-STRUCTURE;
EXAMPLE

CPL: EXTERNAL COMPOOL;
    . . .
 STRUCTURE Q:
      1 A,
      1 B DOUBLE,
      1 C INTEGER,
    . . .
 CLOSE CPL;

PROG: PROGRAM;
    . . .
 STRUCTURE R:
      1 I INTEGER DOUBLE,
      1 V ARRAY(6,5) VECTOR(4),
      1 D CHARACTER(80),
    . . .
 DECLARE R R-STRUCTURE;
 DECLARE $ Q-STRUCTURE(10);
 DECLARE T T-STRUCTURE;
 STRUCTURE T:
      1 E INTEGER;

ILLEGAL - TEMPLATE MUST PRECEDE STRUCTURE
STRUCTURES (CON'T.)

(©) STRUCTURE INITIALIZATION

INITIAL/CONSTANT LISTS ARE ATTACHED TO THE STRUCTURE DECLARATION -- NOT TO A TEMPLATE OR ITS INNERDS.

TERMINALS ARE INITIALIZED BY THE ORDER OF APPEARANCE IN THE TEMPLATE.

- STRUCTURES WITH NO COPIES -

STRUCTURE 0:

1 A,
2 I INTEGER,
2 S SCALAR,
1 B,
2 J INTEGER,
2 K INTEGER,
2 L INTEGER,
2 M ARRAY(10) INTEGER;

DECLARE Z Q-STRUCTURE INITIAL(4, 6.95, 3#2, *)

PARTIAL INITIALIZATION

I = 4  S = 6.95  J = K = L = 2
M IS UNINITIALIZED
STRUCTURES (CON'T.)

- STRUCTURES WITH COPIES -

TWO CHOICES:

- PUT ENOUGH LITERALS IN THE INITIAL/CONSTANT LIST TO
  INITIALIZE THE FIRST COPY -- ALL REMAINING COPIES
  WILL BE INITIALIZED LIKE THE FIRST AUTOMATICALLY.

- PUT ENOUGH LITERALS IN THE INITIAL/CONSTANT FOR ALL
  TERMINALS OF ALL COPIES. EACH COPY IS FULLY
  INITIALIZED BEFORE PASSING TO THE NEXT.

EXAMPLES

STRUCTURE Q:
1 I INTEGER,
1 S SCALAR;

(1) DECLARE Q Q-STRUCTURE(5) INITIAL(6, 81.45);
(2) DECLARE R Q-STRUCTURE(3) INITIAL(1, 84.0, 2, -16.4, 4, 350.8);
(3) DECLARE SS Q-STRUCTURE(20000) INITIAL(5, -40.6, 3, 119.2, *)

COPY 1   COPY 2   COPIES 3-20,000
UNINITIALIZED
STRUCTURES (CON’T.)

INITIALIZATION EXAMPLE

STRUCTURE Q;
  1 QV VECTOR(3),
  1 QM,
  2 QI INTEGER,
  2 QC CHARACTER(80);
DECLARE Z1 Q-STRUCTURE INITIAL(1.5, 2.5, 3.5, -2, 'ALPHA');
DECLARE Z2 Q-STRUCTURE(2) INITIAL(4.5, 5.5, 6.5, -4, 'BETA');
DECLARE Z3 Q-STRUCTURE(2) INITIAL(3#1.5, 1, 'GAMMA',
                                    3#2.5, 2, 'DELTA');
STRUCTURES (CON'T.)

Both copies identically initialized.

Q1 = -4  QC = 'BETA'
QV = [4.5  5.5  6.5]

Q1 = -4  QC = 'BETA'
QV = [4.5  5.5  6.5]

Q1 = 1   QC = 'GAMMA'
QV = [1.5  1.5  1.5]

Q1 = 2   QC = 'DELTA'
QV = [2.5  2.5  2.5]
QUALIFIED REFERENCES

STRUCTURE Q:
1 A,
2 1 INTEGER,
2 B,
3 J INTEGER,
3 S1 SCALAR,
3 C,
4 M MATRIX(4,4),
4 V VECTOR(4),
3 S2 SCALAR,
2 D,
3 CHARX CHARACTER(10),
3 BB BOOLEAN,
2 K INTEGER,
1 MM X-STRUCTURE,
1 VV VECTOR(3);
DECLARE R Q-STRUCTURE INITIAL(...);

(1) ENTIRE STRUCTURE IS REFERENCED VIA NAME ‘R’, I.E., THE MAJOR STRUCTURE NAME.
(2) A TERMINAL IS REFERENCED BY: R, n1, n2, ..., nx, terminal_name

MINOR STRUCTURES ENCOUNTERED
<table>
<thead>
<tr>
<th>Reference</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>R</td>
<td>References the whole structure</td>
</tr>
<tr>
<td>R.A</td>
<td>Sub-structures (minor structures) of varying complexity</td>
</tr>
<tr>
<td>R.A.B</td>
<td></td>
</tr>
<tr>
<td>R.A.B.C</td>
<td></td>
</tr>
<tr>
<td>R.A.D</td>
<td></td>
</tr>
</tbody>
</table>

REFERENCE: THE WHOLE STRUCTURE

R.A.B

REFERENCE SUB-STRUCTURES (MINOR STRUCTURES) OF VARYING COMPLEXITY

R.A.D
STRUCTURES (CON'T.)

STRUCTURE NESTING:

1 Nt PROG:
2 Nt PROGRAM;
3 Nt STRUCTURE A:
4 Nt       1 AI INTEGER,
5 Nt       1 A1,
6 Nt       2 AC CHARACTER(80),
7 Nt       2 AB BOOLEAN;
3 Nt STRUCTURE B:
4 Nt       1 BS SCALAR,
5 Nt       1 B1,
6 Nt       2 BV VECTOR(3),
7 Nt       2 BA A-STRUCTURE;  "CANNOT HAVE COPIES!"
3 Nt STRUCTURE C:
4 Nt       1 BS SCALAR,
5 Nt       1 B1,
6 Nt       2 BV VECTOR(3),
7 Nt       2 BA,
8 Nt       3 AI INTEGER,
9 Nt       3 A1,
10 Nt       4 AC CHARACTER(80),
11 Nt       4 AB BOOLEAN;
5 Nt DECLARE Q A-STRUCTURE;
6 Nt DECLARE R B-STRUCTURE;
7 Nt DECLARE S C-STRUCTURE;
8 Nt DECLARE T C-STRUCTURE;
9 Nt DECLARE U A-STRUCTURE;

C++
STRUCTURES (CON'T.)

Et + +
10 Ht R = S,

**** AV4. ERROR #1 OF SEVERITY 1. ****
**** TREE ORGANIZATIONS DO NOT MATCH ACROSS ASSIGNMENT

C↑ + +

11 Ht S = T;

C↑ + +

12 H↑ R. B1. BA = Q;

C↑ + +

13 H↑ S. B1. BA = U;

14 M↑ CLOSE;
THE QUALIFIED REFERENCE SYSTEM MUST BE USED FOR QUALIFIED STRUCTURES, BUT IS OPTIONAL FOR UNQUALIFIED STRUCTURES.

DEFN. A STRUCTURE IS UNQUALIFIED IF IT HAS THE SAME NAME AS ITS TEMPLATE -- OTHERWISE, IT IS A QUALIFIED STRUCTURE.

EXAMPLE

STRUCTURE Q:
  1 I INTEGER,
  1 S SCALAR,
  1 M MATRIX DOUBLE,
  1 T;
DECLARE Q Q-STRUCTURE;
DECLARE R Q-STRUCTURE;
THEN Q IS UNQUALIFIED.
THEN R IS QUALIFIED.
STRUCTURES (CON'T.)

IF A STRUCTURE IS UNQUALIFIED, THEN RATHER THAN USE THE QUALIFIED REFERENCE MECHANISM, TERMINALS AND MINOR STRUCTURES CAN SIMPLY BE REFERRED TO BY THEIR NAMES.

IN THE PRECEDING EXAMPLE:

\[
\begin{align*}
\text{Q} & \quad \text{I OR Q,I} \\
\text{S} & \quad \text{S OR Q.S} \\
\text{M} & \quad \text{M OR Q,M} \\
\text{T} & \quad \text{T OR Q,T} \\
\text{R} & \quad \text{R,I} \\
\text{S} & \quad \text{R,S} \\
\text{M} & \quad \text{R,M} \\
\text{T} & \quad \text{R,T}
\end{align*}
\]

SHORT FORM (SINCE Q IS UNQUALIFIED)

UNQUALIFIED STRUCTURES ARE NICE ... BUT NOT EVERY STRUCTURE CAN BE UNQUALIFIED.

NOTE: IN TERMS OF CPU OR CORE, THERE IS NO DIFFERENCE BETWEEN A QUALIFIED OR AN UNQUALIFIED STRUCTURE.
RULES FOR UNQUALIFIED STRUCTURES:

(1) BOTH THE TEMPLATE AND THE STRUCTURE MUST BELONG TO THE SAME NAME-SCOPE:

**LEGAL**

P: PROGRAM;

STRUCTURE Q;

1 A,

1 B,

. . .

DECLARE Q Q-STRUCTURE;

. . .

CLOSE P;

**ILLEGAL**

CPL: COMPOOL;

STRUCTURE Q:

1 A,

1 B,

. . .

DECLARE Q Q-STRUCTURE;

CLOSE CPL;

P: PROGRAM;

DECLARE Q Q-STRUCTURE;

(2) THE TEMPLATE CAN CONTAIN NO NESTED STRUCTURES, I.E., STRUCTURE TERMINALS,

FOR EXAMPLE, THE FOLLOWING IS ILLEGAL:

**LEGAL**

STRUCTURE Q:

1 A,

1 I INTEGER,

1 M R-STRUCTURE;

DECLARE Q Q-STRUCTURE;

**ILLEGAL**

CANNOT HAVE NESTED STRUCTURE

DECLARE Q Q-STRUCTURE(4);
STRUCTURES (CON'T.)

(3) ALL NAMES (IDENTIFIERS) WITHIN THE TEMPLATE MUST BE UNIQUE IN THE NAME-SCOPE.

KEY IDEA: IDENTIFIERS WITHIN A TEMPLATE ARE NORMALLY INVISIBLE TO THE OUTSIDE ENVIRONMENT. BY MAKING THE STRUCTURE UNQUALIFIED, THESE HIDDEN NAMES SUDDENLY BECOME VISIBLE.

EXAMPLE

DECLARE I;
STRUCTURE Q:
  1 I INTEGER,
  1 S,
  1 M MATRIX;
SO FAR ... NO PROBLEM (THE TWO I's CANNOT "SEE" EACH OTHER)

DECLARE R Q-STRUCTURE(10);
STILL NO PROBLEM...

DECLARE Q Q-STRUCTURE;
NOW WE WILL HAVE A MULTIPLY-DEFINED SYMBOL "I".
STRUCTURES (CON'T.)

IF THE STRUCTURE TEMPLATE IS NEVER INTENDED TO BE UNQUALIFIED, THEN DUPLICATION OF IDENTIFIERS CAN OCCUR WITHIN THE TEMPLATE AS LONG AS NO AMBIGUITY RESULTS WHEN USING THE QUALIFIED REFERENCE SYSTEM.

EXAMPLE 1

STRUCTURE Q:
  1 Q1,
  2 QS,
  1 Q2,
  2 QS,

LEGAL DUPLICATE NAMES

DECLARE ZQ Q-STRUCTURE;

BECAUSE ZQ.Q1.QS CANNOT BE CONFUSED WITH ZQ.Q2.QS

NOTE THAT:

DECLARE Q Q-STRUCTURE

WOULD RESULT IN MULTIPLY-DEFINED QS.
EXAMPLE 2

STRUCTURE R:
1  R1,  --- ILLEGAL DUPLICATE NAMES
2  RS SCALAR,
1  R1 CHARACTER(80);
DECLARE ZR R-STRUCTURE;

WHAT IS ZR,R1?

SUMMARY OF NAME UNIQUENESS

(1) WITHIN THE SAME NAME-SCOPE, ALL TEMPLATE AND STRUCTURE NAMES
    MUST BE UNIQUE (UNLESS, OF COURSE, WE ARE DEFINING AN UNQUALIFIED
    STRUCTURE -- IN WHICH CASE A TEMPLATE NAME AND A MAJOR STRUCTURE
    NAME WILL MATCH)

(2) CLEARLY, FOR A GIVEN TEMPLATE, THERE CAN BE AT MOST ONE UNQUALIFIED
    STRUCTURE WITH THAT NAME.

(3) IF A TEMPLATE IS UNQUALIFIED, ALL IDENTIFIERS WITHIN IT MUST
    BE UNIQUE IN THE NAME-SCOPE.
(4) IF A TEMPLATE IS QUALIFIED, SOME DUPLICATION OF IDENTIFIERS CAN OCCUR IF NO AMBIGUITY RESULTS.

EXAMPLES

(1) DECLARE S;
    STRUCTURE S;
        1 ...

(2) STRUCTURE S:
    1 A,
    1 B;
    STRUCTURE S:
        1 A,
        1 B;

(3) STRUCTURE S:
    1 A,
    1 B;
    DECLARE S S-STRUCTURE;

(4) DECLARE M MATRIX;
    DECLARE M S-STRUCTURE;
(5) STRUCTURE S:
  1 A,
  2 I,
  2 J,
  1 B,
  2 I,
  2 J;

(6) DECLARE I INTEGER;
STRUCTURE S:
 1 I INTEGER,
 1 J INTEGER;

MISCELLANEOUS COMMENTS

(1) A TEMPLATE DECLARATION DOES NOT SET ASIDE CORE MEMORY -- ONLY THE STRUCTURE DECLARATION DOES THAT.

(2) QUALIFIED AND UNQUALIFIED STRUCTURES ARE EQUIVALENT IN TERMS OF CORE AND CPU.

(3) THERE IS NO INHERENT INEFFECTIVENESS IN COLLECTING RELATED DATA INTO A STRUCTURE.
In terms of core and CPU efficiency, we can write:

```
SIMPLE VARIABLE > ARRAY > STRUCTURE
```

This means that a simple variable generally might be more efficient than an array, and an array in turn might be more efficient than a structure. In many important cases, however, equality holds (more on this later...).
SUMMARY OF TERMINOLOGY

STRUCTURE Q:

\( (1 A, ) \)

\( (2 I, \text{INTEGER}, ) \)

\( 2 J \text{ INTEGER}, \)

\( 1 S \text{ SCALAR}; \)

LEVEL

STRUCTURE DECLARATIONS

DECLARE Q Q-STRUCTURE;
DECLARE R Q-STRUCTURE
INITIAL(6, 9, 15.4);

Q = UNQUALIFIED STRUCTURE
R = QUALIFIED STRUCTURE

\( Q \rightarrow R \)

\( A \rightarrow R.A \)

\( I \rightarrow R.A.I \)

\( J \rightarrow R.A.J \)

\( S \rightarrow R.S \)

Q = QUALIFIED NAMES
STRUCTURES (CON’T.)

A. STRUCTURE SUBSCRIPTING

- COPY SUBSCRIPTING ONLY

Suppose we have a multi-copy structure:

DECLARE Q Q-STRUCTURE(L);

(1) To select the Ith copy

Q$1 OR Q$(I)

Where I is an integer expression (scalar if you want) that had better lie in the range

1 ≤ I ≤ L

at run-time.

Note: Semicolon is only mandatory when terminal subscripting is to be done also.
STRUCTURES (CON'T.)

(2) TO SELECT A SUBSET OF $I$ COPIES STARTING WITH THE $J^{TH}$:

$$Q^S(I \text{ AT } J) \text{ OR } Q^S(I \text{ AT } J)$$

WHERE $I$ IS AN INTEGER LITERAL (I.E., VALUE KNOWN AT COMPILE-TIME) AND $J$ IS AN INTEGER EXPRESSION WITH:

$$1 \leq J \leq L - I + 1$$

(3) TO SELECT A SUBSET OF COPIES STARTING FROM THE $I^{TH}$ AND ENDING WITH THE $J^{TH}$:

$$Q^S(I \text{ TO } J) \text{ OR } Q^S(I \text{ TO } J)$$

WHERE BOTH $I$ AND $J$ ARE INTEGER LITERALS AND:

$$1 \leq I \leq J \leq L$$
COPY SUBSCRIPTING EXAMPLES:

GIVEN

STRUCTURE Q:
1 Q1 INTEGER,
1 Q1,
2 Q$\text{S}$ SCALAR,
2 QC CHARACTER(80);

DECLARE ZQ Q-STRUCTURE(3);

WITH THE FOLLOWING VALUES:

![Diagram showing the structure with values for each Q1 and QS. Q1 values include 1, 2, and 3, and QS values are 1.5, 2.5, and 3.5. QC values for each Q1 are 'A', 'B', and 'C'.]
THEN ZQ₂, SELECTS COPY 2 WITH VALUES:

```
    o
   / \
  zQ  copy 2
    / \      /
   /   \  Q₁  Q₁
  /     /  Q₈  Q₈
 /     /   /  |
/     /   /   Q₁₀
```

Q₁ ≡ 2
Q₈ ≡ 2.5
Q₁₀ ≡ 'B'
STRUCTURES (CON'T.)

GIVEN

STRUCTURE Q:
1 Q1 INTEGER,
1 Q1,
2 QS SCALAR,
2 QC CHARACTER(80);

DECLARE ZQ Q-STRUCTURE(3);

WITH THE FOLLOWING VALUES:

\[ ZQ \]

\[ \begin{array}{c}
\text{copy 1} \\
\text{copy 2} \\
\text{copy 3}
\end{array} \]

\[ \begin{array}{c}
Q1 \equiv 1 \\
Q1 \equiv 2 \\
Q1 \equiv 3
\end{array} \]

\[ \begin{array}{c}
QS \equiv 1.5, QC \equiv 'A' \\
QS \equiv 2.5, QC \equiv 'B' \\
QS \equiv 3.5, QC \equiv 'C'
\end{array} \]
STRUCTURES (CON'T.)

ZQ, Q1 TO 2, SELECTS COPIES 1 AND 2 OF THE SUB-TREE UNDER Q1

QS ≥ 1.5  QC = 'A'
QS ≥ 2.5  QC = 'B'
STRUCTURES (CON'T.)

GIVEN

STRUCTURE Q:
1 Q1 INTEGER,
1 Q1,
2 QS SCALAR,
2 QC CHARACTER(80);

DECLARE ZQ Q-STRUCTURE(3);

WITH THE FOLLOWING VALUES:

Diagram:

- ZQ
  - Q1 = 1
    - QS = 1.5
    - QC = 'A'
  - Q1 = 2
    - QS = 2.5
    - QC = 'B'
  - Q1 = 3
    - QS = 3.5
    - QC = 'C'
STRUCTURES (CON'T.)

QZ.Q13 SELECTS:

\[ q_s \equiv 3.5 \quad q_c \equiv 'c' \]
• SUBSCRIPTING OF STRUCTURE TERMINALS

(1) IF THE STRUCTURE HAS NO COPIES THEN THE TERMINAL IS
SUBSCRIPTED EXACTLY AS THOUGH IT WERE NOT IN A
STRUCTURE.

STRUCTURE Q:
  1 B1 BIT(8),
  1 M MATRIX(4,4),
  1 C CHARACTER(8),
  1 V VECTOR;
DECLARE Q Q-STRUCTURE;

THEN:

<table>
<thead>
<tr>
<th></th>
<th>B1$4</th>
<th>Q.B1$4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>B1$(3 AT 3)</td>
<td>Q.B1$(3 AT 3)</td>
</tr>
<tr>
<td></td>
<td>M$(2,4)</td>
<td>Q.M$(2,4)</td>
</tr>
<tr>
<td></td>
<td>M$(*,3)</td>
<td>Q.M$(*,3)</td>
</tr>
<tr>
<td></td>
<td>C$5</td>
<td>Q.C$5</td>
</tr>
<tr>
<td></td>
<td>C$(4 TO 6)</td>
<td>Q.C$(4 TO 6)</td>
</tr>
<tr>
<td></td>
<td>V$3</td>
<td>Q.V$3</td>
</tr>
<tr>
<td></td>
<td>V$(2 AT 1)</td>
<td>Q.V$(2 AT 1)</td>
</tr>
</tbody>
</table>
STRUCTURES (CON'T.)

STRUCTURE Q:
1 I ARRAY(10) INTEGER,
1 B ARRAY(6) BOOLEAN,
1 M ARRAY(2,4,6) MATRIX,
1 B4 ARRAY(8) BIT(4);
DECLARE R Q-STRUCTURE;

THEN:
R.I$(5)   R.I$(2 AT 3)      ETC.
R.B$(3:)  R.B$(3 TO 5:)     ETC.
R.M$(*,*,*: 2,3)
R.M$(1,3,5; *,1)
R.M$(2,*,6;)
R.M$(*,*,3: 1 TO 2, 2 AT 1)
R.B4$(5:)
R.B4$(5;3)
R.B4$(2 TO 5: 3 AT 1)

THEREFORE, THERE IS NOTHING NEW TO SAY ABOUT SUBSCRIPTING
IF THE STRUCTURE HAS NO COPIES.
(2) IF THE STRUCTURE HAS COPIES THEN WE HAVE THREE MODES OF TERMINAL SUBSCRIPTING:

A  STRUCTURE (COPY) SUBSCRIPTING ONLY
=> NEED TRAILING ";" UNLESS TERMINAL IS AN UNARRAYED INTEGER OR SCALAR.

EXAMPLE

STRUCTURE Q:
  1 I INTEGER,
  1 II ARRAY(5) INTEGER,
  1 B BOOLEAN,
  1 M MATRIX;
DECLARE Q Q-STRUCTURE(50);

THEN:
  I$5        GETS I FROM 5TH COPY
  II$(5;)    GETS II ARRAY FROM 5TH COPY
  II$(10 AT 4;) BEHAVES LIKE A 2-DIMENSIONAL ARRAY

SEMI-COLON NEEDED ———7——"__
BECAUSE BOOLEAN = BIT(1) —- B$(3;) GETS BOOLEAN FROM 3RD COPY
M$(6;)         GETS MATRIX FROM 6TH COPY
STRUCTURES (CON'T.)

B) STRUCTURE AND TERMINAL SUBSCRIPTING

→ COPY SUBSCRIPT; TERMINAL SUBSCRIPTS

EXAMPLE

STRUCTURE Q:
  1 II ARRAY(2,3) INTEGER,
  1 BB ARRAY(6) BIT(9),
  1 MM ARRAY(5,10,15) MATRIX(4,5),
  1 N ARRAY(10) INTEGER,
  1 I INTEGER;
DECLARE R Q-STRUCTURE (20000);
THEN:
  R. II$(4; 2, 2)
  R. II$(100 AT 307; 2, *)
  R. BB$(60; 4; 5 TO 8)
  R. BB$(31; 5;)
  R. MM$(5 AT 9; 5, *, 7: 3, *)
  R. MM$(27; *, *, *: *, 5)
  R. N$(6; 10)
  R. I$(50)
EXAMPLE

STRUCTURE Q:
1 II ARRAY(10) INTEGER,
1 M ARRAY(10,5,6) MATRIX;

STRUCTURE QQ:
1 Z1 Q-STRUCTURE,
1 J ARRAY(5) BIT(16),
1 A,
2 BOOL ARRAY(10) BOOLEAN,
2 C ARRAY(5) CHARACTER(80);
DECLARE P QQ-STRUCTURE(20);

THEN:
P.Z1.II$(*; 6 TO 10)
P.Z1.M$(*; 1 TO 3, 2 TO 5, 4: *,3)
P.Z1.M$(*; *,*,*: 2,3)
P.J$(*; 4 AT 1: 3 AT 1)
P.A.BOOL$(*; 6:)
P.A.C$(*; 3 TO 5: 10 AT 24)
EXAMPLES OF TERMINAL SUBSCRIPTING

GIVEN

1. STRUCTURE Q:
   1. QV VECTOR(3),
   2. Q1,
   3. QB ARRAY(2) BIT(4),
   4. QM MATRIX(3,3);

DECLARE ZQ Q-STRUCTURE;

WITH THE FOLLOWING VALUES:

\[
\begin{bmatrix}
1 & 2 & 3 \\
2 & 3 & 1 \\
3 & 1 & 2
\end{bmatrix}
\]

\[
\begin{bmatrix}
1 & 2 & 3 \\
4 & 5 & 6 \\
7 & 8 & 9
\end{bmatrix}
\]

\[
\begin{bmatrix}
1100_2, 1011_2
\end{bmatrix}
\]
STRUCTURES (CON'T.)

THEN:

\[ ZQ.QV_1 = 1 \]

\[ ZQ.Q1.QB_{1:3} = (0_2 1_2) \]

AND

\[ ZQ.Q1.QM_{2 \text{ TO } 3, 2 \text{ TO } 3} = \begin{bmatrix} 5 & 6 \\ 8 & 9 \end{bmatrix} \]
FURTHER, GIVEN

STRUCTURE \( Q \):
1 QV VECTOR(3),
1 Q1,
2 QB ARRAY(2) BIT(4),
2 QM MATRIX(3,3)
::
DECLARE YQ Q-STRUCTURE(3);
WITH THE FOLLOWING VALUES:

\[
\begin{align*}
QV & \equiv \begin{bmatrix} 1 \\ 2 \\ 3 \end{bmatrix} \\
QM & \equiv \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{bmatrix} \\
QB & \equiv (1001_2 1111_2) \\
QV & \equiv \begin{bmatrix} 4 \\ 5 \\ 6 \end{bmatrix} \\
QM & \equiv \begin{bmatrix} 8 & 7 \\ 6 & 5 & 4 \\ 3 & 2 & 1 \end{bmatrix} \\
QB & \equiv (1101_2 1011_2) \\
QV & \equiv \begin{bmatrix} 7 \\ 8 \\ 9 \end{bmatrix} \\
QM & \equiv \begin{bmatrix} 1 & 7 & 9 \\ 4 & 2 & 8 \\ 6 & 5 & 3 \end{bmatrix} \\
QB & \equiv (1001_2 0110_2)
\end{align*}
\]
then

\[ YQ.QV_{*,3} \equiv (3, 6, 9) \]

result is scalar type

\[ YQ.Q1.QM_{2,1,1} \text{ TO } 2 \equiv (11_2, 10_2) \]

array property unmodified

\[ YQ.Q1.QM_{2,3,3} \equiv 1 \]
STRUCTURES (CON'T.)

TREE (TEMPLATE) EQUIVALENCE

TWO STRUCTURES (OR MINOR STRUCTURES) ARE EQUIVALENT (TREE-EQUIVALENT) IF:

(1) THE ACTUAL "SHAPES" OF THE TREES (ORGANIZATION OF NODES) ARE THE SAME, AND
(2) CORRESPONDING NODES AGREE EXACTLY IN ATTRIBUTES.

A) OBVIOUSLY, TWO STRUCTURES ARE EQUIVALENT IF THEY USE THE SAME TEMPLATE:

| STRUCTURE Q: |
| 1 Q INTEGER, |
| 1 QI, |
| 2 QS SCALAR, |
| 2 QC CHARACTER(80); |
| DECLARE ZQ1 Q-STRUCTURE, |
| ZQ2 Q-STRUCTURE(20); |

ZQ1 AND ZQ2 ARE TREE-EQUIVALENT, (NOTWITHSTANDING THE MISMATCH IN NUMBER OF COPIES).
STRUCTURES (CONT.)

1. STRUCTURES ARE ALSO EQUIVALENT IF THEIR TEMPLATES DIFFER ONLY IN THE IDENTIFIERS:

   STRUCTURE Q:
   1. Q1 INTEGER,
      1. Q1
      2. QS SCALAR,
      2. QC CHARACTER(80);
   DECLARE ZQ Q-STRUCTURE;

   STRUCTURE R:
   1. RI INTEGER,
      1. RI
      2. RS SCALAR,
      2. RC CHARACTER(80);
   DECLARE ZR R-STRUCTURE;

   The tree shapes of ZR and ZQ are the same:

   ![Tree Diagram]

---
MINOR STRUCTURES CAN BE EQUIVALENT EVEN IF THE MAJOR STRUCTURES ARE NOT.

STRUCTURE Q:
1 QI INTEGER,
1 Q1,
2 QS SCALAR,
2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;

STRUCTURE R:
1 RS SCALAR,
1 RC CHARACTER(80);
DECLARE ZR R-STRUCTURE;

The tree shapes of ZQ and ZR clearly are not the same. However, the tree shapes of ZQ.Q1 and ZR are the same:
ADDITIONALLY, HOWEVER, CORRESPONDING NODES MUST AGREE IN ALL ATTRIBUTES, E.G. DATA TYPE, PRECISION, ARRAYNESS ....

<table>
<thead>
<tr>
<th>TYPE</th>
<th>MATCHING REQUIREMENTS</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIT-STRING</td>
<td>number of bits (BOOLEAN is equivalent to BIT(1))</td>
</tr>
<tr>
<td>CHARACTER</td>
<td>maximum declared length</td>
</tr>
<tr>
<td>INTEGER</td>
<td>precision</td>
</tr>
<tr>
<td>SCALAR</td>
<td>precision</td>
</tr>
<tr>
<td>VECTOR</td>
<td>precision, length</td>
</tr>
<tr>
<td>MATRIX</td>
<td>precision, row and column dimensions</td>
</tr>
<tr>
<td>STRUCTURE</td>
<td>specified structure template</td>
</tr>
</tbody>
</table>

Examples:

STRUCTURE Q:
1.QI INTEGER, 1 Q1,
2 QM MATRIX(3,3),
2 QC CHARACTER(80); DECLARE EQ Q-STRUCTURE;

ZQ AND ZR ARE NOT TREE EQUIVALENT

BUT, ZQ.Q1 AND ZR.R1 ARE TREE EQUIVALENT.
STRUCTURES (CON'T.)

FOR TREE EQUIVALENCE, CORRESPONDING TERMINALS OF STRUCTURE TYPE MUST IN TURN USE TREE-EQUIVALENT TEMPLATES:

EXAMPLE

STRUCTURE P:
  1 S SCALAR,
  1 T SCALAR DOUBLE;
STRUCTURE Q:
  1 U SCALAR,
  1 V SCALAR DOUBLE;
STRUCTURE PP:
  1 A,
  1 B P-STRUCTURE;
STRUCTURE QQ:
  1 F,
  1 G Q-STRUCTURE;
DECLARE X PP-STRUCTURE;
DECLARE Y QQ-STRUCTURE;

THEN X IS EQUIVALENT TO Y.
A NESTED STRUCTURE IS NOT THE SAME AS A SIMILAR MINOR STRUCTURE.

FUNCTIONALLY THE SAME, BUT NOT EQUIVALENT

ZS AND ZR ARE NOT EQUIVALENT!

IF WE REDEFINE S WE ARE OK:

```
STRUCTURE S:
  1 SI INTEGER,
  1 SQ Q-STRUCTURE;
```
STRUCTURE ASSIGNMENTS

SYMBOLIC FORM: \( L = R \)

1. \( L \) is receiving structure data item with possible structure subscripting.

2. \( R \) is either a (possibly subscripted) structure data item or a structure function.

3. If \( L \) and \( R \) are major or minor structures, they must be tree-equivalent.
EXAMPLES

Given:

```
STRUCTURE Q:
  1 Q1 INTEGER,
  2 QS SCALAR,
  2 QC CHARACTER(80);
DECLARE ZQ1 Q-STRUCTURE;
DECLARE ZQ2 Q-STRUCTURE(2);
```

where ZQ2 has the values:

```
Q1 ≡ 1
QS ≡ 4.5
QC ≡ 'A'
```

```
Q1 ≡ 2
QS ≡ 2.5
QC ≡ 'B'
```
then

\[ zQ_1 = zQ_2 \]
\[ s \]

results in \( ZQ_1 \) having the values:

\[ Q_1 \equiv 2 \]
\[ Q_S \equiv 2.5 \]
\[ Q_C \equiv 'n' \]
Given:

\[
\text{STRUCTURE } Q:
\begin{align*}
&1 \text{ QI INTEGER,} \\
&1 \text{ Q1,} \\
&2 \text{ QS SCALAR,} \\
&2 \text{ QC CHARACTER(80);} \\
\text{DECLARE } \text{ ZQ1 Q-STRUCTURE;} \\
\text{DECLARE } \text{ ZQ2 Q-STRUCTURE(2);} \\
\end{align*}
\]

where ZQ2 has the values:

\[
\begin{align*}
\text{QI } &\equiv 1 \\
\text{Q1 } &\equiv 1 \\
\text{QS } &\equiv 4.5 \\
\text{QC } &\equiv 'A' \\
\text{QI } &\equiv 2 \\
\text{Q1 } &\equiv 2 \\
\text{QS } &\equiv 2.5 \\
\text{QC } &\equiv 'B'
\end{align*}
\]
and if then the following is executed

\[ S. Q_1.Q_1 = S. Q_2. Q_1; \]

the values of \( Q_1 \) are modified to:

\[ zQ_1 \]

\[ Q_1 \equiv 2 \]

\[ Q_S \equiv 4.5 \]

\[ Q_C \equiv 'A' \]
MULTIPLE ASSIGNMENTS

A NUMBER OF STRUCTURE DATA ITEMS (MAJOR OR MINOR STRUCTURES) MAY BE ASSIGNED THE SAME VALUES BY MEANS OF A MULTIPLE ASSIGNMENT:

\[ l^1, l^2, l^3 \ldots l^n = r^* \]

WHERE:

\[ l^1 \ldots l^n \]

ARE STRUCTURE DATA ITEMS, AND EACH \( l^i \) ARE TREE-EQUIVALENT TO \( r \).

NOTE: NO PARTICULAR ORDER OF ASSIGNMENT CAN BE GUARANTEED!

EXAMPLE:

STRUCTURE Q:

1 I INTEGER,
1 S,
DECLARE Q Q-STRUCTURE(100);

Q$3, Q$9, Q$10 = Q$47;

* \( r \) MAY ALSO BE A STRUCTURE FUNCTION.
STRUCTURES IN RELATIONAL EXPRESSIONS

1) STRUCTURE COMPARISONS CAN BE MADE IN RELATIONAL EXPRESSIONS, WHICH IN TURN MAY BE USED IN
   IF ...
   DO WHILE ...
   AND DO UNTIL ...
   CONSTRUCTS.

2) ONLY CLASS II COMPARATIVE OPERATIONS MAY BE EMPLOYED ON STRUCTURE DATA ITEMS, I.E., MAJOR AND MINOR STRUCTURES. CLASS II OPERATIONS ARE:

   =  ≠

E.G.,

   IF L = R THEN DO;
   DO WHILE L ≠ R;

RULES:
1) L AND R ARE EITHER STRUCTURE DATA ITEMS OR STRUCTURE FUNCTIONS.
2) L AND R MUST BE TREE-EQUIVALENT.
Three Structures in Relational Expressions (Con't.)

3. Two structures are equal if and only if all corresponding terminals have equal values.

Examples:

Structure Q:

1. I INTEGER,
2. S SCALAR;

DECLARE Q-STRUCTURE INITIAL(6, 18.0), L, R;

IF L = R THEN DO;
   TRUE
END;

L.I = 4;
DO WHILE L = R;
   TRUE
END;
STRUCTURE ARGUMENTS AND PARAMETERS

INPUT PARAMETERS (PROCEDURES/FUNCTIONS)

A STRUCTURE DATA ITEM (MAJOR STRUCTURE) CAN BE AN INPUT PARAMETER OF A PROCEDURE OR A FUNCTION. THE TEMPLATE DECLARATION FOR THE STRUCTURE MUST PRECEDE THE STRUCTURE DECLARATION.

THE CORRESPONDING INPUT ARGUMENT CAN BE A MAJOR STRUCTURE, MINOR STRUCTURE OR STRUCTURE FUNCTION -- AND MUST BE TREE-EQUIVALENT!

ASSIGN PARAMETERS (PROCEDURES ONLY)

AN ASSIGN ARGUMENT CAN BE A MAJOR OR MINOR STRUCTURE -- AND MUST BE TREE-EQUIVALENT TO THE CORRESPONDING PARAMETER.

TERMINALS AND MINOR STRUCTURES CAN Possess NO "COPYNESS" -- IF THEY ARE CONTAINED IN A MULTI-COPY STRUCTURE, THEN SUBSCRIPTING MUST BE USED TO ISOLATE A SINGLE COPY.

MAJOR STRUCTURES WITH COPIES CAN BE LEFT UNSUBSCRIPTED (FULL COPYNESS) -- OR MUST BE SUBSCRIPTED TO A SINGLE COPY!!
EXAMPLE 1: POSITION OF TEMPLATE

ANALYZE: PROCEDURE(S1) ASSIGN(S2);
STRUCTURE S1:
1 SI INTEGER,
1 SN,
2 SS SCALAR,
2 SC CHARACTER(80);
DECLARE S1 S-STRUCTURE,
S2 S-STRUCTURE;

executable code

CLOSE ANALYZE;

RULE:

PARAMETER STRUCTURE TEMPLATES BEFORE PARAMETER DECLARATIONS BEFORE LOCAL DATA DECLARATIONS.
EXAMPLE 2: TEMPLATE CAN BE IN AN OUTER SCOPE (E.G., COMPOOL)

CPL: EXTERNAL COMPOOL;
STRUCTURE Q:
  1 I INTEGER,
  1 V VECTOR,

  ...
CLOSE CPL;
COMSUB: PROCEDURE (STRUC);
DECLARE STRUC Q-STRUCTURE;

  ...
CLOSE COMSUB;
EXAMPLE 3: LEGAL AND ILLEGAL INVOCATIONS

STRUCTURE Q:
1 QI INTEGER,
1 Q1,
2 QS SCALAR,
2 QC CHARACTER(80);

STRUCTURE R:
1 RS SCALAR,
1 RC CHARACTER(80);

DECLARE ZQ Q-STRUCTURE,
ZR R-STRUCTURE,
YQ Q-STRUCTURE(10);

TREE: PROCEDURE(D1) ASSIGN(D2),
DECLARE D1 R-STRUCTURE,
D2 Q-STRUCTURE;

procedure body
THEN:

CALL TREE(ZR) ASSIGN(ZQ);
CALL TREE(ZR) ASSIGN(YQ);
S
CALL TREE(ZQ.Q1) ASSIGN(ZQ);
CALL TREE(ZR) ASSIGN(ZR);

Illegal - no tree-equivalence
EXAMPLE 4: LEGAL AND ILLEGAL INVOCATIONS

1 M→ TESTB:
2 M→ PROGRAM;
3 M→ STRUCTURE O:
4 M→ 1 A,
5 M→ 2 I INTEGER,
6 M→ 2 S SCALAR,
7 M→ 1 B,
8 M→ 2 J INTEGER,
9 M→ 2 T SCALAR;
10 M→ DECLARE O Q-STRUCTURE(10) INITIAL(1, 2, 3, 4);
11 M→ DECLARE R Q-STRUCTURE(5) INITIAL(5, 6, 7, 8);
12 M→ DECLARE P Q-STRUCTURE;

6 M→ PROC1:
7 M→ PROCEDURE ASSIGN(U, V, W);
8 M→ DECLARE U INTEGER;
9 M→ DECLARE V Q-STRUCTURE(5);
10 M→ STRUCTURE X:
11 M→ 1 K INTEGER,
12 M→ 1 SS SCALAR;
13 M→ DECLARE W X-STRUCTURE(10);
14 M→ CLOSE;

12 M→ CALL PROC1 ASSIGN(I, [R], [A]);
13 M→ 3
**** FS2  ERROR #1 OF SEVERITY 1. ****
**** THE STRUCTURE COPIES OF ASSIGN ARGUMENT A MUST BE SUBSCRIPTED
**** AWAY,
STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

E↑
13 CALL PROC ASIGN(R.A.I, (Q)), (A));
5↑
7↑
5 AT 2

**** SR3 ERROR #2 OF SEVERITY 1. *****
**** INDEX VALUE IN SUBSCRIPT OF R.A.I IS GREATER THAN THE LEGAL MAXIMUM
**** SY1 ERROR #3 OF SEVERITY 1. *****
**** SUBSCRIPTING OF Q IS ILLEGAL IN CONTEXT OF USE AS AN ASIGN ARGUMENT
**** FS2 ERROR #4 OF SEVERITY 1. *****
**** THE STRUCTURE COPIES OF ASIGN ARGUMENT Q MUST BE SUBSCRIPTED
**** AWAY
**** FS2 ERROR #5 OF SEVERITY 1. *****
**** THE STRUCTURE COPIES OF ASIGN ARGUMENT A MUST BE SUBSCRIPTED
**** AWAY
**** LAST ERROR WAS DETECTED AT STATEMENT 12. *****
STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

STRUCTURE(*)

JUST AS IN THE CASE OF 1-DIMENSIONAL ARRAYS, A STRUCTURE INPUT OR ASSIGN PARAMETER IS ALLOWED TO HAVE A VARIABLE NUMBER OF COPIES, I.E., THE NUMBER OF COPIES IS PASSED TO THE PROCEDURE IN THE STACK AT THE TIME OF INVOCATION.

THE BUILT-IN FUNCTION SIZE MAY LIKewise BE USED FOR SUCH UNKNOWN-COPYINESS STRUCTURES TO OBTAIN THE ACTUAL NUMBER OF COPIES.
EXAMPLE 1

DECLARE R Q-STRUCTURE(10);
DECLARE S Q-STRUCTURE(50);

PROCEDURE ASSIGN(D);
DECLARE D Q-STRUCTURE(*);

DO I = 1 TO SIZE(D);

END;

CLOSE PROC;

CALL PROC ASSIGN(R);
CALL PROC ASSIGN(S);
STRUCTURE ARGUMENTS AND PARAMETERS (CON'T.)

EXAMPLE 2: ARRAY PROCESSING FEATURE CAN LIKewise BE USED

STRUCTURE Q:
1 SCAL1,
1 SCAL2,
1 VECTX VECTOR;
DECLARE Q Q-STRUCTURE(50);
DECLARE R Q-STRUCTURE(20);

PROC: PROCEDURE ASSIGN(E);
DECLARE E Q-STRUCTURE(*);

E, SCAL1, E, SCAL2 = 0;  ARRAYED MULTIPLE ASSIGNMENT STMt

CALL PROC ASSIGN(Q);

CALL PROC ASSIGN(R);
STRUCTURE FUNCTIONS

HAL/S allows functions of structure type which may be substituted for declared structures in many of the previously defined structure operations.

FORM:

```
label: FUNCTION(l1, l2, ...) a-STRUCTURE;
      
      RETURN structure;
      
      CLOSE;
```

NOTES:

(1) A STRUCTURE FUNCTION CANNOT HAVE COPYNESS (REMEMBER THAT ORDINARY FUNCTIONS CANNOT HAVE AN ARRAY DECLARATION).

(2) THE TEMPLATE a MUST BE DEFINED IN AN OUTER SCOPE PRIOR TO THE FUNCTION HEADER.

(3) THE INPUT PARAMETER LIST IS OMITTED ENTIRELY IF THERE ARE NO PARAMETERS.
EXAMPLES

(A) ALPHA: FUNCTION B-STRUCTURE(6);  
   ILLEGAL

(B) ALPHA: FUNCTION B-STRUCTURE;
   STRUCTURE B:
   1 1 INTEGER,
   1 S;

(C) STRUCTURE Q:
   1 QI INTEGER,
   1 Qi,  
   QS SCALAR,
   QC CHARACTER(80);

   TREE: FUNCTION(I,J) Q-STRUCTURE;

   function body

   CLOSE TREE;
STRUCTURE FUNCTIONS (CON'T.)

AS IS THE CASE WITH ALL FUNCTIONS, A STRUCTURE FUNCTION MUST ACTUALLY RETURN A STRUCTURE OF THE REQUISITE TYPE:

```plaintext
STRUCTURE S:
  1 SS SCALAR,
  1 SC CHARACTER(80);

STRUCTURE Q:
  1 QI INTEGER,
  1 Q1 S-STRUCTURE;

TREE: FUNCTION(D1) S-STRUCTURE;
    DECLARE D1 Q-STRUCTURE;
    RETURN D1.Q1;

RETURN D1;

CLOSE TREE;
```

**Illegal - lack of**

**tree-equivalence**

**NOTE:** REMEMBER THAT AGGREGATE DATA (VECTORS, MATRICES, CHARACTER STRINGS, ARRAYS, AND STRUCTURES) ARE NEVER PHYSICALLY PASSED TO OR FROM A PROCEDURE OR FUNCTION. AGGREGATES ARE ALWAYS PASSED BY NAME (REFERENCE) -- NEVER BY VALUE!
A structure function is invoked by employing its name in a reference context. It should be clear, however, that a structure function is not really a structure -- i.e., minor structures or terminals of it cannot be referenced.

```
structure Q:
  1 QI INTEGER,
  1 Q1,
  2 QS SCALAR,
  2 QC CHARACTER(80);
DECLARE ZQ Q-STRUCTURE;
TREE: function Q-STRUCTURE;
{ function body

CLOSE TREE;

ZQ = TREE;  
ZQ.Q1 = TREE.Q1;
```

Also,
```
IF ZQ = TREE THEN DO;

ZQ = TREE;  
ZQ.Q1 = TREE.Q1;
```

LEGAL INVOCATION
ILLEGAL INVOCATION
ARRAY PROCESSING

EXAMPLE

STRUCTURE Q:

1 I INTEGER,
1 A ARRAY(4,9) SCALAR,
1 M ARRAY(6,10,4) MATRIX,
1 C ARRAY(6) CHARACTER(9);
DECLARE Q Q-STRUCTURE(16);
DECLARE J ARRAY(20) INTEGER;
DECLARE B ARRAY(16,4) SCALAR;
DECLARE N ARRAY(16,10,4) MATRIX;
DECLARE D ARRAY(6) CHARACTER(9);
**ARRAY PROCESSING (CON'T.)**

<table>
<thead>
<tr>
<th>a</th>
<th>TYPE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q +</td>
<td>{1:16} MAJOR STRUC</td>
</tr>
<tr>
<td>I +</td>
<td>{1:16} INTEGER</td>
</tr>
<tr>
<td>A +</td>
<td>{3:16,4,9} SCALAR</td>
</tr>
<tr>
<td>M +</td>
<td>{4:16,6,10,4} MATRIX</td>
</tr>
<tr>
<td>C +</td>
<td>{2:16,6} CHARACTER</td>
</tr>
<tr>
<td>J +</td>
<td>{1:20} INTEGER</td>
</tr>
<tr>
<td>B +</td>
<td>{2:16,4} SCALAR</td>
</tr>
<tr>
<td>N +</td>
<td>{3:16,10,4} MATRIX</td>
</tr>
<tr>
<td>D +</td>
<td>{1:6} CHARACTER</td>
</tr>
</tbody>
</table>
ARRAY PROCESSING (CON'T.)

ARRAYNESS IS A FUNCTION OF SUBSCRIPTING!

\[ a \]
\[ A*(\_; \_; 3) \rightarrow (2;16,4) \]
\[ M*(\_; 3; \_; *) \rightarrow (3;16,10,4) \]

TWO OPERANDS HAVE MATCHING ARRAYNESS IF THE N-TUPLES ARE IDENTICAL:

\[ A*(\_; \_; 3) = B = (2;16,4) \]
\[ M*(\_; 3; \_; *) = N = (3;16,10,4) \]

B. ARRAYED EXPRESSIONS

AN ARRAYED EXPRESSION IS AN ORDINARY EXPRESSION IN WHICH THE OPERANDS HAVE ARRAYNESS. AN ARRAYED EXPRESSION CAN BE ASSIGNED TO (OR COMPARED AGAINST) AN ARRAYED DATA ITEM.
ARRAY PROCESSING (CON'T.)

EXAMPLE

DECLARE ARRAY(50), S, T,
DECLARE I INTEGER;

DO FOR I = 1 TO 50;
    $S[I] = T[I]$;
END;

$S = T$;

• IN AN ARRAYED EXPRESSION, EACH OPERAND MAY OR MAY NOT HAVE
  ARRAYNESS -- BUT ALL OPERANDS THAT HAVE ARRAYNESS MUST MATCH
  IN ARRAYNESS.
EXAMPLE:

DECLARE ARRAY(3,6) INTEGER, I, J;
DECLARE K ARRAY(4,10,4) INTEGER;
DECLARE S ARRAY(3,6) SCALAR;

I + J is LEGAL (ARRAYNESSES ARE (2:3,6))
I + J + 6 is LEGAL (ARRAYNESSES ARE (2:3,6))
I + K is ILEGAL (ARRAYNESS OF K IS (3:4,10,4))
I + K$3 AT 1, 6 AT 2,3) IS LEGAL
(K HAS BEEN REDUCED TO ARRAYNESS (2:3,6) BY SUBSCRIPTING)
I + J + S is LEGAL (RESULT WILL BE AN ARRAY(3,6) OF SCALARS)
I J S is LEGAL (RESULT WILL BE AN ARRAY(3,6) OF SCALARS)
USER-DEFINED FUNCTIONS INVOLVED IN AN ARRAYED EXPRESSION WILL BE INVOKED ONLY ONCE UNLESS AN ARRAYED ARGUMENT IS BEING PASSED TO A FUNCTION WHICH ACCEPTS UNARRAYED ARGUMENTS.

DECLARE ARRAY(10), I, J;
DECLARE K;
F1: FUNCTION(ARG);
   DECLARE ARG;
   ...
   CLOSE F1;
F2: FUNCTION(ARG);
   DECLARE ARG ARRAY(10);
   ...
   CLOSE F2;
   ...
   I + J + F1(K);
   F1 INVOKED ONCE
   I + J + F2(I);
   F2 INVOKED ONCE
   I + J + F1(I);
   F1 INVOKED 10 TIMES
   I + J + F2(K);
   ERROR-PARAMETER MISMATCH
ARRAY PROCESSING (CON'T.)

BUILT-IN FUNCTIONS, E.G., SQRT, SIN, ..., ARE BETTER BEHAVED -- THE COMPILER KNOWS THAT THEY CANNOT INFLUENCE "EXTERNAL" DATA.

- A BUILT-IN FUNCTION WITH NO ARGUMENTS OR WITH ARGUMENTS THAT ARE NOT ARRAYED MAY ULTIMATELY BE EVALUATED ONLY ONCE IN AN ARRAYED EXPRESSION

EXAMPLE

DECLARE C SCALAR INITIAL(5);
DECLARE ARRAY(20) SCALAR, S, T;
...
S = T + SIN(C);

CURRENTLY WILL BE EVALUATED 20 TIMES

ULTIMATELY THE EFFECT MAY BE ...

COMPILER-TEMPORARY = SIN(C)
S = T + COMPILER-TEMPORARY

AGAIN, IF SIN HAD BEEN A USER FUNCTION SUCH OPTIMIZATION IS NOT POSSIBLE.
ARRAY PROCESSING (CON'T.)

- If the built-in function has arrayed arguments (and the arraynesses of all arguments match the arrayness of the expression -- an error would otherwise result) then the function is invoked once per elemental evaluation. On each evaluation the function operates on successive elements of the arrayed input arguments.

Example

```
DECLARE ARRAY(20), S, T, U;
S = ARCTAN2(T, U);
```

This is equivalent to:

```
DO FOR I = 1 TO 20;
  S$I = ARCTAN2(T$I, U$I);
END;
```
IMPORTANT NOTE:

IF THE BUILT-IN FUNCTION NORMALLY TAKES AN ARRAYED ARGUMENT THEN THE FUNCTION ACTS ON THE WHOLE ARGUMENT EACH CYCLE THROUGH THE ARRAYED EXPRESSION.

SUCH BUILT-IN FUNCTIONS ARE:

(a IS A 1-, 2-, OR 3-DIMENSIONAL ARRAY OF INTEGERS OR SCALARS)

MAX(a)                  MAXIMUM (GREATEST) ELEMENT OF THE ARRAY
MIN(a)                  MINIMUM (LEAST) ELEMENT OF THE ARRAY
PROD(a)                 PRODUCT OF ALL ELEMENTS OF THE ARRAY
SUM(a)                  SUM OF ALL ELEMENTS OF THE ARRAY
EXAMPLE

DECLARE ARRAY(20), S, T, U;
S = T + SUM(U);

THIS IS EQUIVALENT TO:
DO FOR I = 1 TO 20;
S$I = T$1 + SUM(U);
END; ACTS LIKE A CONSTANT

C. ARRAYED ASSIGNMENTS

AN ARRAYED ASSIGNMENT IS OF ONE OF THE FOLLOWING TWO FORMS:

1. ARRAYED DATA ITEM = UNARRAYED EXPRESSION

2. ARRAYED* DATA ITEM = ARRAYED* EXPRESSION

* - IN THIS CASE, OF COURSE, ALL ARRAYNESSES MUST MATCH.
ARRAY PROCESSING (CON'T.)

IN CASE ①, ALL ELEMENTS OF THE ARRAY ON THE LEFT-HAND-SIDE OF THE ASSIGNMENT STATEMENT ARE IDENTICALLY SET EQUAL TO THE RIGHT-HAND EXPRESSION:

DECLARE ARRAY(10), A, B;

. . .
A = 0;
all 10 elements of A are zeroed

A = SUM(B);
all 10 elements of A are set equal to the sum of all elements of B

A = COS(15);
all 10 elements of A are set equal to the scalar COS(15)

IN CASE ②, SUCCESSIVE ELEMENTS OF THE LEFT-HAND-SIDE ARRAYED DATA ITEM ARE SET TO CORRESPONDING EVALUATIONS OF THE RIGHT-HAND-SIDE ARRAYED EXPRESSION:
DECLARE ARRAY (10) INTEGER, M, N, P;
DECLARE ARRAY (5,10), S, T, U;

P = M + N;

IS EQUIVALENT TO:

DO FOR I = 1 TO 10;
    P$1 = M$1 + N$1;
END;

U = S T;

IS EQUIVALENT TO:

DO FOR I = 1 TO 5;
    DO FOR J = 1 TO 10;
        U$(I,J) = S$(I,J) T$(I,J);
    END;
END;

END;
ARRAY PROCESSING (CONT.)

1 MT TESTD:
2 MT PROGRAM:
3 MT STRUCTURE Q:
2 MT 1 II ARRAY(2, 3) INTEGER,
2 MT 1 BB ARRAY(6) BIT(9),
2 MT 1 NN ARRAY(5, 10, 15) MATRIX(4, 5),
2 MT 1 N ARRAY(10) INTEGER,
2 MT 1 I INTEGER
3 MT DECLARE Q Q-STRUCTURE(30) INITIAL(6#1, 6#BIN'1', 750#(20#0), 10#4, 5);
4 MT STRUCTURE S:
4 MT 1 C ARRAY(10) VECTOR(5);
5 MT DECLARE S S-STRUCTURE(5);
6 MT DECLARE T ARRAY(5, 10) VECTOR(5);
7 MT DECLARE U ARRAY(5, 10, 5) SCALAR;
8 MT [[C]] = [[MM]]
9 MT [[T]] = [[MM]]
10 MT [[U]] = [[MM]]

*** ERROR #1 OF SEVERITY 1.  *****
*** ARRAYNESS OF LEFT HAND SIDE OF ASSIGNMENT DOES NOT MATCH THAT OF RIGHT HAND SIDE
*** ERROR #2 OF SEVERITY 1.  *****
*** TYPE OF U IS ILLEGAL FOR ASSIGNMENT FROM GIVEN RIGHT-HAND SIDE.
ARRAY PROCESSING (CON'T.)

DO FOR TEMPORARY I = 1 TO SIZE([E])
  E.SCAL1, E.SCAL2 = 0;
  I
END;

ST\#8  EQU *
  L 7,120(0)
  STH 7,170(0)
  LFXI 7,1
LBL\#6  EQU *
  STH 7,160(0)
  CH 7,170(0)
  BC 1, +15
ST\#9  EQU *
  MTH 7,707(1)
  LR 6,7
  SRA 6,1
  SRA 7,1
  SER 0,0
  LH 2,100(0)
  STE 0,0(7,2)
  STE 0,2(6,2)
ST\#10  EQU *
  LBL\#8  EQU *
  LFXI 7,1
  RH 7,160(0)
  BC 7, +16
LBL\#7  EQU *

LBL\#6  EQU *

20
HW

E+2
I
E
H°10°
**ARRAY PROCESSING (CONT.)**

11 M↑

DO FOR TEMPORARY I = 1 TO SIZE(EJ);

12 M↑

E. SCAL1 = 0;

13 M↑

E. SCAL2 = 0;

14 M↑

END;

15 M↑

CLOSE PROC;

ST#11 EQU *

L 7,12(0)

STH 7,17(0)

LFXI 7,1

LBL#9 EQU *

STH 7,16(0)

CH 7,17(0)

BC $+17$

ST#12 EQU *

MIH 7,707(1)

SRA 7,1

SER 0,0

LH 2,10(0)

STE 0,0(7,2)

ST#13 EQU *

LH 7,16(0)

MIH 7,707(1)

SRA 7,1

STE 0,2(7,2)

ST#14 EQU *

LBL#11 EQU *

ST#15

LBL#4

EQU *

LFXI 7,1

AH 7,16(0)

BC 7,*-10

BCR 7,4
CALL PROC ASSIGN([Q]);

CALL PROC ASSIGN([R]);

ST#16  EQU *
LA 5, -8(1)
LHI 6, 50
BAL 4, 0(3)

ST#17  EQU *
LA 5, 492(1)
LHI 6, 20
BAL 4, 0(3)
FURTHER ARRAYED ASSIGNMENT EXAMPLES

Given:

```
DECLARE INTEGER,
   I1 ARRAY(2,3),
   I2,
   I3 ARRAY(2,3),
   I4 ARRAY(4);
```

then

```
I1 = I2;   (ARRAYED DATA ITEM = UNARRAYED EXPRESSION)
```

is an arrayed assignment in which all elements of I1 are assigned the value of I2.

```
I1 = I3;   (ARRAYED DATA ITEM = ARRAYED EXPRESSION)
```

assigns each element of I3 to the corresponding element of I1.

```
I1 = I4;
```

is illegal because the arrayness of the receiving data item is \{2:2,3\} while that of the right hand side is \{1:4\}.

```
I2 = I1;
```

is illegal because the right hand side has arrayness while the receiving data item has none.
ARRAY PROCESSING (CON'T.)

NOTE THAT THE FOLLOWING ARE ALSO POSSIBLE:

\[ I_1 = I_1 + I_3, \]
\[ I_1 = I_1 \times I_3, \]
\[ I_1 = I_3^{**2}, \]
\[ I_1 = I_3/I_2, \]
Further given:

```plaintext
STRUCTURE Q:
1 QI INTEGER,
1 Q1,
2 QS ARRAY(4) SCALAR,
2 QC CHARACTER(80);
DECLARE ZQ1 Q-STRUCTURE(2);
DECLARE ZQ2 Q-STRUCTURE(2);
DECLARE S ARRAY(2,4) SCALAR;
```

The following assignments are legal:

```plaintext
ZQ1 = ZQ2;
ZQ1.Q1 = ZQ2.Q1;
ZQ1.Q1.QS = ZQ2.Q1.QS;
ZQ1.Q1.QS = S;
```
D. MULTIPLE ASSIGNMENTS

MULTIPLE ASSIGNMENTS HAVE BEEN DISCUSSED PREVIOUSLY. TO EXTEND THEM TO ALLOW ARRAYED BEHAVIOR WE NEED THE FOLLOWING ADDITIONAL RULE:

IF ONE RECEIVING DATA ITEM POSSESSES ARRAYNESS, THEN ALL MUST POSSESS MATCHING ARRAYNESS.
EXAMPLES OF MULTIPLE ARRAYED ASSIGNMENTS:

Given:

```
DECLARE INTEGER,
   I1 ARRAY(2,3),
   I2,
   I3 ARRAY(4),
   I4 ARRAY(2,3);
```

then

```
I1, I4 = I2;
```

is legal since the arrayness of I1 and I4 match.

However, both of the following are illegal:

```
I1, I2 = I2;
I1, I3 = I4;
```
DECLARE I ARRAY(3) INTEGER,
M MATRIX(2,2),
MA ARRAY(3) MATRIX(2,2),
MB ARRAY(2) MATRIX(2,2)

Let \( M = \begin{bmatrix} 1.75 & 0.25 \\ 0.75 & 1.25 \end{bmatrix} \) and \( I = \begin{bmatrix} 2 \\ 1 \end{bmatrix} \)

then \( M_{I,J} \equiv \begin{pmatrix} \tilde{M}_{2,1} \\ \tilde{M}_{1,1} \end{pmatrix} \equiv \begin{pmatrix} 1.75 \\ 1.75 \end{pmatrix} \)

- A linear 3-array of 2-vectors; subscripting has reduced \( M \) from a matrix to a row-vector, but since \( I \) is arrayed, the entire operand has an effective arrayness even though \( M \) itself has not.
Given:

```
DECLARE ARRAY(2,3),
V VECTOR(3),
I INTEGER;
```

with

```
V ≡ \[
\begin{pmatrix}
1.5 & 4.5 & 7.5 \\
2.5 & 5.5 & 8.5 \\
3.5 & 6.5 & 9.5 \\
-0.5 & -3.5 & -6.5 \\
-1.5 & -4.5 & -7.5 \\
-2.5 & -5.5 & -8.5
\end{pmatrix}
\]
```

and

```
I ≡ \[
\begin{pmatrix}
1 & 2 & 3 \\
3 & 1 & 2
\end{pmatrix}
```

then

```
V_{*:*i}\leftarrow V$(\ast,\ast;I)$
```

is equivalent to

```
V_{i,j;l,j} \text{ for } 1 \leq i \leq 2, 1 \leq j \leq 3
```

NOTE THAT THE ARRAYNESS OF

```
V$(\ast,\ast:)$
```

MATCHES THE

ARRAYNESS OF

```
I
```

then

```
V_{*:*i}\leftarrow V$(\ast,\ast;I)$
```

is equivalent to

```
V_{i,j;l,j} \text{ for } 1 \leq i \leq 2, 1 \leq j \leq 3
```
The arrayed vector subscript I selects an array of scalars from the vector array V as shown below:

\[
\begin{bmatrix}
1.5 & 4.5 & 7.5 \\
2.5 & 5.5 & 8.5 \\
3.5 & 6.5 & 9.5 \\
-0.5 & -3.5 & -6.5 \\
-1.5 & -4.5 & -7.5 \\
-2.5 & -5.5 & -8.5 \\
\end{bmatrix}
\]

In assignment context, the following values of V would be changed:

\[
\begin{bmatrix}
V_{1,1;1} & V_{1,2;2} & V_{1,3;3} \\
V_{2,1;3} & V_{2,2;1} & V_{2,3;2} \\
\end{bmatrix}
\]
ARRAYED SUBSCRIPTING

Q. WHAT HAPPENS WHEN A SUBSCRIPT IS ITSELF AN ARRAY OR HAS ARRAYNESS?
A. THAT IS A GOOD QUESTION.

SOME RULES:
(1) IF BOTH THE OPERAND (I.E., DATA ITEM) AND THE SUBSCRIPT HAVE ARRAYNESS THEN THE ARRAYNESSES MUST MATCH! IN THIS CASE, THE WHOLE OPERAND IS CONSIDERED TO HAVE THAT SAME ARRAYNESS.
(2) IF ONLY THE SUBSCRIPT IS ARRAYED, THEN THE OPERAND ITSELF BECOMES ARRAYED, I.E., IT INHERITS THE ARRAYNESS OF ITS SUBSCRIPT.
ARRAY PROCESSING (CON'T.)

NOTES:

(A) EXPONENTS CAN HAVE ARRAYNESS ALSO.

(B) SUBSCRIPTS, OF COURSE, CAN IN TURN HAVE SUBSCRIPTS AD INFINITUM. ONE OR MORE OF THESE CAN BE ARRAYED -- PROVDED ALL ARRAYNESSES MATCH.

NOW,

(1) IF THE SUBSCRIPTED OPERAND IS PART OF AN ARRAYED EXPRESSION (E.G., IN AN ARRAYED ASSIGNMENT) THEN THE ARRAYED SUBSCRIPT(S) ARE EVALUATED ONCE PER ELEMENTAL EVALUATION OF THE EXPRESSION. NOTE THAT ALL ARRAYNESSES MUST MATCH.

(2) IF THE SUBSCRIPTED OPERAND IS A RECEIVING DATA ITEM IN AN ASSIGNMENT (LEFT-HAND SIDE) THEN THE ARRAYED SUBSCRIPT IS EVALUATED ONCE DURING EACH ELEMENTAL ASSIGNMENT. AGAIN, ARRAYNESSES MUST MATCH.
ARRAY PROCESSING (CON'T.)

IF

DECLARE I ARRAY(2,3) INTEGER
INITIAL((1,2,3,3,1,2));

SO THAT

\[
I = \begin{pmatrix}
1 & 2 & 3 \\
3 & 1 & 2
\end{pmatrix}
\]

Note that an arrayed subscript can actually generate arrayness in an unarrayed data item. For example, if

\( C \) is an unarrayed character string with \( C = 'ABCD' \)

then

\( C_I \) has the arrayness of \( I \).

The values are selected as follows:

Then \( C_I \) would be an arrayed expression with values:

\[
('ABCD' 'ABCDA' 'ABCDB' 'ABCDCA')
('ABCDCA' 'ABCDA' 'ABCDB')
\]
In an assignment context, the following values of C would be changed:

\[
\begin{pmatrix}
C_1 & C_2 & C_3 \\
C_3 & C_1 & C_2
\end{pmatrix}
\]

Note that values of \( C_1 \), \( C_2 \), and \( C_3 \) would be each changed by two elemental assignments. The results of this assignment are therefore likely to be implementation dependent.
ARRAY PROCESSING (CON'T.)

Let MA = \[
\begin{pmatrix}
1.0 & 0.0 \\
3.0 & 2.0 \\
4.0 & 7.0 \\
6.0 & 5.0 \\
8.0 & 3.0 \\
4.0 & 9.0
\end{pmatrix}
\]
\[I_1 = 2\]
\[I_2 = 1\]
\[I_3 = 1\]

Then MA*_{1,1} *= \[
\begin{pmatrix}
M_{11,1} \\
M_{21,1} \\
M_{31,1}
\end{pmatrix}
\] = \[
\begin{pmatrix}
3.0 & 2.0 \\
4.0 & 7.0 \\
8.0 & 3.0
\end{pmatrix}
\]
is also a linear 3-array of 2-vectors: now however MA and I both have arrayness (which correctly match). Three parallel subscript evaluations are effectively performed using corresponding array elements of MA and I each time.

Note MB*_{1,1} *= is illegal since the arrayness of MB does not match the arrayness of I.

However MB*_{1,1} TO 2 * is legal since array subscripting has been used on I to force arrayness matching.

If MB = \[
\begin{pmatrix}
0.5 & 0.5 \\
0.1 & 0.3 \\
0.2 & 0.7 \\
0.4 & 0.8
\end{pmatrix}
\]
\[I_1 = 2\]
\[I_2 = 1\]

then MB*_{1,1} TO 2 * = \[
\begin{pmatrix}
MB_{11,1} \\
MB_{21,1}
\end{pmatrix}
\] = \[
\begin{pmatrix}
0.1 & 0.3 \\
0.2 & 0.7
\end{pmatrix}
\]
ARRAY PROCESSING (CON‘T.)

F. ARRAYED COMPARISONS

ARRAYED OPERANDS MAY BE USED IN RELATIONAL EXPRESSIONS --
BUT IF ONE OR BOTH OPERANDS OF A COMPARISON HAVE ARRAYNESS,
THEN ONLY THE CLASS II COMPARATIVE OPERATORS MAY BE USED,
I.E., = AND ≠.

ADDITIONALLY, OF COURSE, THE ARRAYNESSES MUST MATCH.

FURTHERMORE, A BOOLEAN EXPRESSION IN A CONDITIONAL CLAUSE
CAN BE ARRAYED:

DECLARE BOOL ARRAY(10) BOOLEAN;

...  

IF [BOOL] THEN DO;

IN THIS CASE THE THEN CLAUSE WILL BE EXECUTED IF AND ONLY IF
ALL 10 BOOLEANS ARE TRUE.
ARRAY PROCESSING (CON'T.)

EXAMPLES

DECLARE I ARRAY(4) INTEGER
  INITIAL(1,2,0,3),
  J ARRAY(4) INTEGER
  INITIAL(1,2,0,3),
  K ARRAY(4) INTEGER
  INITIAL(3,1,2,6),
  L ARRAY(2) INTEGER
  INITIAL(0,3),

  (1) IF I = J THEN
      TRUE
  (2) IF J = K THEN
      FALSE
  (3) IF J = K THEN
      TRUE
ARRAY PROCESSING (CONT.)

(4) IF I < J THEN
    [ILLEGAL]
    IF I$3 < J$2 THEN
        [LEGAL]

(5) IF K = L THEN
    [ILLEGAL]
    IF K$(2 AT 1) = L THEN
        FALSE
ARRAY PROCESSING (CONT'D.)

1 AX EI.
2 AX PROGRAM.
3 DECLARE ARRAY(4) BOOLEAN INITIAL(2#OFF, 2#ON), A.
4 DECLARE ARRAY(3) INTEGER.
5 C INITIAL(2),
6 D CONSTANT(2, 3, 1),
7 E INITIAL(2),
8 B1.
9 IF [A] THEN
10 WRITE(6) 'A',
11 END IF
12 IF [A] THEN
13 WRITE(6) 'A$2 AT 1:
14 WRITE(6) 'A$2 AT 1: ';
15 END IF
16 IF [A] THEN
17 WRITE(6) 'A$2 AT 3: ';
18 [B1] = 0
19 [B] = 0
20 [C] = 0
21 [D] = 0
22 [E] = 0
23 IF [A] THEN
24 [B1] = [D]
25 END IF
26 [B1] = [D]
27 [B] = [D]
28 [B1] = [C]
ARRAY PROCESSING (CONT'D.)

17 N: C[l] = [E];
18 N: [E] = SIZE([D]);

19 N: F: FUNCTION(X),
20 N: DECLARE X ARRAY(*) INTEGER,
21 N: RETURN SIZE([X]),
22 N: CLOSE,

23 N: IE1 = F([B1]),
24 N: CLOSE,
G. REVIEW OF INDEFINITELY ARRAYED PARAMETERS

INDEFINITELY ARRAYED PARAMETERS

THE PARAMETERS OF FUNCTIONS AND THE INPUT OR ASSIGN PARAMETERS OF PROCEDURES MAY BE DECLARED TO BE INDEFINITE ARRAYS. THE FORM OF ARRAY SPECIFICATION IS:

ARRAY(*)

Examples:

TWICE: PROCEDURE (A) ASSIGN (B);
DECLARE A ARRAY(*) VECTOR (3);
DECLARE B ARRAY(*) BIT (16);

procedure body

CLOSE TWICE;

REMEMBER THAT THE SIZE FUNCTION CAN BE USED TO FIND THE ACTUAL SIZE OF THE ARRAY AT RUN TIME.
THE NUMBER OF MULTIPLE COPIES IN A STRUCTURE PARAMETER MAY ALSO BE MADE INDEFINITE USING THE FOLLOWING FORM:

```
STRUCTURE(*)
```

Example:

```
FUN: FUNCTION(C) SCALAR;
STRUCTURE Q:
  1 QI INTEGER,
  1 QS SCALAR;
DECLARE C Q-STRUCTURE(*);
}
function body
CLOSE FUN;
```

HERE, TOO, THE SIZE FUNCTION CAN BE USED.

Note that the ability to define an indefinite array does not extend to an arrayed structure terminal.

Example:

```
BAD: FUNCTION(C) SCALAR;
STRUCTURE Q:
  1 QI INTEGER,
  1 QS ARRAY(*) SCALAR; + illegal
DECLARE C Q-STRUCTURE;
}
function body
CLOSE BAD;
```
H. ARRAYED PROCEDURE ARGUMENTS

Both input and assign arguments of a procedure invocation may possess arrayness, however, the corresponding parameters of the procedure must be arrayed also -- and have the same arrayness.

If the parameter is array(*) then one should pass in a 1-dimensional array.

• INPUT ARGUMENTS

The input parameter can be viewed as being assigned into from the input argument. Arraynesses must match, but flexibility is allowed in that the input argument may be a non-contiguous array and/or of differing precision. It may even be of differing data type if an appropriate implicit conversion capability exists.
EXAMPLE:

STRUCTURE Q:
1 I INTEGER,
1 S SCALAR DOUBLE,
1 M MATRIX;

DECLARE Q Q-STRUCTURE(100);

ALPHA: PROCEDURE(T);
DECLARE T ARRAY(50) SCALAR;

CLOSE ALPHA;

CALL ALPHA(I$(50 AT 27));

HAS DATA TYPE INTEGER (IMPLICITLY CONVERTIBLE TO SCALAR) AND ARRAYNESS {1:50}
ARRAY PROCESSING (CON'T.)

ASSIGN ARGUMENTS

RULES (SIMILAR TO THOSE FOR NON-ARRAYED ASSIGN PARAMETERS)

1. The arrayness of the argument must match that of the corresponding parameter.

2. If the parameter is an indefinite array, arrayness matching is ensured if the corresponding argument is a 1-dimensional array.

3. If the argument is part of a structure which has multiple copies, structure subscripting must be used to limit the number of copies in the argument to one.

4. If array subscripting is present it must be such as to select one array element only.

5. If component subscripting is present, where necessary array subscripting must be used to limit the number of array elements in the argument to one.
NOTE: THE MORE STRINGENT RULES FOR ASSIGN PARMS/ARGS RESULT FROM THE FOLLOWING FACTS:

1) WHETHER INPUT OR ASSIGN, ARRAYS ARE PASSED BY POINTER (REFERENCE). ON THE INPUT SIDE THIS MAY BE A POINTER TO AN ARRAY TEMPORARY CREATED BY THE COMPILER TO MAKE DATA CONTIGUOUS, CHANGE PRECISION, ETC.

2) ON THE ASSIGN SIDE, THE PROCEDURE MODIFIES THE ORIGINAL DATA DIRECTLY.
EXAMPLE 1:

```plaintext
ONE: PROCEDURE(A) ASSIGN(B);
DECLARE A ARRAY(2,3) SCALAR,
    B ARRAY(4) BIT(16);

procedure body

CLOSE ONE;

and the following data declarations:

DECLARE P1 ARRAY(2,3) SCALAR,
P2 ARRAY(2,5) SCALAR,
P3 ARRAY(4) BIT(16),
P4 SCALAR,
P5 ARRAY(2,5) BIT(16),
P6 BIT(16);

then some legal and illegal invocations of the procedure are as follows:

CALL ONE(P1) ASSIGN(P3);
CALL ONE(P2) ASSIGN(P3);
CALL ONE(P2) + P1 - P4 ASSIGN(P6);
CALL ONE(P4) ASSIGN(P5);

*1 TO 3
*1 TO 4
*3 TO 5

illegal - not arrayed
illegal - not arrayed
illegal - not arrayed
legal arrayness but illegal subscript
```
EXAMPLE 2:

TWO: PROCEDURE (A) ASSIGN (B);
DECLARE A ARRAY(*) SCALAR,
B ARRAY(*) BIT(16);

procedure body

CLOSE TWO;

CALL TWO(P1) ASSIGN(P3);
CALL TWO(P2) ASSIGN(P3);
CALL TWO(P1) ASSIGN(P6);

Illegal - not arrayed
Illegal - wrong number of array dimensions
I. ARRAYED FUNCTION ARGUMENTS

RULES

A. IF A PARAMETER IS ARRAYED, THEN THE CORRESPONDING ARGUMENT MUST BE IDENTICALLY ARRAYED.

B. IF THE PARAMETER IS UNARRAYED IT IS POSSIBLE FOR THE ARGUMENT TO HAVE ARRAYNESS -- IN THIS CASE, PROVIDED ALL ARRAYNESSES MATCH UP IN THE REST OF THE STATEMENT, THE FUNCTION WILL BE INVOKED ONCE PER ELEMENTAL EVALUATION.

EXAMPLE 1

IF THE FUNCTION EXPECTS AN ARRAY THEN ... IT MUST GET ONE!

DECLARE R ARRAY(10);
RETURN SUM(R);

T = ALPHA(S); ← ILLEGAL -- S IS NOT AN ARRAY(10)
ARRAY PROCESSING (CON'T.)

EXAMPLE 2

ALPHA: FUNCTION(R);
DECLARE R;
RETURN R**2 COS(R);
CLOSE;

DECLARE ARRAY(10), S, T, U;
DECLARE V;

U must have same arrayness as S and T

(A) \[ S = [T] + \text{ALPHA}([U]) \]
(B) \[ S = [T] + \text{ALPHA}(V) \]

(A) EQUIVALENT TO

DO FOR I = 1 TO 10;
S$I = T$I + ALPHACY(U$I)
END;

(B) EQUIVALENT TO

DO FOR I = 1 TO 10;
S$I = T$I + ALPHACY(V)
END;
There are several kinds of operation in the HAL/S language which require operands with multiple components, array elements, and structure copies to be unraveled into a linear string of data elements. The reverse process of "reraveling" a linear string of data elements into components, array elements, and structure copies also occurs. Two major occurrences of these processes are in I/O (see Section 10), and in conversion functions (see Section 6.5).

The standard order in which this unraveling and reraveling takes place is called the "natural sequence". By applying the following rules in the order they are stated, the natural sequence of unraveling is obtained. By applying the rules in reverse order, and replacing "unraveled" by "reraveled", the natural sequence for reraveling is obtained.

RULES FOR MAJOR AND MINOR STRUCTURE:

1. If the operand is a major structure with multiple copies, each copy is unraveled in turn, in order of increasing index. If the operand is a minor structure of a multiple-copy structure, then the copy of the minor structure in each structure copy is unraveled in turn in order of increasing index.

2. The method of unraveling a copy is as follows. Each structure terminal on a "branch" connecting back to the given major or minor structure operand is unraveled in turn. The order taken is the order of appearance of the terminals in the structure template.

3. Each structure terminal is unraveled according to the Rules given below.
example:

STRUCTURE A:
1 B,
2 C SCALAR,
2 D VECTOR(3),
1 E INTEGER;
DECLARE A A-STRUCTURE(3);

- order of unraveling of B is B₁, l = 1, 2, 3
- order of unraveling of each B₁ is C₁, D₁

RULES FOR OTHER OPERANDS:

1. An operand of any type (integer, scalar, vector, matrix, bit, character, or event) may possess arrayness as described in Section 5.4. Each dimension of arrayness, starting from the leftmost is unraveled in turn, in order of increasing index.

2. Integer, scalar, bit, character, and event types are considered for unraveling purposes as having only one data element.

3. Vector types are unraveled component by component, in order of increasing index.

4. Matrix types are unraveled row by row, in order of increasing index. The components of each row are unraveled in turn in order of increasing index.
RAVELING & UNRAVELING (CON'T.)

example:

DECLARE V ARRAY(2,2) VECTOR(3);

- order of unraveling of \( V \) is \( V_{i,j} \), \( i=1,2 \)
- order of unraveling of each \( V_{i,j} \) is \( V_{i,j,k} \), \( j=1,2 \)
- order of unraveling of each \( V_{i,j,k} \) is \( V_{i,j,k} \), \( k=1,2,3 \)

(standard HAL/S subscript notation used)
RAVELING & UNRAVELING (CON'T.)

IF

STRUCTURE Q:
1 A,
2 I INTEGER,
2 S,
1 B,
2 V1 VECTOR,
2 T;

DECLARE Q Q-STRUCTURE(2)
INITIAL(2,4,0,1,2,3,6.7,-2,8,4,4,5,6,9.5);
DECLARE M MATRIX INITIAL(1,0,2,2,0,-1,1,0,3);
DECLARE A ARRAY(7) BOOLEAN
INITIAL(TRUE, FALSE, TRUE, #FALSE);

THEN

(Q, M, A) UNRAVELS INTO THE LINEAR STRING:

2, 4.0, 1, 2, 3, 6.7, -2, 8.4, 4, 5, 6, 9.5,
1, 0, 2, 2, 0, -1, 1, 0, 3, TRUE,
FALSE, TRUE, FALSE, FALSE, FALSE,
FALSE
A. REVIEW AND EXTENSION OF VECTOR/MATRIX CONVERSIONS

With the vector and matrix conversion functions, vectors and matrices can be dynamically manufactured.

These conversion functions accept as input a linearized list of scalars (or integers) that may have been the results of unraveling other data items. The conversion functions then shape this linear stream into a vector or matrix -- this is why these functions are also known as *shaping functions*.
**EXPLICIT CONVERSIONS (CON’T.)**

The explicit conversions allow fairly generalized input streams, but are more restrictive than, say, a write statement. For example, the input stream to an explicit conversion function may not contain a structure.

The argument list of a VECT or MATRIX conversion may take the following general form:

```
(exp1, exp2 ......)
```

1. Each `exp` is an expression of any of the following types:
   - MATRIX
   - INTEGER
   - VECTOR
   - SCALAR

2. Any expression may possess arrayness in the sense described in Section 20.2.

3. The total number of values summed over all expressions must match the length of the vector result, or the product of the row and column dimensions of the result, as appropriate.
EXPLICIT CONVERSIONS (CON'T.)

EXAMPLE (ILLEGAL USAGE)

1 M↑ SHAPE:
2 M↑ PROGRAM;
3 M↑ STRUCTURE Q:
4 M↑ 1 A,
5 M↑ 2 V1 VECTOR,
6 M↑ 2 M1 MATRIX,
7 M↑ 1 B,
8 M↑ 2 V2 VECTOR,
9 M↑ 2 M2 MATRIX;
10 M↑ DECLARE Q Q-STRUCTURE(4) INITIAL(9#1, 9#2, 9#3, 9#4);
11 M↑ DECLARE S ARRAY(4) SCALAR DOUBLE INITIAL(10, 11, 12, 13);
12 M↑ DECLARE MM MATRIX(10, 10) DOUBLE;

C↑

E↑ * +

6 M↑ MM = MATRIX ([Q], [S]);
7 S↑ DOUBLE,10,10

*** QX1 ERROR #1 OF SEVERITY 1. *****
*** CONVERSION FUNCTIONS MAY NOT HAVE ARGUMENTS OF STRUCTURE TYPE
*** QD1 ERROR #2 OF SEVERITY 1. *****
*** DIMENSIONS OF VECTOR/MATRIX CONVERSION FUNCTION DO NOT AGREE WITH THE NUMBER
*** OF DATA ELEMENTS SUPPLIED IN THE ARGUMENT LIST
? M↑ CLOSE;
Explicit Conversions (Con't.)

Example (Proper Usage)

1. N↑ SHAPE:
   1 N↑ PROGRAM;
2 N↑ DECLARE VECTOR(3) INITIAL(1, 2, 3),
   V1, V2, V3, V4, V5, V6, V7, V8, V9, V10;
3 N↑ DECLARE MATRIX(3, 3) INITIAL(1, 0, 0, 0, 1, 0, 0, 0, 1),
   M1, M2, M3, M4, M5;
4 N↑ DECLARE S ARRAY(5, 5) SCALAR DOUBLE INITIAL(3),
5 N↑ DECLARE MM MATRIX(10, 10) DOUBLE;

6 N↑ MM = MATRIX (V1, M1, [5], V2, M2, V3, M3, V4, M4,
   @DOUBLE, 10, 10)
7 N↑ CLOSE,

Written as

MM = MATRIX @(DOUBLE, 10, 10)(V1, M1,
EXPLICIT CONVERSIONS (CON'T.)

IT IS ALSO POSSIBLE TO MIX IN '#' FORM OF REPETITION:

DECLARE V VECTOR INITIAL(1,2,3);
DECLARE M MATRIX INITIAL(1);
DECLARE MM MATRIX(4,4);

\ldots

MM = MATRIX$((4,4)(V,4#6,M))$;

OR

MM = MATRIX$((4,4)(16#0))$;

BOTH WILL RESULT IN SINGLE PRECISION MATRICES.

ALSO,

V = VECTOR(6,8,9);
MATRIX/VECTOR SHAPING (CONVERSION) FUNCTION DEFAULTS ARE:

VECTOR(3) & SINGLE PRECISION
MATRIX(3,3)

B. INTEGER AND SCALAR CONVERSIONS

SIMPLE FORM

The simple form of the INTEGER and SCALAR conversion functions is:

\[
\text{INTEGER}(\text{exp}) \\
\text{SCALAR}(\text{exp})
\]

1. \(\text{exp}\) is an expression of any of the following types:
   - BIT STRING (and BOOLEAN)
   - INTEGER
   - CHARACTER
   - SCALAR

2. \(\text{exp}\) may possess arrayness, in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (See Section 20.2).

3. Conversions to integer or scalar type proceed according to the rules given in Appendix A.
Q. WHAT DOES ALL THAT MEAN?

REMEMBER THAT WE HAVE IMPLICIT CONVERSIONS FOR

- SCALAR SINGLE
- INTEGER SINGLE
- INTEGER DOUBLE

\[\rightarrow\text{SCALAR DOUBLE}\]

- SCALAR DOUBLE
- INTEGER SINGLE
- INTEGER DOUBLE

\[\rightarrow\text{SCALAR SINGLE}\]

- SCALAR SINGLE
- SCALAR DOUBLE
- INTEGER SINGLE

\[\rightarrow\text{INTEGER DOUBLE}\]

- SCALAR SINGLE
- SCALAR DOUBLE
- INTEGER DOUBLE

\[\rightarrow\text{INTEGER SINGLE}\]

IN ASSIGNMENT STATEMENTS AND IN INPUT ARGUMENT \(\Rightarrow\) INPUT PARM.
EXPLICIT CONVERSIONS (CON'T.)

FOR ALL OTHER CASES, THE CONVERSION FUNCTION IS AVAILABLE. ONE CAN, OF COURSE, USE A CONVERSION FUNCTION EVEN WHEN AN IMPLICIT CONVERSION CAPABILITY EXISTS:

DECLARE S SCALAR DOUBLE,
    I INTEGER;

ONE CAN USE:

    S = I;

OR

    S = SCALAR$(@DOUBLE)(I);

EQUIVALENT CODE WILL RESULT.
EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

1. DECLARE ARRAY(20), S SCALAR,
   T SCALAR, I INTEGER;

   . . .
   S = I;
   I = S;

   } CONVERSIONS (ARRAYED DATA)

   OR

   S = SCALAR(I);
   I = INTEGER(S);

   } EXPLICIT CONVERSIONS (ARRAYED DATA)

   AND ALSO,
   S = T + I;

   OR

   S = T + SCALAR(I);

THE OUTPUT WRITER, OF COURSE, WILL SHOW

\[
\begin{align*}
[s] &= [1] \\
\{1\} &= [s] \\
\{s\} &= \text{SCALAR}\{\{1\}\} \\
\{1\} &= \text{INTEGER}\{s\} \\
\{s\} &= \{T\} + \{i\} \\
\{s\} &= \{T\} + \text{SCALAR}\{\{1\}\}
\end{align*}
\]
EXPLICIT CONVERSIONS (CON'T.)

CONVERSIONS TO INTEGER:

- BIT STRING/BOOLEAN
  A bit string is converted to integer by regarding it as the bit pattern of a signed integer of the desired precision. Left padding with zeros, or truncation on the left may occur.

- SCALAR
  A scalar is converted by rounding to the nearest whole number (overflow may result).

- CHARACTER
  A character string is convertible to integer only if its value represents a signed whole number -- or is a null string in which the integer is set to zero. (Runtime errors may occur.)
EXPlicit Conversions (con't.)

Conversion to Scalar:

- Bit String/Boolean first is converted to a double precision integer!

- Integer

Integers are converted directly to scalar. On the AP-101 there is some microcode assistance.

- Character

A character string is convertible to scalar only if its value represents a legal scalar - or integer - valued literal --- or is a null string in which case the scalar is set to zero. (Run time errors may occur.)
EXPLICIT CONVERSIONS (CON'T.)

FURTHER EXAMPLES (SIMPLE INTEGER & SCALAR CONVERSIONS)

DECLARE I INTEGER,
    S SCALAR;

① I = INTEGER('306');
② I = INTEGER('-402');
③ I = INTEGER('ABC');
    ILLEGAL
④ S = SCALAR('10');
⑤ S = SCALAR('-6.94E-31');
⑥ S = SCALAR('AB');
    ILLEGAL
⑦ I = INTEGER('');
    NULL STRING
S = SCALAR('');

I & S WILL BE 0.
EXPLICIT CONVERSIONS (CON’T.)

ALSO, IF

DECLARE B BIT(8) INITIAL(BIN’10110101’),
    I INTEGER;

THEN

    I = B;  IS ILLEGAL

BUT

    I = INTEGER(B);

RESULTS IN

    I = 181
LIST FORM

The list form of the INTEGER and SCALAR conversion functions creates an array result, in addition to type converting the list of expressions constituting its arguments. Its form is as follows:

\[
\begin{align*}
\text{INTEGER} & \quad (exp_1, exp_2, \ldots) \\
& \quad n_1, n_2, \ldots \\
\text{SCALAR} & \quad (exp_1, exp_2, \ldots) \\
& \quad n_1, n_2, \ldots \\
\end{align*}
\]

1. The subscripts \(n_i\) for \(i = 1, 2, \ldots\) are positive integers specifying the number and size of dimensions of the resulting array. The total number of values summed over all the expressions in the list must be consistent with the number of array elements implied. The upper limit on \(i\) is 3.

2. The subscripts may be omitted entirely, in which case a linear 1-dimensional array is created, whose length is equal to the total number of values summed over all the expressions.
EXPLICIT CONVERSIONS (CON’T.)

3. Each exp is an expression of any of the following types:

   INTEGER   MATRIX
   SCALAR    BIT STRING (and BOOLEAN)
   VECTOR    CHARACTER

   and may optionally possess arrayness.

4. Conversions to integer or scalar type proceed according to the rules given in Appendix A.
THE LIST FORM THUS ALLOWS CREATION OF INTEGER/SCALAR ARRAYS OF FROM 1 TO 3 DIMENSIONS.

AS IN THE CASE OF THE VECTOR/MATRIX CONVERSION FUNCTIONS, A PRECISION MAY BE SPECIFIED (DEFAULT IS SINGLE) FOR THE INTEGER AND SCALAR CONVERSION FUNCTIONS:

EXAMPLES.

(1) DECLARE A ARRAY(5) INTEGER
    INITIAL(1,2,3,4,5),
    B ARRAY(5) INTEGER
    INITIAL(6,7,8,9,10),

    INTEGER(A,B) CREATES AN ARRAY(10)
    = (1,2,3,4,5,6,7,8,9,10)

    INTEGER$(@DOUBLE)(A,B) CREATES AN ARRAY(10) OF DP INTEGERS
    = (1,2,3,4,5,6,7,8,9,10)

    INTEGER$(@DOUBLE,10)(A,B) CREATES AN ARRAY(10) OF DP INTEGERS
    = (1,2,3,4,5,6,7,8,9,10)

    SCALAR$(@DOUBLE,2,5)(A,B) CREATES A 2-D ARRAY OF DOUBLE PRECISION SCALARS.
(2) DECLARE B ARRAY(3,4);
    B = SCALAR$(3,4)(4#1, 4#2, 4#3);

    CREATES
    \[
    \begin{bmatrix}
    1 & 1 & 1 & 1 \\
    2 & 2 & 2 & 2 \\
    3 & 3 & 3 & 3
    \end{bmatrix}
    \]

(3) DECLARE C ARRAY(4,4,4) DOUBLE;
    C = SCALAR$(\text{@DOUBLE},4,4,4)
    (4#6,15#9,27#3,18#0);

(4) DECLARE V VECTOR INITIAL(1,2,3);
    DECLARE B BIT(8) INITIAL(BIN(8)'1');
    DECLARE M MATRIX INITIAL(0);

    SCALAR$(\text{@DOUBLE})(V,B,M) CREATES AN ARRAY(13) OF DOUBLE
    PRECISION SCALARS
    = (1,2,3,255,0,0,0,0,0,0,0,0,0)
READALL(5) CAR1:

\[ T = \text{SCALAR} \ (\text{CARD1 TO 20}) \]

\[ R = \text{VECTOR} \ (\text{SCALAR} \ (\text{CARD1} \ TO 20), \ \text{CARD1} \ TO 20, \ \text{CARD1} \ TO 40, \ \text{CARD1} \ TO 60, \ \text{CARD1} \ TO 80) \]

READALL(5) CAR1:

\[ V = \text{VECTOR} \ (\text{SCALAR} \ (\text{CARD1} \ TO 20), \ \text{CARD1} \ TO 20, \ \text{CARD1} \ TO 40, \ \text{CARD1} \ TO 60) \]

EPHGMT = \text{SCALAR} \ (\text{CARD1} \ TO 80)

IF EPHGMT = 0 THEN

DO:

READALL(5) CAR1:

READALL(5) CAR2:

READALL(5) C1:

RNP = MATRIX \ (\text{SCALAR} \ (\text{CARD1} \ TO 20), \ \text{CARD1} \ TO 40, \ \text{CARD1} \ TO 60, \ \text{CARD1} \ TO 80, \ \text{CARD1} \ TO 80, \ \text{CARD1} \ TO 80, \ \text{CARD1} \ TO 80, \ \text{CARD1} \ TO 80) \]

\[ T = \text{SCALAR} \ (\text{CARD1} \ TO 20) \]

PNP = RNP;

END:
C. BIT CONVERSION FUNCTION

There are two distinct modes of the bit conversion function:

1. Bit [SIMPLE] String
   - BIT STRING/BOOLEAN
   - INTEGER
   - SCALAR
   - CHARACTER

2. Bit [RADIX] String
   - CHARACTER
   (with specification of a radix)

Key points:

- Bit always produces a 32 bit value, although the function itself allows subscripts that may chop this down.

- Bit cannot create an array of bit strings (contrast to the integer & scalar conversion functions) -- but it may be used in the array processing sense to convert an entire array.
EXPLICIT CONVERSIONS (CONT.)

SIMPLE FORM

THE SIMPLE FORM OF BIT CONVERSION IS AS FOLLOWS:

1. exp is an expression of any of the following types:
   - INTEGER
   - BIT STRING (and BOOLEAN)
   - SCALAR
   - CHARACTER

2. exp may possess arrayness in which case the arrayness must match that of the expression of which the conversion forms a part. The result is to cause an elemental conversion for every elemental evaluation of the outer expression (see Section 20.2).

3. Conversion to bit string type proceeds according to the rules given in Appendix A. The result is always a 32-bit string.

4. subscript represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on bit string data items as described in Section 17.3.

5. If subscript is absent, the result of the function is the entire bit string generated by the conversion.
EXPLICIT CONVERSIONS (CON'T.)

SINCE THERE ARE NO IMPLICIT CONVERSIONS FROM INTEGER OR SCALAR TO BIT STRINGS, THE BIT CONVERSION FUNCTION IS IMPORTANT.

EXAMPLES: (SIMPLE BIT CONVERSION)

If I is a halfword integer with \( I = 5 \)
then \( \text{BIT}(I) = 00000005_{16} \)

If C is a character data item with \( C = '10110011101' \)
then \( \text{BIT}(C) = 000000000000000000010110011101_{2} \)

\( \text{BIT}_{17} \text{ TO } 32(C) = 0000010110011101_{2} \)

and \( \text{BIT}_{28} \text{ TO } 32(C) = 11101_{2} \)
The radix form of BIT conversion is used when a character value is to be converted by an explicit rule to a bit string. A radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

<table>
<thead>
<tr>
<th>Radix</th>
<th>Digit String</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\text{@BIN}$</td>
<td>binary</td>
</tr>
<tr>
<td>$\text{@OCT}$</td>
<td>octal</td>
</tr>
<tr>
<td>$\text{@DEC}$</td>
<td>decimal</td>
</tr>
<tr>
<td>$\text{@HEX}$</td>
<td>hexadecimal</td>
</tr>
</tbody>
</table>

1. $\text{exp}$ is an expression of character type whose value must consist entirely of a string of digits legal for the specified radix.

2. The radices have the following meanings:

3. $\text{exp}$ may possess arrayness with the same implications as in the simple form of BIT conversion.

4. The conversion generates the binary representation of the input digit string. The binary representation is truncated or padded with binary zeros on the left to create a 32-bit string.
EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

ALL 32-BIT

BIT@HEX('FA0') = 00000FA0

BIT@DEC('1024') = 00000400

BIT@OCT('177777') = 0000FFFF

BIT@HEX('P0F1F2F3F4') = F1F2F3F4

TRUNCATION ON THE LEFT
D. CHARACTER CONVERSION FUNCTION

As is the case with the bit conversion function, there are two distinct modes of the character conversion function:

1. Character string/Boolean scalar

2. Character string with specification of a radix
The simple form of CHARACTER conversion is as follows:

\[ \text{CHARACTER}_{\text{subscript}}(\text{exp}) \]

1. \( \text{exp} \) is an expression of any of the following types:
   - INTEGER
   - BIT STRING (and BOOLEAN)
   - SCALAR
   - CHARACTER

2. \( \text{exp} \) may possess arrayness, with the same implications as in the BIT conversion function. (See Section 21.3).

3. Conversion to character type proceeds according to the rules given in Appendix A. The length of the result of conversion depends on the type of the input data.

4. \( \text{subscript} \) represents component subscripting on the result of the conversion. It possesses the same forms as component subscripting on character data items as described in Section 6.1.

5. If \( \text{subscript} \) is absent, then the result of the function is the entire string of characters generated by the conversion.
EXPLICIT CONVERSIONS (CON'T.)

EXAMPLES

If I is a halfword integer with $I = 173$
then $\text{CHARACTER}(I) \equiv '173'$

$\text{CHARACTER}_1 \text{ TO } 2(I) \equiv '17'$

$\text{CHARACTER}_1 \text{ TO } 5(I) \equiv '173'$

If B is a bit string of length 4 with
$B \equiv 0101_2$
then

$\text{CHARACTER}(B) \equiv '101'$

(note removal of leading zeroes)

NOTE THAT THE LENGTH OF THE RESULTANT CHARACTER STRING IS DATA DEPENDENT, WHEREAS FOR THE BIT CONVERSION FUNCTION IT IS ALWAYS 32 BITS.
EXPLICIT CONVERSIONS (CON'T.)

RADIX FORM

The radix form of CHARACTER conversion is used when a bit string value is to be converted by an explicit rule to a character string. Analogous to the radix form of BIT function, a radix specifying the conversion rule is supplied in place of a subscript. The possible forms are as follows:

- $\text{CHARACTER}_\text{BIN}(\exp)$
- $\text{CHARACTER}_\text{OCT}(\exp)$
- $\text{CHARACTER}_\text{DEC}(\exp)$
- $\text{CHARACTER}_\text{HEX}(\exp)$

1. $\exp$ is an expression of bit string type, and possibly possessing arrayness, with the same implications as in the BIT conversion function.

2. The value of the bit string is converted to a string of digits as specified by the radix, removing leading zeroes.

3. The radices have the following meanings:

<table>
<thead>
<tr>
<th>radix</th>
<th>digit string</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIN</td>
<td>binary</td>
</tr>
<tr>
<td>OCT</td>
<td>octal</td>
</tr>
<tr>
<td>DEC</td>
<td>decimal</td>
</tr>
<tr>
<td>HEX</td>
<td>hexadecimal</td>
</tr>
</tbody>
</table>

4. The length of the resulting string varies depending on the value of $\exp$. 

EXAMPLES:

\[
\text{CHARACTER}_{\text{BIN}}(\text{BIN'}001010') \equiv '001010'
\]

\[
\text{CHARACTER}_{\text{OCT}}(\text{BIN'}001010') \equiv '12'
\]

\[
\text{CHARACTER}_{\text{DEC}}(\text{BIN'}001010') \equiv '10'
\]

\[
\text{CHARACTER}_{\text{HEX}}(\text{BIN'}001010') = '0A'
\]
PRIME NUMBER GENERATOR

//CHS1040T JOB 7491, SCHUL END RQ C, TIME=1, PRTY=1, REGION=350K,
// NOTFy=CHS1040
//HAL EXEC HALSCCLD, ACCT=NOTIFY, OPTION='LIST'
//HAL SYSPRINT DD SYSOUT=2
//HAL SYSIN DD *

ERATOSTHENES: PROGRAM;
REPLACE N BY "5000";
DECLARE INTEGER, I, J, COUNT INITIAL(1);
DECLARE SIEVE ARRAY(N) INTEGER;
DO FOR I = 2 TO N;
  SIEVE$I = 1;
END;
DO FOR I = 2 TO N;
  IF SIEVE$I = 0 THEN DO;
    WRITE (6) COUNT, SIEVE$I;
    COUNT = COUNT + 1;
    DO FOR J = 2 TO N BY I;
      SIEVE$J = 0;
    END;
  END;
END;
CLOSE ERATOSTHENES;
/*
//GO CHANNELS DD SYSOUT=2
//GO REQUESTS DD *
EXECUTE TEMPPNAME;
AT END: PROFILE;
/*
PARMN FIELD: LIST

COMPLETE LIST OF COMPIL-TIME OPTIONS IN EFFECT

*** TYPE 1 OPTIONS ***

NODUMP INSTEAD OF DUMP
NOLISTING2 INSTEAD OF LISTING2
LIST INSTEAD OF NOLIST
TRACE INSTEAD OF NOTRACE
NODECK INSTEAD OF DECK
TABLES INSTEAD OF NOTABLES
NOTABLEST INSTEAD OF TABLEST
NOADDRS INSTEAD OF ADDRS
NOSRN INSTEAD OF SRN
NOSDL INSTEAD OF SDL
NOTABDNM INSTEAD OF TABDMP
ZCON INSTEAD OF NOZCON
NOCFDATA INSTEAD OF FCDATA

*** TYPE 2 OPTIONS ***

TITLE =
LINECT = 59
PAGES = 250
SYMBOLS = 200
MACROSIZE = 500
LITSTRINGS = 2000
COMPUNIT = 0
XREFSIZE = 2000
CARDDTYPE =
LABELSIZE = 1200
1 MT ERATOSTHENES:
2 MT PROGRAM;
3 MT REPLACE N BY "5000";
4 MT DECLARE INTEGER,
5 MT       I, J,
6 MT       COUNT INITIAL(0)
7 MT DECLARE SIEVE ARRAY(N) INTEGER)
8 MT DO FOR I = 2 TO N;
9 MT SIEVE = 1,
10 MT IF SIEVE \= 0 THEN
11 MT       DO;
12 MT WRITE(6) COUNT, SIEVE ;
13 MT       I
14 MT       COUNT = COUNT + 1;
15 MT       DO FOR J = 2 I TO N BY I;
16 MT SIEVE = 0;
17 MT       J
18 MT END;
19 MT END;
20 MT CLOSE ERATOSTHENES;
**Eratosthenes Program**  
HAL/S Compilation  
INTERNETRICS, INC.  
OCTOBER 11, 1975

**Symbol & Cross Reference Table Listing:**  
(Cross Reference Flag Key: 4 = Assignment, 2 = Reference, 1 = Subscript Use, 0 = Definition)

<table>
<thead>
<tr>
<th>DCL NAME</th>
<th>TYPE</th>
<th>ATTRIBUTES &amp; CROSS REFERENCE</th>
</tr>
</thead>
<tbody>
<tr>
<td>COUNT</td>
<td>INTEGER</td>
<td>SINGLE, ALIGNED, STATIC, INITIAL</td>
</tr>
<tr>
<td>ERATOSTHENES</td>
<td>PROGRAM</td>
<td></td>
</tr>
<tr>
<td>I</td>
<td>INTEGER</td>
<td>SINGLE, ALIGNED, STATIC</td>
</tr>
<tr>
<td>J</td>
<td>INTEGER</td>
<td>SINGLE, ALIGNED, STATIC</td>
</tr>
<tr>
<td>N</td>
<td>REPLACE MACRO</td>
<td>MACRO-TEXT INDEX=1</td>
</tr>
<tr>
<td>SIEVE</td>
<td>INTEGER ARRAY</td>
<td>ARRAY(5000), SINGLE, ALIGNED, STATIC</td>
</tr>
</tbody>
</table>

**Macro Text Listing:**

LOC TEXT

1 5000

HAL/S Compilation  
INTERNETRICS, INC.  
OCTOBER 11, 1975

**Optional Table Sizes**

<table>
<thead>
<tr>
<th>NAME</th>
<th>REQUESTED</th>
<th>USED</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYMBOLES</td>
<td>200</td>
<td>6</td>
</tr>
<tr>
<td>MACROSISE</td>
<td>500</td>
<td>5</td>
</tr>
<tr>
<td>LITSTRINGS</td>
<td>2000</td>
<td>0</td>
</tr>
<tr>
<td>XREFSIZE</td>
<td>2000</td>
<td>24</td>
</tr>
</tbody>
</table>

\[ y = 5 \]
CALLS TO SCAN = 187
CALLS TO IDENTIFY = 28
NUMBER OF REDUCTIONS = 300
MAX STACK SIZE = 13
MAX IND. STACK SIZE = 6
END IND. STACK SIZE = 4
END ARRAY STACK SIZE = 8
MAX EXT+ARRAY INDEX = 3
STATEMENT COUNT = 10
MINOR COMPACTIFIES = 0
MAJOR COMPACTIFIES = 0
MAX NESTING DEPTH = 1
FREE STRING AREA = 39643


17 CARDS WERE PROCESSED.
NO ERRORS WERE DETECTED DURING PHASE 1.

TOTAL CPU TIME FOR PHASE 1 0:0:0.69.
CPU TIME FOR PHASE 1 SET UP 0:0:0.09.
CPU TIME FOR PHASE 1 PROCESSING 0:0:0.54.
CPU TIME FOR PHASE 1 CLEAN UP 0:0:0.06.
PROCESSING RATE: 1600 CARDS PER MINUTE.

HAL/S COMPILING INTERMETRICS, INC.

HAL/S PHASE 2 ENTERED OCTOBER 11, 1975. CLOCK TIME = 13:31:8.01

HAL/S COMPILING INTERMETRICS, INC.

EsdID Name Type Length Block Name
0001 qERATOS 0000:000126 ERATOSTHENES
0002 qERATOS 0000:000048
0003 qERATOS 0000:000048
0004 qERATOS 0000:0002716
0005 qERATOS 0002
0006 IINIT 0002
0007 IOUT 0002
<table>
<thead>
<tr>
<th>LOCCTR</th>
<th>CODE</th>
<th>LABEL</th>
<th>INSN</th>
<th>OPERANDS</th>
<th>SYMBOLIC OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>000000</td>
<td>STA1</td>
<td>#0ERATOS</td>
<td>EQU *</td>
<td>TIME = 143</td>
<td>ERATOSTHENES</td>
</tr>
<tr>
<td>47FF0010</td>
<td>ERATOS</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td>0000120</td>
<td></td>
<td>DC</td>
<td>P'0000120</td>
<td>0ERATOS</td>
</tr>
<tr>
<td>00000B</td>
<td>0050</td>
<td></td>
<td>DC</td>
<td>X'0050</td>
<td></td>
</tr>
<tr>
<td>00000A</td>
<td>0CC5D9C1</td>
<td></td>
<td>DC</td>
<td>X'C5D9C1</td>
<td></td>
</tr>
<tr>
<td>00000E</td>
<td>E3D6E2E3</td>
<td></td>
<td>DC</td>
<td>X'E3D6E2E3</td>
<td></td>
</tr>
<tr>
<td>000012</td>
<td>CBC5D5C5</td>
<td></td>
<td>DC</td>
<td>X'C5D5C5</td>
<td></td>
</tr>
<tr>
<td>000016</td>
<td>E2</td>
<td></td>
<td>DC</td>
<td>X'E2</td>
<td></td>
</tr>
<tr>
<td>000017</td>
<td>00</td>
<td></td>
<td>DC</td>
<td>X'00</td>
<td></td>
</tr>
<tr>
<td>5800F004</td>
<td></td>
<td></td>
<td>LM</td>
<td>6.10.4(11)</td>
<td></td>
</tr>
<tr>
<td>00001C</td>
<td>906AB028</td>
<td></td>
<td>MV1</td>
<td>8(13),1</td>
<td></td>
</tr>
<tr>
<td>000020</td>
<td>9201D000</td>
<td></td>
<td>BALR</td>
<td>14.11</td>
<td></td>
</tr>
<tr>
<td>000024</td>
<td>O5E8</td>
<td></td>
<td>DC</td>
<td>X'0001</td>
<td></td>
</tr>
<tr>
<td>000026</td>
<td>0001</td>
<td></td>
<td>EQU</td>
<td>TIME = 0</td>
<td></td>
</tr>
<tr>
<td>000028</td>
<td>STA2</td>
<td></td>
<td>CSECT</td>
<td>ESDID = 0004</td>
<td></td>
</tr>
<tr>
<td>000029</td>
<td>STA3</td>
<td></td>
<td>DC</td>
<td>X'0001</td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td>#0ERATOS</td>
<td></td>
<td>EQU</td>
<td>TIME = 0</td>
<td></td>
</tr>
<tr>
<td>000004</td>
<td>0801</td>
<td></td>
<td>CSECT</td>
<td>ESDID = 9001</td>
<td></td>
</tr>
<tr>
<td>06006</td>
<td>STA4</td>
<td></td>
<td>EQU</td>
<td>TIME = 54</td>
<td></td>
</tr>
<tr>
<td>000006</td>
<td>#0ERATOS</td>
<td></td>
<td>CSECT</td>
<td>ESDID = 0001</td>
<td></td>
</tr>
<tr>
<td>090020</td>
<td>41900002</td>
<td></td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00002C</td>
<td>LBL#2</td>
<td></td>
<td>STH</td>
<td>9,0(0,10)</td>
<td>I</td>
</tr>
<tr>
<td>000030</td>
<td>O5E8</td>
<td></td>
<td>BALR</td>
<td>14.11</td>
<td></td>
</tr>
<tr>
<td>000032</td>
<td>0005</td>
<td></td>
<td>DC</td>
<td>X'0005</td>
<td></td>
</tr>
<tr>
<td>4930B046</td>
<td></td>
<td></td>
<td>CH</td>
<td>9,70(0,11)</td>
<td></td>
</tr>
<tr>
<td>47FF0056</td>
<td></td>
<td></td>
<td>BC</td>
<td>2,06(15,0)</td>
<td>L80056 LBL#3</td>
</tr>
<tr>
<td>00003C</td>
<td>STA6</td>
<td></td>
<td>EQU</td>
<td>TIME = 56</td>
<td></td>
</tr>
<tr>
<td>00003E</td>
<td>1A99</td>
<td></td>
<td>AR</td>
<td>9,9</td>
<td></td>
</tr>
<tr>
<td>000042</td>
<td>402A0000</td>
<td></td>
<td>LH</td>
<td>2,0(0,10)</td>
<td>I</td>
</tr>
<tr>
<td>000046</td>
<td>O5E9</td>
<td></td>
<td>STH</td>
<td>2,4(9,18)</td>
<td>SIEVE</td>
</tr>
<tr>
<td>000048</td>
<td>0006</td>
<td></td>
<td>BALR</td>
<td>14.11</td>
<td></td>
</tr>
<tr>
<td>00004A</td>
<td>ST#7</td>
<td></td>
<td>DC</td>
<td>X'0005</td>
<td></td>
</tr>
<tr>
<td>00004A</td>
<td>LBL#4</td>
<td></td>
<td>EQU</td>
<td>TIME = 58</td>
<td></td>
</tr>
<tr>
<td>41900001</td>
<td></td>
<td></td>
<td>LA</td>
<td>9,1(0,9)</td>
<td></td>
</tr>
<tr>
<td>4A90A000</td>
<td></td>
<td></td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>47FF002C</td>
<td></td>
<td></td>
<td>LA</td>
<td>9,0(0,10)</td>
<td>I</td>
</tr>
<tr>
<td>000056</td>
<td>LBL#3</td>
<td></td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>000056</td>
<td>O5E0</td>
<td></td>
<td>BALR</td>
<td>14.11</td>
<td></td>
</tr>
<tr>
<td>000058</td>
<td>0007</td>
<td></td>
<td>DC</td>
<td>X'0007</td>
<td></td>
</tr>
</tbody>
</table>
HAL/S Compilation

LOCCTR CODE  LABEL
00007A 1009
00007C 4829A004
00007E 1222
00007E 478F010D
00007E 05EB
000000 000A
000002 05EB
00000E 4100006
00000B 4100003
00000A 05EC
00007C 0000008
00007D 4000004
000094 05EC
000096 0000008
00009A 1602
00009C 05EC
00000E 0000008
000002 05EB
00000B 000B
000006 05EB
000006 4830A004
00000A 48300044
00000E 4030A004
000022 05EB
000004 000C
000086        ST#13
000086 41400002
0000BA 4C40A000
0000BE 4850A000
0000C2 4050D050
0000C6 LBL#8
0000C6 4040A002
0000CA 05EB:
0000CC 000D
0000CE 412F0106
0000D2 9100D050
0000D6 05E0
0000D8 4780E010
0000DC 4940B046
0000E0 0742
0000E2 05E0
0000E4 47F0E00A
0000E8 4940B046
0000EC 0722
0000EE        ST#14
0000EE 1A44
0000F0 1822
0000F2 4024A004
0000F6 05EB
0000F9 000E
0000FA        ST#15
0000FA LBL#10
0000FA 4040D050
0000FE 4A40A002
HAL/S COMPILATION

EQU  *  TIME = 335
LA  4,2(0,0)
MH  4,0(0,10)
LH  5,0(0,10)
STH  5,00(0,13)
EQU  *  TIME = 47
BALR 14,11
DC X'0000D'
LN  2,262(15,0)
TN  30(13),128
BALR 14,0
BC  8,16(0,14)
CH  4,70(0,11)
BCR  4,2
BALR 14,0
BC  15,10(0,14)
CH  4,70(0,11)
BCR  2,2
EQU  *  TIME = 52
AR  4,4
SR  2,2
STH  2,4(4,10)
BALR 14,11
DC X'0000E'
EQU  *  TIME = 52
EQU  *
LH  4,00(0,13)
AH  4,2(0,10)

IN TERMETRICS, INC.
<table>
<thead>
<tr>
<th>LOCCTR</th>
<th>CODE</th>
<th>LABEL</th>
<th>INSN</th>
<th>OPERANDS</th>
<th>SYMBOLIC OPERAND</th>
</tr>
</thead>
<tbody>
<tr>
<td>000102</td>
<td>47FF00C6</td>
<td></td>
<td>BC</td>
<td>15,198(15,0)</td>
<td>0000C6 LBL#3</td>
</tr>
<tr>
<td>000106</td>
<td></td>
<td>LBL#9</td>
<td>EQU</td>
<td>*</td>
<td></td>
</tr>
<tr>
<td>000106</td>
<td>05EB</td>
<td></td>
<td>BALR</td>
<td>14,11</td>
<td></td>
</tr>
<tr>
<td>000108</td>
<td>000F</td>
<td></td>
<td>DC</td>
<td>X'000F'</td>
<td></td>
</tr>
<tr>
<td>00010A</td>
<td></td>
<td>ST#16</td>
<td>EQU</td>
<td>* TIME = 0</td>
<td></td>
</tr>
<tr>
<td>00010A</td>
<td>05EB</td>
<td></td>
<td>BALR</td>
<td>14,11</td>
<td></td>
</tr>
<tr>
<td>00010C</td>
<td>0010</td>
<td></td>
<td>DC</td>
<td>X'0010'</td>
<td></td>
</tr>
<tr>
<td>00010E</td>
<td></td>
<td>ST#17</td>
<td>EQU</td>
<td>* TIME = 50</td>
<td></td>
</tr>
<tr>
<td>00010E</td>
<td></td>
<td>LBL#7</td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010E</td>
<td></td>
<td>LBL#11</td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00010E</td>
<td>41900001</td>
<td></td>
<td>LA</td>
<td>9,1(0,0)</td>
<td>00005E LBL#5</td>
</tr>
<tr>
<td>000112</td>
<td>4A90A000</td>
<td></td>
<td>AH</td>
<td>9,0(0,10)</td>
<td></td>
</tr>
<tr>
<td>000116</td>
<td>47FF005E</td>
<td></td>
<td>BC</td>
<td>15,94(15,0)</td>
<td></td>
</tr>
<tr>
<td>00011A</td>
<td></td>
<td>LBL#6</td>
<td>EQU</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00011A</td>
<td>05EB</td>
<td></td>
<td>BALR</td>
<td>14,11</td>
<td></td>
</tr>
<tr>
<td>00011C</td>
<td>0011</td>
<td></td>
<td>DC</td>
<td>X'0011'</td>
<td></td>
</tr>
<tr>
<td>00011E</td>
<td></td>
<td>ST#18</td>
<td>EQU</td>
<td>* TIME = 18</td>
<td></td>
</tr>
<tr>
<td>00011E</td>
<td>05EB</td>
<td></td>
<td>BALR</td>
<td>14,11</td>
<td></td>
</tr>
<tr>
<td>000120</td>
<td>0012</td>
<td></td>
<td>DC</td>
<td>X'0012'</td>
<td></td>
</tr>
<tr>
<td>000122</td>
<td>47F0C004</td>
<td></td>
<td>BC</td>
<td>15,4(0,12)</td>
<td></td>
</tr>
<tr>
<td>RLD POS</td>
<td>REF</td>
<td>FLAG</td>
<td>ADDRESS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>---------</td>
<td>-----</td>
<td>------</td>
<td>---------</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>002</td>
<td>08</td>
<td>000005</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>006</td>
<td>08</td>
<td>000080</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>007</td>
<td>09</td>
<td>000097</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0001</td>
<td>007</td>
<td>09</td>
<td>00009F</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>000120</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>003</td>
<td>08</td>
<td>000131</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>003</td>
<td>09</td>
<td>000135</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>001</td>
<td>08</td>
<td>000139</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>000151</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>000155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>000159</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>00015D</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>004</td>
<td>08</td>
<td>000161</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0002</td>
<td>005</td>
<td>08</td>
<td>000169</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```
HAL/S COMPIKATION

RLD POS  REF  FLAG  ADDRESS

0001  002   08   000005
0001  006   08   000080
0001  007   09   000097
0001  007   09   00009F
0002  004   08   000120
0002  003   08   000131
0002  003   09   000135
0002  001   08   000139
0002  004   08   000151
0002  004   08   000155
0002  004   08   000159
0002  004   08   00015D
0002  004   08   000161
0002  005   08   000169
```
LOC  B DISP  NAME
UND ERATOSTH ENES  STACK=88
000000  10 000  I
000002  10 002  J
000004  10 004  COUNT
000006  10 004  SIEVE
INSTRUCTION FREQUENCIES
INSH  COUNT
BALR  5
BCR  2
LTR  1
LR  1
AR  3
SR  1
STH  7
LA  0
BC  11
LH  6
CH  4
AH  4
MH  1
L  1
TM  1
MVI  1
LM  1

48 HALMAT OPERATORS CONVERTED
50 INSTRUCTIONS GENERATED
368 BYTES OF PROGRAM, 10086 BYTES OF DATA

MAX. OPERAND STACK SIZE  =9
END OPERAND STACK SIZE   =9
HAL/S COMPILATION
NUMBER OF STATEMENT LABELS USED  =11
MAX. STORAGE DESCRIPTOR STACK SIZE =1
END STORAGE DESCRIPTOR STACK SIZE =0
NUMBER OF MINOR COMPACTIFIES  =0
NUMBER OF MAJOR COMPACTIFIES  =0
FREE STRING AREA             =54356
HAL/S COMPILER PHASE 3 -- VERSION 15.0 OF SEPTEMBER 21, 1975.

HAL/S PHASE 3 ENTERED OCTOBER 11, 1975. CLOCK TIME = 13:31:10.66

SIMULATION DATA FILE #HERATOS HAS BEEN CREATED

PAGING AREA SIZE (PAGES) = 97
NUMBER OF LOCATES = 201

PREDICTED SDF SIZE (PAGES) = 1
ACTUAL SDF SIZE (PAGES) = 1

DIRECTORY FREE SPACE (BYTES) = 72
DATA FREE SPACE (BYTES) = 920
SDF SIZE (BYTES) = 688
SDF DENSITY (%) = 40
NUMBER OF BLOCK NODES = 1
NUMBER OF SYMBOL NODES = 5
NUMBER OF STATEMENT NODES = 18

NUMBER OF BLOCKS DELETED = 0
NUMBER OF SYMBOLS DELETED = 1
NUMBER OF TEMPLATES DELETED = 0
NUMBER OF MINOR COMPACTIFIES = 0
NUMBER OF MAJOR COMPACTIFIES = 0

END OF HAL/S PHASE 3 OCTOBER 11, 1975. CLOCK TIME = 13:31:11.61
TOTAL CPU TIME IN PHASE 3 = 6:0:0.18
CPU TIME FOR PHASE 3 INITIALIZATION = 6:0:0.02
CPU TIME FOR PHASE 3 FILE GENERATION = 6:0:0.05
CPU TIME FOR PHASE 3 FILE EMISSION = 6:0:0.11
<table>
<thead>
<tr>
<th>HAL/S-360 V13.0 START TIME: 13:31:46.60 DAY: 75/294</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
</tr>
<tr>
<td>2</td>
</tr>
<tr>
<td>3</td>
</tr>
<tr>
<td>4</td>
</tr>
<tr>
<td>5</td>
</tr>
<tr>
<td>6</td>
</tr>
<tr>
<td>7</td>
</tr>
<tr>
<td>8</td>
</tr>
<tr>
<td>9</td>
</tr>
<tr>
<td>10</td>
</tr>
<tr>
<td>11</td>
</tr>
<tr>
<td>12</td>
</tr>
<tr>
<td>13</td>
</tr>
<tr>
<td>14</td>
</tr>
<tr>
<td>15</td>
</tr>
<tr>
<td>16</td>
</tr>
<tr>
<td>17</td>
</tr>
<tr>
<td>18</td>
</tr>
<tr>
<td>19</td>
</tr>
<tr>
<td>20</td>
</tr>
<tr>
<td>21</td>
</tr>
<tr>
<td>22</td>
</tr>
<tr>
<td>23</td>
</tr>
<tr>
<td>24</td>
</tr>
<tr>
<td>25</td>
</tr>
<tr>
<td>26</td>
</tr>
<tr>
<td>27</td>
</tr>
<tr>
<td>28</td>
</tr>
<tr>
<td>29</td>
</tr>
<tr>
<td>30</td>
</tr>
<tr>
<td>31</td>
</tr>
<tr>
<td>32</td>
</tr>
<tr>
<td>33</td>
</tr>
<tr>
<td>34</td>
</tr>
<tr>
<td>35</td>
</tr>
<tr>
<td>36</td>
</tr>
<tr>
<td>37</td>
</tr>
<tr>
<td>38</td>
</tr>
<tr>
<td>39</td>
</tr>
<tr>
<td>40</td>
</tr>
<tr>
<td>41</td>
</tr>
<tr>
<td>42</td>
</tr>
<tr>
<td>43</td>
</tr>
<tr>
<td>44</td>
</tr>
<tr>
<td>45</td>
</tr>
<tr>
<td>46</td>
</tr>
<tr>
<td>47</td>
</tr>
</tbody>
</table>
### Statement Profile for Program Eratosthenes

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use and % of Total</th>
<th>Time</th>
<th>Cost and % of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>1-10</td>
<td>1</td>
<td>0.00</td>
<td>14</td>
</tr>
</tbody>
</table>

**2 to 4 Not Executed**

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use and % of Total</th>
<th>Time</th>
<th>Cost and % of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>15</td>
<td>0.03</td>
<td>18</td>
</tr>
<tr>
<td>6</td>
<td>4,999</td>
<td>10.83</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>1</td>
<td>0.00</td>
<td>5</td>
</tr>
<tr>
<td>8</td>
<td>5,000</td>
<td>10.83</td>
<td>10</td>
</tr>
<tr>
<td>9</td>
<td>4,999</td>
<td>10.83</td>
<td>7</td>
</tr>
<tr>
<td>10</td>
<td>669</td>
<td>1.44</td>
<td>0</td>
</tr>
<tr>
<td>11</td>
<td>669</td>
<td>1.44</td>
<td>11</td>
</tr>
<tr>
<td>12</td>
<td>669</td>
<td>1.44</td>
<td>6</td>
</tr>
<tr>
<td>13</td>
<td>11,730</td>
<td>22.43</td>
<td>34</td>
</tr>
<tr>
<td>14</td>
<td>11,069</td>
<td>22.90</td>
<td>5</td>
</tr>
<tr>
<td>15</td>
<td>669</td>
<td>1.44</td>
<td>5</td>
</tr>
<tr>
<td>16</td>
<td>669</td>
<td>1.44</td>
<td>0</td>
</tr>
<tr>
<td>17</td>
<td>1</td>
<td>0.00</td>
<td>5</td>
</tr>
<tr>
<td>18</td>
<td>1</td>
<td>0.00</td>
<td>2</td>
</tr>
</tbody>
</table>

### Compilation Unit Use Summary

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use and % of Total</th>
<th>Time</th>
<th>Cost and % of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>46,154</td>
<td>100.00</td>
<td></td>
<td>629,169</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Statement</th>
<th>Use and % of Total</th>
<th>Time</th>
<th>Cost and % of Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>46,154</td>
<td>100.00</td>
<td></td>
<td>629,169</td>
</tr>
</tbody>
</table>
DATA STORAGE AND ACCESS

- DATA IS NOT NECESSARILY PHYSICALLY ALLOCATED IN THE ORDER IN WHICH THE USER MAKES THE DECLARES.

EXAMPLE

CPL: COMPOOL;

DECLARE ARR1 ARRAY(3,6) SCALAR;
DECLARE ARR2 ARRAY(20) INTEGER;
DECLARE I INTEGER;
DECLARE S1 SCALAR DOUBLE;
DECLARE S2 SCALAR;
DECLARE S3 SCALAR;

CLOSE CPL;

WOULD PROBABLY BE ALLOCATED AS:

```
   #PCPL
  SIMPLE DATA
   #PCPL
  AGGREGATE DATA
```

S1
S3
S2
I
ARR2
ARR1
GENERAL ALLOCATION RULES:

1) HAL REORDERS DATA IN AN ATTEMPT TO PROVIDE ADDRESSING COVERAGE FOR THE DATA USING A MINIMUM OF DISTINCT BASE REGISTER VALUES.

2) SIMPLE DATA (OFFSET = 0) COMES BEFORE AGGREGATE DATA (OFFSET ≠ 0).
   SIMPLE DATA: INTEGER, SCALAR, STRINGS (BIT & CHARACTER).
   AGGREGATE DATA: ARRAYS, VECTORS, MATRICES AND STRUCTURES.

3) WITHIN EACH OF THE TWO GROUPS, DATA IS ORDERED SUCH THAT ITEMS REQUIRING THE SAME BOUNDARY ALIGNMENTS ARE ADJACENT -- THUS MINIMIZING WASTED SPACE.

4) WITHIN THE AGGREGATE GROUP SINGLE DIMENSIONAL ARRAYS COME BEFORE MULTI-DIMENSIONAL ARRAYS.
5) Structure templates are internally ordered such that the minimum boundary alignment within any node level is required. (More on this later.)

Generally, then, data is not allocated in the declared order except in some obvious cases, e.g.:

```
DECLARE VECTOR DOUBLE, V1, V2, V3, V4, V5, V6, V7, V8, V9;
```

In such a case the compiler will not alter the order since nothing would be gained.

Nonetheless, when the HALS compiler is given free rein to reorder data, users should never try to predict the resulting order, nor assume that the order generated will not change when compiler improvements are made.
DATA STORAGE AND ACCESS (CON'T.)

WHEN USERS MUST CONTROL THE PHYSICAL LAYOUT OF DATA (E.G., FOR INTERFACING TO EXTERNALLY DEFINED DATA LAYOUTS), THE RIGID KEYWORD MAY BE USED.

RIGID TELLS THE COMPILER TO ALLOCATE DATA IN THE DECLARED ORDER. IT MAY BE SPECIFIED ONLY ON COMPOOL BLOCK HEADERS, AND ON STRUCTURE TEMPLATES.

EXAMPLE 1:

CPL: COMPOOL RIGID;

DECLARE ARR1 ARRAY(3,6) SCALAR;
DECLARE ARR2 ARRAY(20) INTEGER;
DECLARE I INTEGER;
DECLARE S1 SCALAR DOUBLE;
DECLARE S2 SCALAR;
DECLARE S3 SCALAR DOUBLE;
CLOSE CPL;
DATA STORAGE AND ACCESS (CONTINUED)

RESULTS IN

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ARR1</td>
<td>1 HW</td>
</tr>
<tr>
<td>ARR2</td>
<td>1 HW</td>
</tr>
<tr>
<td></td>
<td>3 HW</td>
</tr>
<tr>
<td></td>
<td>4 HW</td>
</tr>
<tr>
<td>I</td>
<td>1 HW</td>
</tr>
<tr>
<td>S1</td>
<td>2 HW</td>
</tr>
<tr>
<td>S2</td>
<td>2 HW</td>
</tr>
<tr>
<td>S3</td>
<td>4 HW</td>
</tr>
</tbody>
</table>

18 SCALARS = 36 HW
20 INTEGERS = 20 HW
1 HW
3 HW
4 HW
2 HW
2 HW
4 HW

NOTES:
1) UNLESS USERS ARE CAREFUL, RIGID CAN CAUSE WASTED SPACE.
2) ON THE FC, DOUBLE PRECISION SCALARS ARE ALLOCATED ON DOUBLE-WORD BOUNDARIES EVEN THOUGH THE HARDWARE (AP-101) DOES NOT REQUIRE IT.
WHY? FOR STORAGE LAYOUT COMPATIBILITY WITH THE 360.
EXAMPLE 2:

```
STRUCTURE Q RIGID;
  1 S1 SCALAR,
  1 I1 INTEGER,
  1 B1 BIT(16),
  1 S2 SCALAR DOUBLE,
  1 V VECTOR;

DECLARE Q Q-STRUCTURE;
```

RESULTS IN:

<table>
<thead>
<tr>
<th></th>
<th>HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>2</td>
</tr>
<tr>
<td>I1</td>
<td>1</td>
</tr>
<tr>
<td>B1</td>
<td>1</td>
</tr>
<tr>
<td>S2</td>
<td>4</td>
</tr>
<tr>
<td>V$1</td>
<td>2</td>
</tr>
<tr>
<td>V$2</td>
<td>2</td>
</tr>
<tr>
<td>V$3</td>
<td>2</td>
</tr>
</tbody>
</table>
DATA STORAGE AND ACCESS (CONTINUED)

NOTES

1) USE OF RIGID ON STRUCTURE TEMPLATES IS ESPECIALLY LIKELY TO CAUSE WASTE OF MEMORY UNLESS CARE IS TAKEN.

2) RIGID MUST BE SPECIFIED ON THE TEMPLATE DECLARATION -- OR A MINOR STRUCTURE THEREOF. RIGID MAY NOT BE APPENDED TO A STRUCTURE DECLARATION.

3) WHETHER A STRUCTURE TEMPLATE IS RIGID OR NOT, THE BOUNDARY ALIGNMENT SELECTED FOR THE TEMPLATE IS DETERMINED BY THE TERMINAL WITH THE MOST RESTRICTIVE BOUNDARY ALIGNMENT REQUIREMENTS.

4) MINOR STRUCTURES ARE ALWAYS ALLOCATED AS SEPARATE, DISCRETE UNITS WITHIN THE OVERALL STRUCTURE.

5) THE KEYWORD "RIGID" ON A COMPOOL BLOCK HEADER DOES NOT GUARANTEE STRUCTURE RIGIDITY -- ONLY WHEN RIGID IS APPLIED TO A TEMPLATE DOES IT PENETRATE.
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE 3:

CPL: COMPOOL RIGID;

DECLARE SCALAR, S1, S2;
DECLARE ARRAY(10) INTEGER DOUBLE, I1, I2, I3;

STRUCTURE Q:
    1 V VECTOR,
    1 SS SCALAR,
    1 J INTEGER;

DECLARE Q Q-STRUCTURE;
DECLARE W MATRIX(9,9) DOUBLE;

CLOSE CPL;

RESULTS IN:

<table>
<thead>
<tr>
<th></th>
<th>S1</th>
<th>S2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>I1</td>
<td>I2</td>
</tr>
<tr>
<td></td>
<td>I3</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Q</td>
<td></td>
</tr>
<tr>
<td></td>
<td>W</td>
<td></td>
</tr>
</tbody>
</table>

ALL DATA HAS BEEN ALLOCATED IN THE DECLARED ORDER.

INTERNAL TO Q, TERMINALS J, SS, AND J MAY BE REORDERED!
EXAMPLE 4:

STRUCTURE Q:

1 A,
   2 B1 BIT(4),
   2 B2 BIT(8),
   2 I INTEGER,
1 B RIGID,
   2 S1 SCALAR,
   2 S2 SCALAR DOUBLE,
   2 J INTEGER,
1 C,
   2 V1 VECTOR,
   2 V2 VECTOR DOUBLE;

THIS MUCH OF Q WILL BE RIGID

WE CALL A, B, AND C MINOR STRUCTURES OR FORKS. RIGID MAY BE APPLIED TO MINOR STRUCTURES IN WHICH CASE IT PROPAGATES DOWNWARDS TO ALL LOWER TERMINALS.
DATA STORAGE AND ACCESS (CONTINUED)

- The physical allocation of rigid structures is fairly simple to grasp once the idea of data alignment and independence of minor structures is understood.

- As will be seen, the concept of dense structures makes things slightly more complicated.

- But as to what happens when HAL/S is free to reorder structures -- that is a much more complex topic that will be taken up at a later time.

Before discussing dense we need some facts about physical organizations of data:
### DATA STORAGE AND ACCESS (CONTINUED)

#### INTEGER

<table>
<thead>
<tr>
<th></th>
<th>360</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 HW (2 BYTES)</td>
<td>1 HW (2 BYTES)</td>
<td></td>
</tr>
</tbody>
</table>

#### INTEGER

<table>
<thead>
<tr>
<th></th>
<th>360</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>2 HW (FULLWORD) (4 BYTES)</td>
<td>2 HW (FULLWORD) (4 BYTES)</td>
<td></td>
</tr>
</tbody>
</table>

#### SCALAR

<table>
<thead>
<tr>
<th></th>
<th>360</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 24</td>
<td>8 24</td>
<td></td>
</tr>
<tr>
<td>2 HW (FULLWORD) (4 BYTES)</td>
<td>2 HW (FULLWORD) (4 BYTES)</td>
<td></td>
</tr>
</tbody>
</table>

#### SCALAR

<table>
<thead>
<tr>
<th></th>
<th>360</th>
<th>FC</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 56</td>
<td>8 56</td>
<td></td>
</tr>
<tr>
<td>4 HW (DOUBLE WORD) (8 BYTES)</td>
<td>4 HW (DOUBLE WORD) (8 BYTES)</td>
<td></td>
</tr>
</tbody>
</table>

### SUMMARY: EXACT CORRESPONDENCE IN LENGTH AND BOUNDARY ALIGNMENTS BETWEEN FC AND 360.
DATA STORAGE AND ACCESS (CONTINUED)

CHARACTER STRINGS:

<table>
<thead>
<tr>
<th>M</th>
<th>C</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>SIZE</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INFO</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

M = MAX NUMBER OF CHARACTERS
C = CURRENT NUMBER OF CHARACTERS

NOTE: AN FC CHARACTER STRING ALWAYS OCCUPIES AN INTEGRAL NUMBER OF HALFWORDS (2 CHARACTERS PER HW). AN UNUSED 8 BITS MAY EXIST AT THE END.

A 360 CHARACTER STRING IS BYTE-ORIENTED. NO WASTED SPACE AT THE END.

BUT, THE FC DATA OPTION (SPECIFIABLE TO THE 360 COMPILER) WILL RESULT IN THE GENERATION OF FC-EQUIVALENT CHARACTER STRINGS.
### DATA STORAGE AND ACCESS (CONTINUED)

#### BIT STRINGS:

**A) BIT(1) BOOLEAN THROUGH BIT(8)**

<table>
<thead>
<tr>
<th>FC</th>
<th>0</th>
<th>1 HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td>0</td>
<td>1 BYTE</td>
</tr>
<tr>
<td>360 (FC DATA)</td>
<td>0</td>
<td>1 HW</td>
</tr>
</tbody>
</table>

**B) BIT(9) - BIT(16)**

<table>
<thead>
<tr>
<th>FC</th>
<th>0</th>
<th>1 HW</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td>0</td>
<td>1 HW</td>
</tr>
</tbody>
</table>

**C) BIT(17) - BIT(32)**

<table>
<thead>
<tr>
<th>FC</th>
<th>0</th>
<th>1 FW</th>
</tr>
</thead>
<tbody>
<tr>
<td>360</td>
<td>0</td>
<td>1 FW</td>
</tr>
</tbody>
</table>
DATA STORAGE AND ACCESS (CONTINUED)

NOTES:
(1) HAL/S ALWAYS CAUSES IMPLICIT INITIALIZATION OF BIT STRINGS TO ZERO.
(2) AT RUN TIME, HAL/S ASSUMES PORTION OF WORD NOT OCCUPIED BY THE BIT STRING IS ZERO.
(3) BIT STRINGS ARE ALWAYS (EXCEPT FOR DENSE BIT STRINGS) RIGHT-JUSTIFIED IN THEIR MEMORY LOCATION -- COMPONENT SUBSCRIPTING STILL STARTS WITH 1 BEGINNING WITH THE LEFT-MOST BIT OF THE ACTUAL STRING.

DECLARE B12 BIT(12) INITIAL(BIN'110111110010');

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
0 0 0 0 1 1 0 1 1 1 1 1 0 0 1 0

UNUSED B12 B12$12
```

B12$1
DATA STORAGE AND ACCESS (CONTINUED)

EVENT VARIABLES (ACTS LIKE A BOOLEAN)

0      15
FC ←USED BY FCOS →  E  1 HW

7
360  0   E  1 BYTE

360 (FC DATA)

0    7
1 HW

NAME VARIABLES* (POINTER DATA)

FC

16 BIT ADDRESS  1 HW

360

0  24 BIT ADDRESS  1 FW

* NAME VARIABLES REPRESENT AN ALLOCATION INCOMPATIBILITY BETWEEN THE FC AND 360 COMPILERS THAT FCDATA CANNOT OVERCOME!
DENSE/ALIGNED

THE KEYWORD DENSE IS ONLY EFFECTIVE WHEN APPLIED TO STRUCTURE DATA (FORKS OR TERMINALS) AND EVEN THEN ONLY CONTROLS PACKING OF BIT AND BOOLEAN DATA.

DENSE ALLOWS BIT STRINGS WITHIN STRUCTURES TO BE COMPRESSED -- SAVING CORE MEMORY BUT INCREASING ACCESS TIME GENERALLY.

EXAMPLE 1:

DECLARE S SCALAR DENSE;  ← INEFFECTUAL
DECLARE B6 BIT(6) DENSE;
STRUCTURE Q DENSE:
  1 B4 BIT(4),
  1 B10 BIT(10),
  1 B2 BIT(2);
DECLARE Q Q-STRUCTURE;
DECLARE R Q-STRUCTURE DENSE; ← ILLEGAL
DATA STORAGE AND ACCESS (CONTINUED)

ALL DATA IS INHERENTLY ALIGNED (UNLESS DENSE IS SPECIFIED) THUS THERE IS NO NEED TO SPECIFY ALIGNED ON DECLARED DATA -- ALTHOUGH THE KEYWORD ALIGNED WILL BE AUTOMATICALLY SUPPLIED IN THE CROSS-REFERENCE TABLE.

DECLARE I INTEGER ALIGNED;

LEGAL BUT WASTED KEYPUNCHING

ALIGNED, HOWEVER, CAN BE USED WITHIN A STRUCTURE TEMPLATE TO "TURN OFF" A DENSE SPECIFICATION THAT MAY BE IN FORCE.
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE:
STRUCTURE A:
1 A1 DENSE,
  2 A11 BIT(6),
  2 A12 INTEGER,
  2 A13 ARRAY(10) BOOLEAN
  ALIGNED,
1 A2 CHARACTER(80);
DECLARE ZA A-STRUCTURE;

NOTE: WITH EXAMPLEs IT FOLLOWS IN
THAT DENSE NEVER HAS AN EFFECT ON
ARRAYS OF BIT STRINGS!
NOTES:

1) DENSE IS ONLY EFFECTIVE ON SINGLE (UNARRAYED BIT STRINGS) WITHIN STRUCTURES.

2) BOTH DENSE AND RIGID CAN BE SPECIFIED FOR A STRUCTURE TEMPLATE.

<table>
<thead>
<tr>
<th>DENSE</th>
<th>RIGID</th>
<th>NOTES</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADJACENT BIT STRINGS ARE PACKED WHEN POSSIBLE</td>
<td>BIT STRINGS ARE SORTED (WITHIN A MINOR STRUCTURE) AND PACKED TOGETHER.</td>
<td></td>
</tr>
</tbody>
</table>

A FURTHER COMPLICATION EXISTS IN THAT STRUCTURES MAY BE PARTIALLY DENSE AND/OR PARTIALLY RIGID.

THE DETAILED RULES FOLLOWED BY THE COMPILER ARE TOO COMPLEX TO BE TREATED HERE.
EXAMPLE:

STRUCTURE Q DENSE RIGID:

1 A,
2 B1 BIT(4),
2 B2 BIT(8),
2 B3 BIT(2),
2 B4 BIT(6),
1 B,
2 B5 BIT(5),
2 B6 BIT(5),
1 C,
2 B7 BIT(3),
1 D,
2 B8 BIT(5),

NOTE THAT MINOR STRUCTURES ARE INDEPENDENTLY ALLOCATED!!
DATA STORAGE AND ACCESS (CONTINUED)

A PROGRAMMER USES A DENSE BIT STRING THE SAME WAY A NON-DENSE ONE IS USED ...

STRUCTURE Q DENSE:
1 BOOL1 BOOLEAN,
1 BOOL2 BOOLEAN,
1 B10 BIT(10),
1 B4 BIT(4);
DECLARE Q Q-STRUCTURE INITIAL(TRUE, FALSE, BIN'111', BIN'1')

... IF BOOL1 AND NOT BOOL2 THEN DO;
... IF B4 = BIN'1011' THEN DO;
...
INSTEAD OF THIS,

\[
\text{STRUCTURE } Q \text{ DENSE RIGID:}
\]

1 ENGINE\_ON BOOLEAN,
1 LOW\_FUEL BOOLEAN,
1 WHEELS\_DOWN BOOLEAN,

1 SOMETHING\_IS\_BURNING BOOLEAN;

DECLARE \( Q \) Q-STRUCTURE;

DO THIS:

\[
\text{STRUCTURE } Q:
\]

1 DISCRETES BIT(16);
DECLARE \( Q \) Q-STRUCTURE;

REPLACE ENGINE\_ON BY "DISCRETES\$1";
REPLACE LOW\_FUEL BY "DISCRETES\$2";

\ldots

REPLACE SOMETHING\_IS\_BURNING BY "DISCRETES\$16";
DATA STORAGE AND ACCESS (CONTINUED)

TEMPORARY DATA

- NORMAL DECLARE'D DATA IS PERMANENTLY ALLOCATED TO A STATIC DATA AREA.

- THE TEMPORARY FACILITY ALLOWS USERS TO OBTAIN SCRATCH SPACE FOR LOCALIZED CALCULATIONS THAT WILL BE OBTAINED FROM THE STACK.

- TEMPORARY VARIABLES EXIST ONLY WITHIN THE DO ... END GROUP WITHIN WHICH THEY ARE DEFINED.

- TWO FORMS EXIST:
  - TEMPORARY S SCALAR;
  - TEMPORARY MATRIX DOUBLE, M1, M2;

AND

DO FOR TEMPORARY I = 1 TO 9;
A SPECIAL LOOP VARIABLE FORM.
RESTRICTIONS

1) TEMPORARY DATA ITEMS MAY NOT BE INITIALIZED.
2) TEMPORARY EVENT VARIABLES AND NAME VARIABLES ARE NOT ALLOWED.
3) THE NAME OF A TEMPORARY MAY NOT DUPLICATE THE NAME OF ANOTHER TEMPORARY DATA ITEM IN THE SAME DO...END GROUP.
4) THE NAME OF A TEMPORARY DATA ITEM MAY NOT DUPLICATE THE NAME OF AN ORDINARY DATA ITEM KNOWN BY THE SCOPING RULES TO THE BODY OF THE DO...END GROUP.
5) TEMPORARY VARIABLES CAN ONLY BE DEFINED IMMEDIATELY FOLLOWING A DO ... AND PRIOR TO THE NEXT EXECUTABLE FORM (EXCEPT FOR THE LOOP TEMPORARY).
DATA STORAGE AND ACCESS (CONTINUED)

6) TEMPORARY VARIABLES MAY NOT BE DEFINED FOR A DO CASE BLOCK.

EXAMPLES

```
DO FOR TEMPORARY I = 1 TO 10;
  . . .
  M$1 = M$I + 1;
END
I IS ONLY KNOWN WITHIN THE DO...END BLOCK. I IS ALWAYS A SINGLE PRECISION INTEGER IN THIS LOOP VARIABLE SPECIAL FORM.

DO;
  TEMPORARY I INTEGER DOUBLE;
  TEMPORARY MATRIX, M1, M2, M3;
  . . .
  I = I + 1
  M1 = M2 + M3;
END;

I, M1, M2, M3 ARE ONLY KNOWN WITHIN THIS DO...END GROUP (AND ANYTHING NESTED WITHIN IT).
```
DATA STORAGE AND ACCESS (CONTINUED)

WRONG:

DO;
  I = I + 1;
  TEMPORARY INTEGER J;
  J = 2I;
END;

Does not immediately follow DO

RIGHT:

DO;
  TEMPORARY INTEGER J;
  I = I + 1;
  J = 2I;
END;

WRONG:

DO;
  TEMPORARY INTEGER J INITIAL(3);
  ...
END;

Temporaries cannot be initialized

RIGHT:

DO;
  TEMPORARY INTEGER J;
  J = 3;
END;
DATA STORAGE AND ACCESS (CONTINUED)

WRONG:

DO;
  STRUCTURE Q:
    1 A SCALAR,
    1 B SCALAR;
  TEMPORARY R Q-STRUCTURE;
  
  END;

RIGHT:

STRUCTURE Q:
  1 A SCALAR,
  1 B SCALAR;

  ..

  DO;
  TEMPORARY R Q-STRUCTURE;
  
  END;

IN DECLARE GROUP OF ANY OUTER BLOCK

WRONG:

DO;
  TEMPORARY INTEGER I;
  
  DO;
    TEMPORARY INTEGER I;
    
    END;

END;

TWO I'S IN SAME DO...END GROUP
DATA STORAGE AND ACCESS (CONTINUED)

RIGHT:

DO;
  TEMPORARY INTEGER I;
  ...
  DO;
  TEMPORARY INTEGER I1;
  ...
  END;
END;

THIS IS OK!

DO;
  TEMPORARY MATRIX M1;
  ...
  END;

2 SEPARATE (NON-NESTED DO...END GROUPS)

NOTE: IN A CASE LIKE THIS, NOT ONLY WILL M1 AND M2 BE ALLOCATED IN THE STACK, BUT M1 AND M2 WILL QUITE LIKELY OCCUPY THE SAME LOCATIONS.
DATA STORAGE AND ACCESS (CONTINUED)

THIS IS WRONG:

DO CASE J;
    TEMPORARY MATRIX M1;
    DO;
        
        CASE 1
        END;
    DO;
        
        CASE 2
        END;
    END;

BUT, THIS IS OK!

DO CASE J;
    DO;
        TEMPORARY MATRIX M1;
        
        END;
    END;
    DO;
        
        END;
    END;
WRONG:

PROC: PROCEDURE
    DECLARE I INTEGER;
    . . .
    DO FOR TEMPORARY I = 1 TO 5;
    . . .
    END;

TEMPORARIES SCOPE INTO NESTED DO...END GROUPS ALSO ---

DO;
    TEMPORARY Z ARRAY(10) INTEGER;
    . . .
    Z$1 = 3;
    . . .
    (DO FOR TEMPORARY I = 1 TO 10;
    IF Z$I = 0 THEN . . .
    END;
    . . .
    END;}
DATA STORAGE AND ACCESS (CONTINUED)

TEMPORARIES ALSO SCOPE INTO NESTED BLOCKS BUT SINCE TEMPORARIES ARE IN THE STACK WE HAVE THE SAME STACK "WALK-BACK" PROBLEM THAT EXISTS FOR PARAMETERS.

EXAMPLE:

```
DO;
    TEMPORARY MATRIX M;
    
    CALL PROC_A;
    
    PROC_A: PROCEDURE;
    
    IF M$(2,3) = 0 THEN . . .
    
    CLOSE PROC_A;

END;
```

M IS VISIBLE BECAUSE WE ARE STILL IN THE DO . . . END GROUP WHERE M WAS DEFINED. A STACK WALK-BACK LOOP WILL BE GENERATED, HOWEVER.
LOCK GROUPS

The concept of a lock group is linked to that of an update block. This is due to the fact that a data item which has been declared to be locked can only be accessed (referenced or modified) within an update block.

Although any declared data item may have a lock group specification, in practice only certain compool variables will need such protection.

The current implementation allows for 15 separate lock groups. Controlling crucial variables on a group basis turns out to be far more practical than protecting them individually.

If a variable is to be protected then it is given a lock specification at the time it is declared.
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE:

DECLARE WVECT VECTOR(3),
QVECT VECTOR(3) LOCK(7),
RVECT VECTOR(3) LOCK(4),

HERE WVECT IS UNPROTECTED, QVECT IS LOCKED AND BELONGS TO GROUP 7
AND RVECT IS LOCKED AND BELONGS TO GROUP 4. WVECT CAN BE REFERENCED
OR ASSIGNED ANYWHERE. QVECT AND RVECT CAN ONLY BE ACCESSED WITHIN
AN UPDATE BLOCK;

FOR THE SAKE OF FLEXIBILITY, A VARIABLE CAN BE DECLARED AS LOCK(*)
THIS TECHNICALLY SIGNIFIES THAT THE VARIABLES BELONGS TO ALL LOCK
GROUPS SIMULTANEOUSLY.

EXAMPLE:

DECLARE VEHICLE_POSITION VECTOR DOUBLE LOCK(*);
DATA STORAGE AND ACCESS (CONTINUED)

THE FOLLOWING EXAMPLES ILLUSTRATE THE POSITIONING OF THE CONSTRUCT WITHIN DECLARATIONS:

EXAMPLES:

| DECLARE I INTEGER DOUBLE LOCK(3);    |
| DECLARE S SCALAR INITIAL(5.5) LOCK(*); |
| DECLARE V VECTOR(3) LOCK(1) INITIAL(0); |
| DECLARE B ARRAY(1000) BOOLEAN LOCK(*); |
| STRUCTURE Q DENSE:                 |
|   1 QI INTEGER,                    |
|   1 QS SCALAR,                     |
|   1 QB BIT(16);                   |
| DECLARE ZQ Q-STRUCTURE(20) LOCK(3); |

UPDATE BLOCKS

AN UPDATE BLOCK IS AN EXPLICITLY DELIMITED BODY OF CODE WHEREIN LOCKED DATA MAY BE REFERENCED OR MODIFIED. AN UPDATE BLOCK SUPERFICIALLY LOOKS LIKE ANY OTHER HAL BLOCK.

label: UPDATE

body

CLOSE label;
THE UPDATE BLOCK, HOWEVER, IS UNIQUE IN SEVERAL RESPECTS:

1) THE BLOCK LABEL IS OPTIONAL. IF IT IS OMITTED THE COMPILER AUTOMATICALLY GENERATES LABELS, I.E.

   $UPDATE1, $UPDATE2, ETC.

2) UPDATE BLOCKS CANNOT BE CALLED, INVOKED, OR SCHEDULE. INSTEAD, THEY ARE EXECUTED WHEN THEY ARE ENCOUNTERED IN THE PATH OF EXECUTION.

NOTE: ALTHOUGH ONE "FALLS INTO" UPDATE BLOCKS (AND THAT IS WHY A LABEL IS OPTIONAL) THEY STILL ARE SEPARATE CSECTS (CONTROL SECTIONS) LIKE ANY OTHER BLOCK AND THUS THERE IS A HIDDEN BRANCH TO THE BLOCK WHICH LOOKS LIKE A NORMAL PROCEDURE CALL. THIS CAN BE SEEN IN THE GENERATED CODE.
DATA STORAGE AND ACCESS (CONTINUED)

3) A BLOCK LABEL, IF THE USER SUPPLIES ONE, MAY BE CONSIDERED AS A STATEMENT LABEL, I.E. IT IS POSSIBLE TO USE A GO TO STATEMENT TO GET TO AN UPDATE BLOCK.

```
I = I + 1;
IF I < 0 THEN GO TO ENTER;
J = J + 1;
ENTER: UPDATE;
E
M = M + U,U^T N;
CLOSE ENTER;
```

4) AT THE BEGINNING OF THE EMMITTED CODE FOR AN UPDATE BLOCK THERE IS A SUPERVISOR CALL (SVC) TO THE FCOS DESCRIBING WHICH LOCK GROUPS WILL BE USED BY THE UPDATE BLOCK. EXECUTION MAY BE HELD UP AT THIS POINT IF ONE OR MORE OF THE REQUESTED GROUPS ARE ALREADY IN USE.
DATA STORAGE AND ACCESS (CONTINUED)

5) AT THE END OF THE EMMITTED CODE FOR AN UPDATE BLOCK THERE IS AN SVC WHICH INFORMS THE FCOS THAT THE LOCK GROUPS ARE NOW FREE.

6) AN UPDATE BLOCK CAN HAVE DECLARED LOCAL DATA JUST LIKE ANY OTHER BLOCK.

7) THE FOLLOWING (UNUSUAL) RESTRICTIONS APPLY, HOWEVER:
   - NO I/O STATEMENTS OF ANY KIND ARE LEGAL, E.G. READ, WRITE, ETC. (HAL/S WILL NOT STOP %SVC, HOWEVER).
   - MOST REAL-TIME STATEMENTS ARE ILLEGAL, NAMELY: SCHEDULE, WAIT, CANCEL, TERMINATE, AND UPDATE PRIORITY.
DATA STORAGE AND ACCESS (CONTINUED)

- An update block can have nested procedures and functions but no update blocks or task blocks.
- The only procedure or function invocations which are legal are those referencing procedure or function blocks defined within it.

Update blocks should generally be as fast as possible since they tie up lock groups that may keep more critical update blocks from being executed.

Nor should update blocks be too frequent since there is not only the overhead of 2 FCOS services but all GPC's are usually synched upon entry to an update block.
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE:

UPDATE;
DECLARE I INTEGER,
    S SCALAR;
    V = V/S;
.
WRITE(6) V; ← illegal
.
INNER: PROCEDURE
   }
   body of procedure

CLOSE INNER;
.
CALL INNER;
CALL OUTER; ← illegal - outer not defined in update block
UPDATE PRIORITY ALPHA TO 50; ← illegal
.
.
CLOSE;
NOTE:  THE COMPILER AUTOMATICALLY INFORMS FCOS ON WHICH LOCK GROUPS
TO LOCK BASED ON WHICH LOCKED VARIABLES ARE REFERENCED.

FURTHERMORE, IF LOCKED DATA IS ONLY REFERENCED (NOT
ASSIGNED) FCOS IS INFORMED OF THIS ALSO -- ALTHOUGH
REFERENCE-ONLY UPDATE BLOCKS ARE NOT CURRENTLY GIVEN ANY
FAVORED TREATMENT.
EXECUTION OF UPDATE BLOCKS

THE BEHAVIOR OF PROCESSES ON ENCOUNTERING UPDATE BLOCKS HAS ALREADY BEEN DESCRIBED IN THIS SECTION, BUT ONLY SUPERFICIALLY BY EXAMPLE. THIS BEHAVIOR IS NOW RE-EXAMINED IN MORE DETAIL.

THE SIMPLEST CASE IS THAT OF TWO PROCESSES WISHING TO USE DATA ITEMS FROM THE SAME LOCK GROUP. EACH PROCESS HAS TO EXECUTE AN UPDATE BLOCK TO USE THE PROTECTED DATA ITEMS. THE FOLLOWING ACTIVITY TAKES PLACE:

- IF BOTH OF THE PROCESSES REQUIRE DATA ITEMS FROM THE SAME LOCK GROUP TO BE MODIFIED THEN THE FIRST PROCESS TO ENTER ITS UPDATE BLOCK MUST COMPLETE EXECUTION OF IT BEFORE THE SECOND PROCESS CAN ENTER ITS OWN UPDATE BLOCK. THE RTE PLACES THE SECOND PROCESS IN A WAITING STATE FOR THIS PERIOD OF TIME.

- IF ONE OR BOTH OF THE PROCESSES ONLY REQUIRE TO REFERENCE THE DATA THEN IN SOME IMPLEMENTATIONS OF HAL/S, THE BEHAVIOR OF THE RTE WILL BE THE SAME AS BEFORE. ALTERNATIVELY, IN OTHER IMPLEMENTATIONS, TO REDUCE THE SECOND PROCESS' WAITING TIME, THE RTE MAY ALLOW PARTIAL OVERLAP IN EXECUTION OF THE UPDATE BLOCKS, CONSISTENT WITH EXCLUSIVE USE OF DATA BY THE PROCESS MODIFYING IT.
DATA STORAGE AND ACCESS (CONTINUED)

IF THE TWO PROCESSES WISH TO USE DATA FROM MORE THAN ONE LOCK GROUP, THE RTE TRACKS THE USE OF EACH LOCK GROUP IN THE ABOVE WAY. IF ONE OR BOTH PROCESSES USE DATA PROTECTED BY LOCK(*), THEN THE SITUATION IS EQUIVALENT TO ONE IN WHICH THE PROCESS OR PROCESSES WISH TO USE DATA IN EVERY LOCK GROUP.

IF DATA IS SHARED BY MORE THAN TWO PROCESSES, THEN ALL PROCESSES EXCEPT ONE ARE PUT IN A WAITING STATE BY THE RTE. THE EVENTUAL ORDER IN WHICH THE PROCESSES COMPLETE EXECUTION OF THEIR UPDATE BLOCKS WILL DEPEND ON THE CONTENTS OF THE PROCESS QUEUE AND THE RELATIVE PRIORITY OF THE PROCESSES.
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE

IN SOME REAL TIME APPLICATION, IT IS REQUIRED THAT A PROCESS ALPHA PRINT THE VALUES OF A COVARIANCE MATRIX M ONCE EVERY 19 SECONDS. THE VALUES ARE UPDATED ONCE EVERY 1.5 SECONDS BY A SECOND PROCESS BETA. THE IMPLEMENTATION MUST GUARANTEE THAT A PARTIALLY UPDATED COVARIANCE MATRIX NOT BE PRINTED.

THE COVARIANCE MATRIX M IS DECLARED THUS:

```
DECLARE M MATRIX(3,3) LOCK(1);
```

TWO TASK BLOCKS CORRESPONDING TO ALPHA AND BETA ARE SHOWN BELOW:
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE CONTINUED:

\[
\begin{align*}
\text{ALPHA: TASK;} \\
\text{DECLARE M_LOCAL MATRIX(3,3);} \\
U_1: \text{UPDATE;} \\
\quad M_{\text{LOCAL}} = M; \\
\text{CLOSE U_1;} \\
\text{WRITE(6) 'COVARIANCE=', M_LOCAL;} \\
\text{CLOSE ALPHA;} \\
\end{align*}
\]

\[
\begin{align*}
\text{BETA TASK;} \\
\text{DECLARE VT VECTOR(3);} \\
U_2: \text{UPDATE;} \\
V = (\phi M \phi' + QA)z; \\
M = V V'(QB + z.V); \\
\text{CLOSE U_2;} \\
\text{CLOSE BETA;} \\
\end{align*}
\]
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE CONTINUED:

ALPHA: TASK;
DECLARE M_LOCAL MATRIX(3,3);
U1: UPDATE;
   M_LOCAL = M;
CLOSE U1;
WRITE(6) 'COVARIANCE=', M_LOCAL;
CLOSE ALPHA;

BETA TASK;
DECLARE VT VECTOR(3);
U2: UPDATE;
   E T
   V = (PHI M PHI + QA)Z;
   M = V V/(QB + Z.V);
CLOSE U2;
CLOSE BETA;
EXAMPLE CONTINUED:

PROCESSES ALPHA AND BETA COULD BE CREATED BY INVOKING THESE TASK BLOCKS WITH CYCLIC SCHEDULE STATEMENTS OF THE FOLLOWING FORM:

| SCHEDULE ALPHA PRIORITY(10), REPEAT EVERY(19); |
| SCHEDULE BETA PRIORITY(20), REPEAT EVERY(1.5); |

THE FOLLOWING DIAGRAM SHOWS THE STATE TRANSITIONS OF THE PROCESSES:
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE CONTINUED:

NOTE THAT IF IN THIS EXAMPLE PROCESS SWAPS OCCURRED ONLY ON STATEMENT BOUNDARIES, UPDATE BLOCKS WOULD NOT BE NEEDED SINCE ALPHA COULD NOT EVER BE BROUGHT INTO EXECUTION WITH COVARIANCE MATRIX M PARTLY UPDATED,
DATA STORAGE AND ACCESS (CONTINUED)

LOCKED ASSIGN ARGUMENTS

The rule that locked data items can only appear in update blocks has one sole exception: it is possible for locked data items to appear as assign arguments in procedure invocations. This provides the ability to "parameterize" update blocks, as will be shown in an ensuing example.

The following rules govern the passage of locked assign arguments:
1. If the argument is a data item belonging to lock group $n$, then the corresponding parameter must be declared Lock($n$) or Lock(*).

2. If the argument is a data item belonging to all lock groups, the corresponding parameter must be declared Lock(*).

3. Argument and parameter must also match in the senses described, as applicable.
EXAMPLE:

DECLARE A SCALAR LOCK(1),
B SCALAR LOCK(2),
C SCALAR LOCK(*);
PICK: PROCEDURE (P) ASSIGN (Q, R);
   DECLARE P SCALAR,
     Q SCALAR LOCK(1),
     R SCALAR LOCK(*);

body of procedure

CLOSE PICK;

For the above procedure definitions and declarations, the following invocations are legal:

CALL PICK(1.0) ASSIGN (A, B);
CALL PICK(2.0) ASSIGN (A, C);

The following are illegal:

CALL PICK(N) ASSIGN (A, B);  // locked data item as input argument
CALL PICK(3.0) ASSIGN (C, B);  // unmatched lock group
DATA STORAGE AND ACCESS (CONTINUED)

EXAMPLE CONTINUED:

The procedure PICK may contain an update block changing the values of Q and R:

```
PICK: PROCEDURE(P) ASSIGN(Q,R);  
DECLARE P SCALAR, 
    Q SCALAR LOCK(1), 
    R SCALAR LOCK(*); 
...
U: UPDATE; 
    Q = Q + P; 
    R = R - P; 
CLOSE U; 
...
CLOSE PICK; 
```

PICK may be invoked with different locked assign arguments, thus effectively parameterizing the update block.

```
CALL PICK(1) ASSIGN(A,B);  
CALL PICK(2) ASSIGN(A,C);  
```

updates A and B 
updates A and C
DATA STORAGE AND ACCESS (CONTINUED)

FURTHER NOTES:

1) STRUCTURES MAY BE LOCKED:

   STRUCTURE Q:
   1 A,
   2 A1 SCALAR,
   2 A2 SCALAR,
   1 B,
   2 B1 INTEGER,
   2 B2 INTEGER;
   DECLARE Q Q-STRUCTURE LOCK(5);

   IN WHICH CASE AN UPDATE BLOCK IS NEEDED TO ACCESS ANY PART OF THE
   STRUCTURE.

2) THE KEYWORD LOCK MAY NOT APPEAR WITHIN THE TEMPLATE ITSELF,
   I.E., EITHER THE ENTIRE STRUCTURE IS LOCKED OR NO PIECE OF IT IS.

   THIS TURNS OUT TO BE AN UNFORTUNATE RESTRICTION IN THE CASE OF
   NAME VARIABLES (AS WE WILL SEE LATER), I.E. THE FOLLOWING IS
   ILLEGAL (ALTHOUGH LOGICALLY REASONABLE).
DATA STORAGE AND ACCESS (CONTINUED)

STRUCTURE Q:
1 A SCALAR,
1 NM NAME MATRIX LOCK(3),
1 V VECTOR;

3) A DISADVANTAGE OF LOCK GROUPS IS THAT AN UPDATE BLOCK IS LEGALLY
REQUIRED EVEN WHEN THE SOFTWARE DESIGN GUARANTEES THAT CERTAIN
"SAFE" TIMES EXIST WHEN ONLY ONE PROCESS WILL BE ACCESSING THE
LOCKED DATA.

4) THE NEW %COPY FACILITY DOES ALLOW LOCKED DATA TO BE COPIED OR
UPDATED WITHOUT THE NEED FOR AN UPDATE BLOCK.
EXCLUSIVE PROCEDURES/FUNCTIONS

AN EXCLUSIVE PROCEDURE OR FUNCTION (USUALLY A COMSUB) IS ONE WHICH POSSESSES A SPECIAL FCOS INTERFACE THAT GUARANTEES THAT ONLY ONE PROCESS AT A TIME MAY ENTER THE BLOCK, I.E. THE FIRST PROCESS TO ENTER THE BLOCK LOCKS OUT ALL OTHER POTENTIAL CALLERS UNTIL IT HAS FINISHED AND HAS EXITED THE BLOCK.

DEFINING AN EXCLUSIVE PROCEDURE

THE FORM OF THE OPENING STATEMENT OF AN EXCLUSIVE PROCEDURE IS AS SHOWN BELOW:

\[
\text{label: PROCEDURE}(i_1, i_2, \ldots) \text{ ASSIGN}(a_1, a_2, \ldots) \text{ EXCLUSIVE;}
\]

1. \textit{label} is a legal HAL/S identifier constituting the procedure name.
2. \(i_1, i_2, \ldots\) and \(a_1, a_2, \ldots\) are lists of input and assign parameters as described.
3. The keyword \text{EXCLUSIVE} designates an exclusive procedure.
EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

NOTE: ALTHOUGH IT IS FAR EASIER TO MAKE A BLOCK EXCLUSIVE THAN REENTANT, EACH EXECUTION OF AN EXCLUSIVE BLOCK CARRIES THE OVERHEAD OF 2 FCOS SUPERVISOR CALLS.

EXCLUSIVE BLOCKS ALSO MAY RESULT IN TEMPORARY "LOCK-OUT" OF MORE CRITICAL PROCESSES.

Example:

```
P: PROCEDURE(A) EXCLUSIVE;
DECLARE A SCALAR;
{
    procedure body
}
CLOSE P;
```

This is all it takes to make a block exclusive.

The template corresponding to an exclusive external procedure must also bear the keyword EXCLUSIVE.
Example:

The template corresponding to

```
P: PROCEDURE(A) EXCLUSIVE;
   DECLARE A SCALAR;
```

```
procedure body
```

`CLOSE P;`

would be:

```
P: EXTERNAL PROCEDURE(A) EXCLUSIVE;
   DECLARE A SCALAR;
   CLOSE P;
```
DEFINING AN EXCLUSIVE FUNCTION

THE FORM OF THE OPENING STATEMENT OF AN EXCLUSIVE FUNCTION IS AS SHOWN BELOW:

```
label: FUNCTION(l1, l2, ...) attributes EXCLUSIVE;
```

1. `label` is a legal HAL/S identifier constituting the function name.

2. `l1, l2, ...` is a list of input parameters as described earlier.

3. `attributes` defines the type and, where applicable, precision of the function.

4. The keyword EXCLUSIVE designates an exclusive function.
EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

Example:

F: FUNCTION BOOLEAN EXCLUSIVE;

function body

CLOSE F;

The template corresponding to an exclusive external function must also bear the keyword EXCLUSIVE.

Example:

The template corresponding to:

F: FUNCTION BOOLEAN EXCLUSIVE;

function body

CLOSE F;

would be:

F: EXTERNAL FUNCTION BOOLEAN EXCLUSIVE;
CLOSE F;
EXCLUSIVE PROCEDURES/FUNCTIONS (CONTINUED)

BEHAVIOR OF EXCLUSIVE PROCEDURES AND FUNCTIONS

If an exclusive procedure or function is in use by a process A, and a process B tries to invoke it, then the RTE places process B in the waiting state until process A returns from its use.

Example:

Two processes ALPHA and BETA can invoke the following procedure:

```
P: PROCEDURE EXCLUSIVE;
    procedure body
    CLOSE P;
```
Suppose that ALPHA invokes P first and during its execution, BETA tries to invoke it. The state transitions for this situation is shown below:

- **ALPHA**
  - Executing
  - Ready
  - Waiting

- **BETA**
  - Executing
  - Ready
  - Waiting

- **Other unrelated transitions**

**Transitions**

- **ALPHA enters P**
- **ALPHA leaves P**
- **BETA tries to enter P**
- **BETA allows BETA to enter P**
REENTRANT PROCEDURES/FUNCTIONS

- DUE TO THE HAL/S STACK MECHANISM AND THE METHOD OF PARAMETER PASSAGE AND PROCEDURE CALL, EVERY PROCEDURE/FUNCTION IS ALMOST REENTRANT.

- TO MAKE A BLOCK LEGALLY REENTRANT IT IS NECESSARY TO APPEND THE KEYWORD `REENTRANT` TO THE BLOCK HEADER.

- TO MAKE A BLOCK ACTUALLY REENTRANT REQUIRES MORE WORK AND A LOT OF CARE!

REENTRANCY MEANS, OF COURSE, THAT TWO OR MORE PROCESSES (PROGRAMS OR TASKS) MAY BE "SIMULTANEOUSLY" EXECUTING THE BLOCK.
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

DEFINING A REENTRANT PROCEDURE

THE FORM OF THE OPENING STATEMENT OF A REENTRANT PROCEDURE IS SHOWN BELOW:

```
label: PROCEDURE(l, l2, ...) ASSIGN(a1, a2, ...) REENTRANT;
```

1. `label` is a legal HAL/S identifier constituting the procedure name.
2. `l, l2`, and `a1, a2, ...` are lists of input and assign parameters.
3. The keyword `REENTRANT` indicates that the procedure is to be considered reentrant.

Example:

```
P: PROCEDURE REENTRANT

<table>
<thead>
<tr>
<th>procedure body</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLOSE P;</td>
</tr>
</tbody>
</table>
```

If `P` were an external procedure, the corresponding template would be:

```
P: EXTERNAL PROCEDURE REENTRANT;
CLOSE P;
```
DEFINING A REENTRANT FUNCTION

The form of an opening statement of a reentrant function is shown below:

```%
| label: FUNCTION(l^1, l^2,...) attributes REENTRANT;
1. Label is a legal HAL/S identifier constituting the function name.
2. l^1, l^2,... is a list of input parameters as described in Section 11.2.
3. attributes defines the type and, where applicable, precision of the function as described in Section 11.2.
4. The keyword REENTRANT indicates that the function is to be considered reentrant.
```

The template corresponding to an external reentrant function must also possess the keyword REENTRANT.
Example:

```
P: FUNCTION MATRIX(4,4) REENTRANT;

<table>
<thead>
<tr>
<th>function body</th>
</tr>
</thead>
</table>

CLOSE P;
```

If F were an external function, the corresponding template would be:

```
P: EXTERNAL FUNCTION MATRIX(4,4) REENTRANT;

CLOSE P;
```
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

BEHAVIOR OF REENTRANT PROCEDURES AND FUNCTIONS

If a reentrant procedure or function is in use by a process A, and a process B tries to invoke it, the RTE allows the invocation to proceed without restriction.

Example:

Two processes, ALPHA and BETA, can invoke the following procedure:

```
P: PROCEDURE REENTRANT;

procedure body

CLOSE P;
```

Suppose that ALPHA invokes P first and during its execution, BETA invokes it. The state transitions for this situation is as shown below (compare corresponding example for exclusive procedure):
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

- executing

ALPHA enters P

other unrelated transitions ready

ALPHA leaves P

BETA enters P

other unrelated transitions

ready

waiting

Beta leaves P

ALPHA and BETA both executing P
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

TRUE REENTRANCY GENERALLY REQUIRES THAT EVERY PROCESS ENTERING
A REENTRANT BLOCK BE GIVEN ITS OWN COPY OF ANY LOCAL DATA THE
BLOCK MAY USE. THIS IS ACCOMPLISHED BY SOMEHOW CAUSING THE
LOCAL DATA TO BE ESTABLISHED IN THE STACK OF THE CALLING PROCESS
(AKIN TO USING A GETMAIN'ED AREA) RATHER THAN HAVING A SINGLE
COPY OF THE DATA PERMANENTLY ALLOCATED IN A STATIC AREA.

NOTE: PARAMETERS PASSED TO THE BLOCK ARE ALREADY IN THE STACK
OF THE CALLING PROCESS SO NO PROBLEM EXISTS HERE.

LOCAL DATA WHICH IS TO BE DYNAMICALLY ALLOCATED (FROM THE
CALLER'S STACK) MUST BE DECLARED WITH THE AUTOMATIC ATTRIBUTE.
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

In a non-reentrant block the keyword AUTOMATIC, in conjunction
with an initial list, simply meant that code was to be automatically generated by the compiler to cause the indicated
initialization at each entry to the block. It thus made no sense to not have an initial list.

With a reentrant block, however, AUTOMATIC additionally means
that the allocation of the data is to be made from the work
area (stack) provided by the caller. In this context AUTOMATIC
makes sense even without an initial list.

Note: In most non-contrived cases, all data declared local to a
reentrant procedure should be declared as AUTOMATIC. The default
(which is STATIC) would mean that all users are asynchronously
modifying the same memory location.
EXAMPLES:

IN THE REENTRANT PROCEDURE:

```
P: PROCEDURE(A) ASSIGN(B) REENTRANT;
   DECLARE A VECTOR;
   DECLARE B SCALAR;
   DECLARE V VECTOR(3) AUTOMATIC;
   ...
   V = VECTOR(B, 0, 0);
   B = V.A;
   CLOSE P;
```

NOTE: A AND B, BEING PARAMETERS, ARE ALREADY IN A DYNAMICALLY ALLOCATED AREA.
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

IN CONTRAST, SUPPOSE THE NUMBER OF TIMES A REENTRANT
PROCEDURE IS INVOKED IS REQUIRED TO BE KNOWN AND PRINTED
EVERY 10 INVOCATIONS. IN THIS UNUSUAL, AND RATHER ARTI-
FICIAL CASE, IT WOULD BE APPROPRIATE TO USE A LOCAL
DATA ITEM NOT DECLARED AUTOMATIC:

```
| P2: PROCEDURE(A,B) ASSIGN(C) REENTRANT;          |
| DECLARE VECTOR, A, B, C;                         |
| DECLARE COUNT INTEGER INITIAL(0);               |
| COUNT = COUNT + 1;                              |
| IF REMAINDER(COUNT, 10) = 0 THEN                 |
|   WRITE(6) 'NUMBER OF ENTRIES='| |COUNT;  |
|                                                 |
| CLOSE P2;                                       |
```

IN AN IMPLEMENTATION WHERE PROCESS SWAPS CAN ONLY OCCUR AT THE
END OF EVERY EXECUTABLE STATEMENT, THE CODE SHOWN WOULD MAINTAIN
A CORRECT COUNT OF THE NUMBER OF INVOCATIONS.
REENTRANT PROCEDURES/FUNCTIONS (CONTINUED)

HISTORICAL NOTE:

AT VARIOUS TIMES IN THE PAST IT HAS BEEN SUGGESTED THAT THE DEFAULT BE AUTOMATIC FOR DATA DECLARED IN A REENTRANT PROCEDURE INSTEAD OF THE NORMAL STATIC. THIS HAS NOT BEEN DONE.

FURTHER CONSIDERATIONS FOR ACHIEVING REENTRANCY:

1) PROCEDURES AND FUNCTIONS DEFINED WITHIN A REENTRANT BLOCK MUST ALSO POSSESS THE REENTRANT ATTRIBUTE IF THEY TO POSSESS LOCAL DATA WHICH IS REQUIRED TO PARTICIPATE IN THE REENTRANCY.

2) UPDATE BLOCKS WITHIN A REENTRANT BLOCK MUST NOT DECLARE ANY LOCAL DATA (STATIC OR AUTOMATIC).

3) A PROCEDURE OR FUNCTION CALLED BY A REENTRANT BLOCK MUST ITSELF ALSO BE REENTRANT.
HALS CSECTS

IN GENERAL, A HALS COMPILATION RESULTS IN THE GENERATION OF A
NUMBER OF CSECTS (CONTROL SECTIONS). WE MIGHT CALL THIS A
CSECT FAMILY SINCE ALL CSECTS EMITTED BY A SINGLE COMPILATION
WILL SHARE 6 (OR LESS) LETTERS -- THE GENERIC NAME.

THE GENERIC NAME IS ESTABLISHED BY TAKING THE FULL NAME OF THE
COMPILATION BLOCK ITSELF, REMOVING ALL UNDERSCORES, AND THEN
TAKING THE FIRST 6 CHARACTERS OF WHAT REMAINS (RIGHT-PADDING WITH BLANKS IF NECESSARY). FOR THIS REASON, COMPIATION UNIT
NAMES SHOULD BE CHOSEN SUCH THAT THE GENERIC NAMES WILL BE
UNIQUE.

EXAMPLE:

CG1_GNC: COMPOOL;

CLOSE;

THE GENERIC NAME WILL BE

CG1GNC
ASSUMING THE GENERIC NAME IS NNNNNN HAL/S PRODUCES THE FOLLOWING CSECTS:

(MAJOR DECLARED DATA CSECTS)

#PNNNNNNN  A COMPOOL DATA CSECT

#DNNNNNNN  A PROGRAM OR COMSUB DATA CSECT

(OUTER CODE BLOCKS)

$ONNNNNNN  A PROGRAM CODE BLOCK

#CNNNNNNN  A COMSUB CODE BLOCK

(INTERNAL CODE BLOCKS)

 appart NNNNNN  INTERNAL PROCEDURES, FUNCTIONS, UPDATE BLOCKS.

  a = (A-Z)
  n = (1-9)

$CNNNNNNN  TASK CODE BLOCKS.

  C = 1-9, then A-Z
HAL/S CSECTs (CONTINUED)
(360 ONLY - FSIM TYPE CSECTs)

#FNNNNNNN
  FSIM CSECT
  (ALSO CONTAINS LITERALS AND ADCONS)
#TNNNNNNN
  COST/USE CSECT
  (FC ONLY)
#ZNNNNNNN
  ZCON CSECT FOR COMSUB OR REMOTE DATA
#ENNNNNNN
  PROCESS DIRECTORY ENTRY (PROGRAMS AND
  TASKS ONLY)
#XNNNNNNN
  CONTROL AREA FOR EXCLUSIVE BLOCK(S)
#RNNNNNNN
  DATA AREA FOR REMOTE DATA
HAL/S CSECTS (CONTINUED)

IN ADDITION, HAL/S SPECIFIES INFORMATION TO HALLINK (360) AND THE AP-101 LINKAGE EDITOR THAT ENABLES THE PRODUCTION OF STACK CSECTS (ONE PER PROCESS).

\[\text{aONNNNNN} \quad \text{PROGRAM STACK} \]
\[\text{aONNNNNN} \quad \text{TASK STACK} \]
\[\text{C = 1-9, THEN A-Z} \]

ALSO, THE COMPILER GENERATES A BLOCK TEMPLATE

\[\text{aONNNNNN} \]

AND A SIMULATION DATA FILE (SDF)

\[\text{##NNNNNN} \]
HAL/S CSECTS (CONTINUED)

FURTHERMORE, IN AN FC LOAD MODULE THE FOLLOWING TYPES OF CSECTS WILL BE SEEN:

<table>
<thead>
<tr>
<th>Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aaNNNNNN</td>
<td>HAL/S LIBRARY ROUTINE OR NON-HAL (E.G. FCOS) ROUTINE.</td>
</tr>
<tr>
<td>ZCON</td>
<td>a = (A-Z)</td>
</tr>
<tr>
<td>#QNNNNNN</td>
<td>ZCON FOR 'LIBRARY ROUTINE.</td>
</tr>
<tr>
<td>#LNNNNNN</td>
<td>DATA FOR A LIBRARY ROUTINE.</td>
</tr>
<tr>
<td>#SNNNNN</td>
<td>SECTOR # LIBRARY ROUTINE.</td>
</tr>
</tbody>
</table>
DATA CSECTS

#PNNNNNNN  COMPOOL DATA CSECT

A COMPOOL compilation results in the generation of a single
CSECT and its contents are the same for both FC and 360.

A #P contains COMPOOL declared data and users can control
the order of data allocated within it by means of the RIGID
keyword.

KEY POINTS:
1) REPLACE MACROS TAKE UP NO SPACE.
2) ENTRY POINTS CAN BE CREATED BY THE USER (VIA THE EQUATE EXTERNAL
   FACILITY) WHICH ALLOW NON-HAL MODULES TO ACCESS COMPOOL DATA.

EXAMPLE

DECLARE ARR ARRAY(10) SCALAR DOUBLE;
EQUATE EXTERNAL EXTNNAME TO ARR$4;

AN EQUATE EXTERNAL TAKES UP NO SPACE IN THE #P. WHAT IT DOES IS ....
TO GENERATE AN ESD RECORD IN THE OBJECT DECK WHICH MARKS THE
SPECIFIED VARIABLE AS AN ENTRY POINT (NAMED BY EXTNAME IN THE
PREVIOUS EXAMPLE) OF THE CSECT IN WHICH THE VARIABLE IS
LOCATED.

THE EQUATE EXTERNAL IS DESIGNED FOR EXTERNAL (NON-HAL) USE ONLY.
IT IS ALSO NOT RESTRICTED TO COMPOOL DATA, BUT MAY BE USED WITH
ANY DECLARED (STATIC) DATA. THE EQUATE MAY POINT TO ANY PREVIOUSLY
DEFINED HAL VARIABLE AND COMPLEX SUBSCRIPTING IS ALLOWED PROVIDED
ALL SUBSCRIPTING CAN BE EVALUATED AT COMPILE TIME.

3) THE #P (LIKE ALL OTHER HAL/S DATA CSECTS) CONTAINS DATA WHICH
IS, IN GENERAL, MODIFIED. AS A RESULT THE ENTIRE #P IS UNPROTECTED
EVEN WHEN PORTIONS OF IT CONTAIN CONSTANT DATA. AT RUN-TIME, THEN,
CONSTANT DATA IS NO MORE SACRED THAN VARIABLE DATA.
DATA CSECTS (CONTINUED)

CONSTANT DATA

(A DIGRESSION)

IN A COMPOOL, ALL DECLARED DATA WITH INITIAL LISTS WILL BE PHYSICALLY ALLOCATED IN THE #P CSECT.

DATA DECLARED WITH A CONSTANT LIST, ON THE OTHER HAND, IS A MORE COMPLEX STORY:

DATA TYPES THAT CAN PARTICIPATE IN COMPILE-TIME EXPRESSION EVALUATION.

Scalars
- Integer
- Bit Strings
- Booleans
- Character Strings

"LITERALS"
DATA CSECTS (CONTINUED)

Thus, simple compool constant data is not given space in the #P -- such data is often (incorrectly) called literal data.

Since all of the source lines of the compool do get put in its template, however, the actual values of the literal data are visible to all compilations that include the compool.

Note: All such literal data is inaccessible (almost) to diagnostics.

Q. If literals are not in the #P, where do they go?

A. When a compool is included in a program, the compiler sees both its literals plus any similar literals that may have been declared in the program -- all are treated alike.

Then the following happens:
DATA CSECTS (CONTINUED)

1) If a literal is not used, it is simply flushed.

2) If the literal is used, it will be allocated in the #D CSECT (data area of the program/comsub) unless its use is so specialized that it actually gets built into an instruction.

3) If the literal is allocated in the #D then it may:
   A) Be of different type (e.g. the item was declared as an integer but usages were in a scalar context).
   B) Be of different precision or scaling (e.g. if PI is declared constant(3.14) but (2 * PI) is used, then the value stored will be 6.28).
   C) Share with another literal, i.e. two literals with exactly the same value will share the same space.
#DNNNNNNN PROGRAM/COMSUB DATA CSECT

SIMILAR TO #P
1) EQUATE EXTERNAL CAN BE USED.
2) CONTAINS ALL DATA DECLARE'D IN THE PROGRAM/COMSUB INCLUDING LOCAL DATA OF ALL INTERNAL BLOCKS.

BUT DIFFERENT:
1) RIGID CANNOT BE USED.
2) MAY CONTAIN A MIXTURE OF BOTH COMPOOL LITERALS (PROVIDED THEY ARE USED) AND ITS OWN LITERALS (AGAIN, ONLY IF USED).
3) ALSO CONTAINS SPECIAL DATA NEEDED BY FCOS OR THE COMPILER, E.G.
   ADDRESS CONSTANTS
   LOCAL BLOCK DATA AREAS

AGAIN, ALTHOUGH INVARIANT DATA MAY RESIDE IN THE #D, IT IS COMPLETELY UNPROTECTED FROM WRITES.
DATA CSECTS (CONTINUED)

aONNNNNN PROGRAM STACKS
a1NNNNNN TASK STACKS
;
@ZNNNNNN TASK STACKS

WE HAVE NOT YET DISCUSSED THE TASK BLOCK BECAUSE IT IS VIRTUALLY UNUSED IN THE CURRENT SHUTTLE DESIGN.

FOR THE CURRENT DISCUSSION WE WILL THEREFORE LIMIT ATTENTION TO HAL/S PROGRAMS AND THEIR ASSOCIATED STACK.

Q. WHAT IS A STACK?

A. A STACK IS A SPECIAL DATA CSECT WHICH IS DEDICATED TO A SINGLE PROCESS, I.E. HAL/S PROGRAM. THE STACK HAS DYNAMICALLY CHANGING CONTENTS -- THE SAME LOCATION WILL GENERALLY CONTAIN DIFFERENT DATA AS A FUNCTION OF TIME. THE STACK IS USED FOR THE FOLLOWING PURPOSES:
DATA CSECTS (CONTINUED)

(1) REGISTER SAVE AREAS
(2) PARAMETER PASSING
(3) SCRATCH SPACE NEEDED BY THE COMPILER
(4) USER DEFINED TEMPORARIES
(5) AND OTHER THINGS.
DATA CSECTS (CONTINUED)

SOME FACTS:

1) THE COMPILER NEVER GENERATES A STACK. STACKS ARE EITHER GENERATED AT LINK EDIT TIME (VIA HALLINK ON THE 360 AND THE AP-101 LINKAGE EDITOR ON THE FC) -- OR ARE DOLED OUT BY THE FCOS AT THE TIME A PROGRAM IS INITIATED.

2) THE COMPILER DOES INFORM THE LINKAGE EDITOR OF THE SIZE OF THE STACK FRAME REQUIRED BY EACH CODE CSECT. THIS IS DONE VIA SYM CARDS ISSUED IN THE OBJECT DECK.

3) A STACK IS DYNAMICALLY DIVIDED INTO STACK FRAMES. EACH CODE BLOCK (GENERALLY) HAS A STACK FRAME PROVIDED FOR IT WHILE IT IS IN EXECUTION. THE STACK FRAME IS ALWAYS THE SAME SIZE FOR A PARTICULAR CODE BLOCK BUT MAY OCCUPY DIFFERENT LOCATIONS IN THE STACK CSECT DEPENDING ON THE CALL CHAIN BY WHICH THE CODE BLOCK WAS REACHED.
DATA CSECTS (CONTINUED)

A STACK AT ONE POINT IN TIME:

![Diagram of a stack at one point in time]

AND LATER:

![Diagram of a stack later]

NOTE: IF THE LINKAGE EDITOR EVER CALCULATES A REQUIRED STACK LENGTH INCORRECTLY, THERE IS NOTHING AT FLIGHT TIME TO KEEP US FROM FALLING OFF THE END.
STACK SIZE CALCULATION

1) A SPECIAL SYM CARD IS GENERATED (FOR EACH CODE BLOCK) IN THE OBJECT MODULE DEFINING THE SIZE OF THE STACK FRAME REQUIRED BY THE CODE BLOCK (ON THE FC THE MINIMUM SIZE IS 18 HALFWORDS).

2) THE LINKAGE EDITOR PERFORMS AN ANALYSIS OF ALL CALLS AND FUNCTION INVOCATIONS (BUILDS A CALL TREE) AND CALCULATES THE SIZE NEEDED FOR EACH PROGRAM (PROCESS) STACK.

3) THE LINKAGE EDITOR THEN EITHER EMITS AN APPROPRIATELY NAMED STACK CSECT OF THE RIGHT LENGTH OR INFORMS THE FCOS (VIA THE #E CSECT) OF WHAT SIZE STACK WILL BE NEEDED WHEN THE PROGRAM IS SCHEDULED.

NOTE: ONLY PROGRAMS ARE GIVEN STACKS SINCE THEY ARE PROCESSES. (TAKS ARE TOO!) ANY INTERNAL OR EXTERNAL BLOCKS (COMSUBS) CALLED BY THE PROGRAM UTILIZE THE STACK OF THE PROGRAM -- AND THE LINKAGE EDITOR HAS TAKEN ALL OF THIS INTO ACCOUNT.
IF WE FOLLOW THE LONGEST PATH THE STACK FRAMES WILL LOOK LIKE:

<table>
<thead>
<tr>
<th></th>
<th>P1</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td></td>
<td>18</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>P2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>55</td>
<td>P3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>105</td>
<td>A1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>18</td>
<td>A3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>B1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>B2</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>100</td>
<td>B3</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Stack is fully utilized

NOTES:
1) IN ACTUAL EXECUTION A STACK WOULD RARELY BE COMPLETELY USED.
2) IT IS EASY TO SEE WHY RECURSION IS NOT ALLOWED.
DATA CSECTS (CONTINUED)

STACKS SAVE CORE BECAUSE …

1) LOCAL (TRANSIENT) SPACE NEEDS WITHIN A SINGLE PROCESS ARE SATISFIED FROM A REUSABLE AREA.

2) DEPENDING ON THE FCOS IMPLEMENTATION, STACKS THEMSELVES CAN SHARE THE SAME STORAGE IF THE PROCESSES CAN NEVER BE SIMULTANEOUSLY ACTIVE.

USERS CAN PUT SCRATCH DATA IN THE STACK VIA THE TEMPORARY FACILITY. THIS WOULD ONLY INCREASE THE SIZE OF THE STACK NEEDED IF THE CORE BLOCK INVOLVED IS IN THE MAXIMAL CHAIN — THAT IS, THE GREATEST AMOUNT OF STACK SPACE.
THE HAL/S STACK MECHANISM

THE HAL/S RUNTIME STACK PROVIDES:

- A SIMPLE MEANS FOR SAVING AND RESTORING ENVIRONMENTS FOR CALLS.
- A DYNAMIC RECORD OF ERROR ENVIRONMENTS.
- EFFICIENT TEMPORARY STORAGE ALLOCATION (COMPILER AND USER)
- THE MEANS BY WHICH SEPARATE PROCESSES MAY SIMULTANEOUSLY EXECUTE THE SAME CODE BLOCK AND YET MAINTAIN INDEPENDENT DATA.

EACH HAL/S PROCESS HAS ITS OWN STACK.

EACH STACK IS A CONTIGUOUS AREA OF MEMORY LARGE ENOUGH TO SERVE ITS PROCESS' NEEDS.
THE HAL/S STACK MECHANISM (CONTINUED)

A SINGLE PROCESS STACK

CURRENT STACK POINTER

PREVIOUS STACK FRAMES

- REGISTER SAVE AREA AND PARAMETERS
- ON ERROR DATA
- TEMPORARY SPACE FOR CURRENT BLOCK
- OTHER THINGS?

CURRENT STACK FRAME

AREA OF THE STACK BEING USED FOR CURRENTLY EXECUTING BLOCK WITHIN THIS PROCESS (PROGRAM, PROCEDURE, FUNCTION, ETC.).
THE HAL/S STACK MECHANISM (CONTINUED)

WHEN A JUMP TO A NEW BLOCK IS MADE:

CALL X(A) ASSIGN(B);

• THE STACK AREA ADDRESS IS INCREMENTED BY THE SIZE OF THE CURRENT STACK FRAME.
• A NEW STACK FRAME IS USED FOR THE NEWLY ENTERED ROUTINE.
• THE NEW STACK FRAME CONTAINS ALL THE INFORMATION NECESSARY TO RESTORE THE PREVIOUS ENVIRONMENT.
THE HAL/S STACK MECHANISM (CONTINUED)

WHEN THE CALLED ROUTINE IS FINISHED:

RETURN;

RESTORED STACK POINTER

STACK IS "POPPED"
BY RESTORING
PREVIOUS STACK
POINTER

OLD ENVIRONMENT RESUMED

DISCARDED STACK FRAME
(MAY BE USED BY ANOTHER CALL)
HAL/S STACK MECHANISM (CONTINUED)

- DATA DECLARED TO BE STATIC IN THE CURRENT BLOCK IS NOT ALLOCATED IN THE STACK AREA.

- REENTRANT PROCEDURES HAVE ANY AUTOMATIC DATA ALLOCATED IN THE STACK.

- ANY TEMPORARY DATA EITHER DECLARED (TEMPORARY STATEMENT) BY THE PROGRAMMER OR IMPLICITLY NEEDED BY THE COMPILER IS IN THE STACK AREA.

- THE COMPILER KNOWS EXACTLY HOW MUCH STACK AREA EACH CODE BLOCK REQUIRES THAT SIZE INFORMATION IS PLACED IN THE OBJECT DECK.

- THE HALLINK PROGRAM BUILDS A TREE OF ALL BLOCK REFERENCES.
  - RECURSION IS DETECTED HERE.
  - BY ADDING STACK REQUIREMENTS FOR EACH LIMB OF THE TREE, THE CALLING SEQUENCE WHICH REQUIRES THE MOST TOTAL STACK AREA IS FOUND.
  - A STACK OF PROPER SIZE IS CREATED AND MADE PART OF THE PROGRAM.
  - A SEPARATE CALCULATION IS MADE FOR EACH POTENTIAL PROCESS (PROGRAM OR TASK).
THE HAL/S STACK MECHANISM (CONTINUED)

- The mechanization of the stack requires an interface between the operating system and the generated code.
- How is stack location established? (Passed at initiation or loaded by code.)
- What is the stack layout? (Needed for error handling.)
- How are stack frames chained together? (Stack walk.)
- Other interfaces.
ASSEMBLY LANGUAGE ROUTINES MUST BE INTERFACED TO THE HAL/S SYSTEM

- OBJECT MODULE (RESULT OF ASSEMBLY) MUST BECOME PART OF HALLINK INPUT.
- HAND-WRITTEN TEMPLATE MUST BE INCLUDED IN COMPILATION OF HAL/S CALLER.

// EXEC HALSCLD
//HAL/SYSIN DD *
ASMSUB: EXTERNAL PROCEDURE (A);
DECLARE A INTEGER;
CLOSE AMSUB;
TEST: PROGRAM;
;
CALL ASMSUB(23);
;
CLOSE TEST;

//LKED.SYSIN DD <assembly object module>
ASSEMBLY LANGUAGE ROUTINES MUST OBEY HAL/S LINKAGE CONVENTIONS

ASSEMBLY LANGUAGE MACROS ARE PROVIDED TO HELP.

#CASMSUB HMAIN

\[ \cdots \]

(ACCESS ARGUMENTS IN REGISTERS OR VIA R13)

\[ \cdots \]

HCALL HALSUB

\[ \cdots \]

HEXIT

\[ \cdots \]

END
HAL/S-360 LINKAGE CONVENTIONS

R12 ——> SYSTEM INTRINSICS  (E.G.  PROCEDURE CALLER)
R13 ——> CURRENT STACK FRAME  PROCEDURE EXITER
R14 ——> RETURN ADDRESS
R15 ——> ENTRY POINT ADDRESS
F0, F2, F4 ——> 1ST THREE SCALAR ARGUMENTS
R0, R1, R2, R3, R4 ——> 1ST FIVE INTEGER OR BIT ARGUMENTS ... OR ... POINTERS TO ARGUMENTS OF OTHER DATA TYPES

EXAMPLE:

CALL SUB1(A, B, C) ASSIGN(D);

LH R0, A  A INTEGER
LA R1, B  B VECTOR
LE F0, C  C SCALAR
LA R2, D  D BIT(16)
BALR R14, R12  GO TO PROCEDURE CALLER
DC A(SUB1)  ENTRY POINT ADDRESS
HALLINK STACK CALCULATION

ACTION: CREATE SEPARATE STACKS OF LENGTH 300 FOR BOTH P1 AND P2.
THE ERROR ENVIRONMENT IS REPRESENTED ON THE STACK

INCLUDES "ACTIVE" BIT - TURNED ON BY ACTION | ON ERROR, OFF BY ACTION OFF ERROR

ERROR RECOVERY EXECUTIVE SEARCHES UP THE STACK FOR ERROR HANDLING SPECIFICATIONS

| 1:5 | ACTION |
| 2:8 | ACTION |
| 3:  | ACTION |

| 1:6 | ACTION |
| 2:8 | ACTION |
| 4:1 | ACTION |

| 1:5 | ACTION |
| 3:1 | ACTION |
| 3:7 | ACTION |
EXECUTION OF UPDATE BLOCKS AND EXCLUSIVE PROCEDURES IS
ALSO REPRESENTED ON THE STACK.

TO OPERATING SYSTEM REPRESENTATION
OF LOCK GROUPS

TO OPERATING SYSTEM REPRESENTATION
OF EXCLUSIVE PROCEDURE LOCKS

FOR TERMINATE AND ON ERROR
<statement>, MUST "PEEL BACK"
THE STACK AND FREE LOCK GROUPS
AND EXCLUSIVE PROCEDURES.
ERROR RECOVERY FEATURES

ON ERROR
OFF ERROR
SEND ERROR

- USED FOR SPECIAL HANDLING OF UNUSUAL CONDITIONS.

- SYSTEM-DEFINED ERRORS
  ARITHMETIC OVERFLOW, END OF FILE, ETC.

- USER-DEFINED ERRORS
  Signalled with SEND ERROR statement
  SYSTEM-DEFINED ERRORS CAN ALSO BE SIMULATED
ON ERROR\texttt{min} \texttt{<statement>};

- WHEN THE ERROR OCCURS, EXIT ANY CALLED BLOCKS, EXECUTE THE <statement>, AND CONTINUE WITH THE STATEMENT FOLLOWING THIS ONE IN THE PROGRAM.

ON ERROR\texttt{min} IGNORE;

- CONTINUE FROM THE POINT WHERE THE ERROR OCCURRED, USING A "STANDARD SYSTEM FIXUP" FOR THE ERROR.

ON ERROR\texttt{min} SYSTEM;

- A STANDARD ACTION (USUALLY PRINT A MESSAGE AND USE THE "STANDARD SYSTEM FIXUP" IS TAKEN.)
ON ERROR \( \text{min} \) \{ \text{IGNORE} \} \[ \text{SYSTEM} \]

AND \{ \text{SET} \}

\{ \text{SIGNAL} \}

\{ \text{RESET} \} \langle \text{event} \rangle \}

\( \text{SEND ERROR} \text{\_\_min} \)

(SIMULATE A SYSTEM- OR USER-DEFINED ERROR)
- Each error is assigned a group number and a number within the group.

- Each block can handle a certain set of errors (on error somewhere in the block).

- Handling of errors can be activated and deactivated dynamically (through execution of on/off error statements).
THE ERROR ENVIRONMENT IS ESTABLISHED THROUGH THE DYNAMIC NESTING OF CALLS

P1: PROCEDURE;
HANDLES ERRORS
1:5, 2:8, 3:

CALL P2;

P2: PROCEDURE;
HANDLES ERRORS
1:6, 2:8, 4:1

CALL P3;

P3: PROCEDURE;
HANDLES ERRORS
1:5, 3:1, 3:7

BEFORE AND AFTER CALL P2; ERROR 2:8 IS HANDLED HERE.

DURING P2 AND P3 ERROR 2:8 IS HANDLED HERE.

ERRORS 3:1 AND 3:7 HANDLED HERE; OTHERS IN GROUP 3 HANDLED IN P1.
NAME VARIABLES

Q. WHAT IS IN A NAME?

A. AN ADDRESS (POINTER).

360: 00 24 BIT ADDRESS

360: 00 F0000016 (NULL)

FC: 16 BIT ADDRESS

FC: 0000 (NULL)

NOTE: FC NAME VARIABLES ARE ALWAYS HALFWORDS AND 360 NAME VARIABLES ARE ALWAYS FULLWORDS (AND FULLWORD ALIGNED). THIS IS THE ONE FUNDAMENTAL INCONSISTENCY BETWEEN FC AND 360 STORAGE ALLOCATIONS AND IS NOT SURMOUNTABLE BY USE OF THE FCDATA OPTION.
NAME VARIABLES (CONTINUED)

PROPERTIES:

- Name variables are a class of data items whose values are pointers (addresses) to other data items.

- A name variable is established by declaring it as though it were a data item except that the keyword NAME is used.

- A name variable can only point at data that matches it in type, precision, arrayness, etc.

- A mechanism exists for initializing a name variable to point to a given data item. Name-variables may also be dynamically "re-directed" to other data items.

- Data items may be manipulated in the normal ways through references to name variables pointing to them.
DECLARE I INTEGER INITIAL(3),
NI NAME INTEGER INITIAL (NAME(I));
NI = NI + 1;
/* SAME AS I = I + 1; */

Automatic reference
NAME VARIABLES (CONTINUED)

USES FOR NAME VARIABLES

- Move around a pointer to a block of data (or code)
  rather than the whole block.

- Accommodate data structures of dynamically varying
  size in a fixed-size memory.

PROBLEMS WITH UNRESTRICTED POINTERS

DECLARE I INTEGER;
    S SCALAR,
    N NAME;
NAME(N) = NAME(I); /* Point N at I */
L:  N = N + 1;
    NAME(N) = NAME(S); /* What code is compiled? */
    IF N < S THEN GO TO L; /* Point N at S */
    /* Conversion required? */
SO, A NAME VARIABLE IS DECLARED TO POINT TO VARIABLES OF A GIVEN DATA TYPE ...

DECLARE NI NAME INTEGER,
    NS NAME SCALAR,
    NV NAME VECTOR(4),
    NA NAME ARRAY(2,2) BOOLEAN;

DECLARE INTEGER DOUBLE,
    I, J, K,
    N NAME;
    /* FACTORED ATTRIBUTES */
HENCE THE COMPILER CAN CHECK FOR TYPE COMPATIBILITY.

DECLARE I INTEGER,
    S SCALAR,
    N NAME INTEGER;

NAME(N) = NAME(I); /* POINT N AT I */

L:  N = N + 1;       /* CODE FOR INTEGER ADDITION */
    NAME(N) = NAME(S); /* ERROR-TYPE MISMATCH */
    IF N < S THEN GO TO L; /* N DEFERENCED AND CONVERTED TO SCALAR */
NAME VARIABLES (CONTINUED)

NAME VARIABLES CAN POINT TO DATA (OR CODE) WITH
THE FOLLOWING ATTRIBUTES:

- SINGLE OR DOUBLE
- INTEGER
- SCALAR
- VECTOR
- MATRIX
- OR *
- CHARACTER
- BIT
- BOOLEAN
- ARRAY
- STRUCTURE
- VECTOR
- PROGRAM
- EVENT
- TASK

DECLARE MATRIX(2,3), A, B, NB NAME INITIAL(NAME(B));
DECLARE MATRIX(3,3), C, D, ND NAME INITIAL(NAME(D));
A = NB; /* SAME AS A = B; */
A = ND; /* ERROR-DIMENSION MISMATCH */
NAME(NB) = NAME(ND); /* ERROR-DIMENSION MISMATCH */
NB = ND$(2 AT 1,*); /* SAME AS B = D$(2 AT 1,*); */
NAME VARIABLES (CONTINUED)

NAME DATA ITEMS POINTING TO DATA

Declarations of NAME data items for pointing to data have exactly the same form as declarations of ordinary data items, except that the keyword NAME immediately follows the identifier name declared.

Examples:

DECLARE A NAME ARRAY(100) SCALAR;
DECLARE MATRIX(3,3) DOUBLE, M1 NAME, M2 NAME;
DECLARE B NAME BIT(16),
C NAME CHARACTER(80);
STRUCTURE Q:
  1 QX INTEGER,
  1 QS SCALAR,
  1 Q1,
  2 QN BIT(16),
  2 QC CHARACTER(80);
DECLARE QO NAME Q-STRUCTURE;

Given the above declarations:

A may only point to 1-dimensional single precision scalar arrays of size 100.

M1, M2 may only point to 3x3 double precision matrices.

B may only point to 16-bit strings.

C may only point to character strings of maximum length 80.

QO may only point to Q-STRUCTURES with a single copy.
NAME VARIABLES (CONTINUED)

NAME DATA ITEMS POINTING TO CODE BLOCKS

Declarations of NAME data items for pointing to programs and tasks have the following basic form:

- DECLARE name NAME PROGRAM;
- DECLARE name NAME TASK;

1. name is any legal HAL/S identifier name.

Such declarations can be part of a compound or factored declaration statement.

Examples:

- DECLARE P1 NAME PROGRAM;
- DECLARE T1 NAME TASK;
- DECLARE P2 NAME PROGRAM,
  T2 NAME TASK,
  S1 NAME SCALAR;

Given the above declarations:

P1, P2 may only point to program blocks.
T1, T2 may only point to task blocks.
NAME VARIABLES (CONTINUED)

POINTERS TO CODE BLOCKS

P1: EXTERNAL PROGRAM
CLOSE P1;
P2: EXTERNAL PROGRAM
CLOSE P2;
P3: EXTERNAL PROGRAM
CLOSE P3;

MASTER: PROGRAM
DECLARE NP NAME PROGRAM,
   I INTEGER;

DO WHILE TRUE;
   DO FOR I = 1 TO 3;
      DO CASE I;
         NAME(NP) = NAME(P1);  // PROMOTE P1, P2, P3 IN TURN */
         NAME(NP) = NAME(P2);
         NAME(NP) = NAME(P3);
      END;
      UPDATE PRIORITY NP TO 200;  // TO HIGH PRIORITY FOR 1 SEC */
      WAIT 1;
      UPDATE PRIORITY NP TO 50;  // THEN REVERT TO NORMAL PRI. */
   END;

END;

CLOSE MASTER;

CODE BLOCK
NAME VARIABLES (CONTINUED)

NAME DATA ITEMS AS STRUCTURE TERMINALS

Examples:

```
STRUCTURE O:
  1 QS NAME SCALAR,
     1 Q1,
     2 QC NAME CHARACTER(80),
     2 QR NAME PROGRAM,
     2 QR NAME BOOLEAN,
     1 Q2,
     2 QA ARRAY(4) BIT(16);
```

Note that NAME data items for pointing to events can appear in a structure template, even though events themselves cannot. Note also that NAME data items in a template A may point to structures, even those possessing A as template.
Examples:

The following are legal definitions:

```
| STRUCTURE R: |
|  1 OR NAME R-STRUCTURE; |
|  1 QE NAME EVENT; |
| DECLARE ZR R-STRUCTURE; |
| DECLARE NZR NAME R-STRUCTURE; |
```

In this example NZR may point to ZR. ZR.OR may also point to ZR. The implications of this ability will be investigated later.
## NAME VARIABLES (CONTINUED)

### PROPERTIES OF DECLARED NAME DATA ITEMS

<table>
<thead>
<tr>
<th>ATTRIBUTE OF NAME DATA ITEM</th>
<th>Applies to Data or Code Block Pointed To</th>
<th>Applies to NAME Data Item Itself</th>
<th>Comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>ARRAY()</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BIT()</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>BOOLEAN</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CHARACTER()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>EVENT</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>MATRIX()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>INTEGER</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SCALAR</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STRUCTURE()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>TASK</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>SINGLE</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>DOUBLE</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>DECIMAL</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>ALIGNED</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>RIGID</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>INITIAL()</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>CONSTANT()</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>STATIC</td>
<td></td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>AUTOMATIC</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Notes:

1. The forms ARRAY(*) or STRUCTURE(*) are illegal.
2. The form CHARACTER(*) when used for a NAME data item enables it to point to a character data item of any maximum length.

See note (1)
NAME VARIABLES (CONTINUED)

NAME DATA ITEMS AND TEMPORARIES

The nature and purpose of temporary data items were described in Section 26.3. The following rules summarize relationships between temporary data items and NAME data items.

1. No NAME data item may point to a temporary data item.

2. NAME data items may not themselves be declared as temporary data items.

EXAMPLE:

DO;
   TEMPORARY INTEGER, I, J;
   \_ILLEGAL\_ TEMPORARY NV NAME VECTOR;
   . . .
   END;
NAME VARIABLES (CONTINUED)

INDIRECT ACCESS THROUGH NAME DATA ITEMS

Examples:

```
DECLARE VECTOR(3), V, NV NAME;
DECLARE SCALAR, S, NS NAME;
DECLARE NT NAME TASK;

T: TASK;

{ task body }

CLOSE T;
```

If NV + V, NS + S and NT + T*, then

```
NS = NV.NV;
SCHEDULE NT IN NS PRIORITY(50);
```

effectively performs the operations:

```
S = V.V;
SCHEDULE T IN S PRIORITY(50);
```

* In this and following examples "*" means "points to".
NAME VARIABLES (CONTINUED)

The foregoing statements about appearances of NAME data items, while appearing simple and unequivocal, contain a number of subtle implications arising from:

- interactions in structure data items;
- the effects of subscripting.
NAME VARIABLES (CONTINUED)

INDIRECT ACCESSING AND STRUCTURES

The subtleties of indirect accessing in conjunction with structures arise as a consequence of these two facts:

- Any structure may possess NAME structure terminals some of which may point to structure data items.
- Such a NAME structure terminal can actually point back to the structure containing it.

These subtleties are best illustrated by the extended examination of an apparently very simple example. By the rules given in Section 20.2, the following are legal structure declarations:

```
STRUCTURE A:
  1 C SCALAR,
  1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE, z1, z2, z3;
DECLARE Z4 NAME A-STRUCTURE;
```

\( Z_1.B \) is a NAME structure terminal of \( A-STRUCTURE \) type, which may therefore legally point to \( Z_2 \). Pictorially:

![Diagram](image-url)
Because $z_{1.0}$ points to $z_2$, any appearance of $z_2$ may be substituted by $z_{1.0}$, so achieving indirect access to $z_2$.

It is crucially important at this point to understand that because $z_{1.0}$ points to $z_2$, parts of $z_2$ as well as $z_2$ itself may be indirectly accessed. For example, to achieve indirect access to $z_{2.0}$, the appearance of $z_2$ in the qualified name is substituted by $z_{1.0}$. That is, indirect access to $z_{2.0}$ is achieved by the qualified form $z_{1.0}.z_{2.0}$. 
To illustrate this substitution process further, if Z4 points to Z2, then Z2.C is indirectly accessed by the qualified form Z4.C, and if Z4 points to Z1, then Z2.C is indirectly accessed by the qualified form Z4.B.C.

Multiple levels of indirection are handled in the same way. Suppose for example that in addition Z2.B points to Z3. Then pictorially:
Using the same kind of substitution as before, Z3 may be indirectly accessed by the qualified form Z1.B.B, so that in its turn, structure terminal C in Z3 may be indirectly accessed by the qualified reference Z1.B.B.C.

Restating how the form Z1.B.B.C was arrived at, the following steps were taken:

- Substitution of Z2.B.C for Z3.C (since Z2.B points to Z3);
There are other curious consequences arising from the interaction of indirect accessing with structures. Suppose now, for example, that Z2.B points to Z1 rather than Z3. Then, pictorially:

Now Z2.C can be indirectly accessed by the qualified form Z1.B.C, since Z1.B points to Z2. Since Z2.B points to Z1, the following forms are also possible:

Z2.B.B.C
Z1.B.B.B.C
Z2.B.B.B.B.C
Z1.B.B.B.B.B.B.C

This example illustrates the logical consequence of a closed indirection loop between two structures.
NAME VARIABLES (CONTINUED)

1 MT ALPHA;
1 MT PROGRAM;
2 MT STRUCTURE A:
  1 C SCALAR,
  1 B NAME A-STRUCTURE;
3 MT DECLARE A-STRUCTURE,
4 MT DECLARE Z4 NAME A-STRUCTURE;
5 MT NAME(Z1.B) = NAME(Z2);
6 MT Z2.C = 5;
7 MT Z1.B.C = 6;
8 MT WRITE(6) Z2.C, Z1.B.C;
9 MT NAME(Z2.B) = NAME(Z3);
10 MT Z3.C = 7;
11 MT Z2.B.C = 8;
12 MT Z1.B.B.C = 9;
14 MT CLOSE;

CALL NAME2
TENPHONE ASSUMED AS A MEMBER NAME
HAL/5-350 V15.0 START TIME: 22:34:02.23 DAY: 76/075
6 0.000000E+00  6.000000E+00  6.000000E+00
9 0.000000E+00  9.000000E+00  9.000000E+00

10-22
**NAME VARIABLES (CONTINUED)**

<table>
<thead>
<tr>
<th>adress</th>
<th>variable</th>
<th>value</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>0000009</td>
<td>EC21</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000000A</td>
<td>EC19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000000C</td>
<td>EC19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000002E</td>
<td>SSE5</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000003E</td>
<td>SSE5</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000003E</td>
<td>SSE5</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000004D</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000005E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000006E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000007E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000008E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000009E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000010E</td>
<td>9A19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000011E</td>
<td>SSE8</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000012E</td>
<td>SSE8</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000013E</td>
<td>SSE8</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000014E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000015E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000016E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000017E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000018E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000019E</td>
<td>9A29</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00001AE</td>
<td>EC31</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00001BE</td>
<td>EC31</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000020E</td>
<td>EC31</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000021E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000022E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000023E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000024E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000025E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000026E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000027E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000028E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000029E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00002AE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00002BE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00002CE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00002DE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00002EE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000030E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000031E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000032E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000033E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000034E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000035E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000036E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000037E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000038E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000039E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00003AE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00003BE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00003CE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00003DE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00003EE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000040E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000041E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000042E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000043E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000044E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000045E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000046E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000047E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000048E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>000049E</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00004AE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00004BE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00004CE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00004DE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
<tr>
<td>00004EE</td>
<td>SC 19</td>
<td>0000</td>
<td></td>
</tr>
</tbody>
</table>
NAME VARIABLES (CONTINUED)

1 MT BETA:
1 MT PROGRAM:
2 MT STRUCTURE A:
2 MT 1 C SCALAR,
2 MT 1 B NAME A-STRUCTURE;
3 MT DECLARE A-STRUCTURE.
3 MT 24: 22;
4 MT NAME(Z1.B) = NAME(Z2.B);
5 MT NAME(Z2.B) = NAME(Z1.B);
6 MT Z1.B.C = 1;
7 MT Z2.B.B.C = 2;
8 MT Z1 B B B C = 3;
9 MT Z2.B.B.B.C = 4;
10 MT Z2.B.B.B.B.C = 5;
11 MT Z1.B.B.B.B.B.C = 6;
12 MT Z2.B.B.B.B.B.B.B.B.B.C = 20;
13 MT CLOSE;
INDIRECT ACCESS AND SUBSCRIPTING

In this discussion, for simplicity, subscripting in connection with structures or structure terminals will at first be excluded. With this restriction, subscripting on NAME data items is straightforward in its meaning.

Subscripting is effective on the data item that is being indirectly accessed.

With this interpretation, it is clear that such subscripts must be legal for the data type pointed to. In particular, NAME data items pointing to programs and tasks may not be subscripted.
NAME VARIABLES (CONTINUED)

Examples:

DECLARE VECTOR(3), V, NV NAME;
DECLARE ARRAY(2) CHARACTER(4), C, NC NAME;
DECLARE BIT(4), B, NB NAME;

Let \( V = \begin{bmatrix} 0.5 \\ 1.5 \\ 2.5 \end{bmatrix} \), \( C = (\text{ABCD} \ ' \text{EFGH}') \), \( B = 1010_2 \)

Then if \( NV \rightarrow V \), \( NC \rightarrow C \), \( NB \rightarrow B \):

\( NV_3 \equiv 2.5 \) since \( V_3 \) is indirectly referenced,
\( NC_{1:3} \equiv \text{'C'} \) since \( C_{1:3} \) is indirectly referenced,
\( NB_2 \rightarrow 3 \equiv 01_2 \) since \( B_2 \rightarrow 3 \) is indirectly referenced.

\( NV_5 \), \( NB_9 \) are illegal since the subscripting is illegal for \( V \) and \( B \) respectively. Such subscripting is always illegal since \( NV \) can only point to 3-vectors, and \( B \) to 4-bit strings.
The complexities arising from structure subscripting are best studied by another apparently simple example. Suppose that the following declarations are made:

```
STRUCTURE A:
  1 C MATRIX(3,3),
  1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

Let copies 1, 2 and 3 of Z1.B point respectively to copies 2, 3 and 1 respectively of Z2. Pictorially:
According to the substitution process previously described, the three copies of structure terminal C and Z2 can be indirectly accessed by specifying the three copies of Z1.B.C:

- Z1.B.C₁; indirectly accesses Z2.C₂;
- Z1.B.C₂; indirectly accesses Z2.C₃;
- Z1.B.C₃; indirectly accesses Z2.C₁;
NAME VARIABLES (CONTINUED).

Using the terminology of Section 20.1, Z2.C is an operand with arrayness (1:3). Indirectly accessed as Z1.B.C, the operand still has arrayness (1:3) but the order of the individual elements is different. In general of course the three copies of Z1.B may point to three different structures (all with template A), resulting in operand Z1.B.C being synthesized from three different sources.

Note that the structure subscript is effective before indirection not after. As a further illustration, in

Z1.B.C_{1;3,3}

the structure subscript selects copy 1 of the pointers Z1.B. Note, however, that in contrast the component subscript selects the component in row 3 and column 3 of C in the structure to which Z1.B points.

This is not always true for structure subscripts. For example, let Z3 point to Z2. Then in

Z3.B.C_{1;3,3}

the structure subscript selects copy 1 of Z2, which is pointed to by Z3.
These examples illustrate the following general rule:

A structure subscript may either be effective on the data being indirectly accessed, or upon the NAME data item accessing it, depending on whether the data pointed to has copies, or whether the NAME data item itself has copies.

*Note that since a structure terminal which is itself a structure (or a NAME data item pointing to a structure) cannot possess copies, the two forms of structure subscripting are mutually exclusive.
NAME VARIABLES (CONTINUED)

THE NAME PSEUDO-FUNCTION

- ORDINARY REFERENCE TO NAME VARIABLE ACCESSES THE VARIABLE WHICH IT POINTS TO. (A DEREFERENCED USAGE.)
- NAME PSEUDO-FUNCTION IS USED TO ACCESS OR CHANGE THE (POINTER) VALUE OF THE NAME VARIABLE ITSELF.

DECLARE SCALAR, S1, S2, NS NAME, NS1 NAME INITIAL(NAME(S1));

\[
\begin{align*}
S2 &= NS1; \\
\text{NAME}(NS) &= \text{NAME}(NS1); \\
\text{NAME}(NS) &= \text{NAME}(S2); \\
\text{NAME}(NS) &= NS1; \\
NS &= \text{NAME}(NS1);
\end{align*}
\]

ERROR-TYPE MISMATCH!
NAME VARIABLES (CONTINUED)

IN GENERAL:

- NAME VARIABLE by itself denotes the ordinary variable which it points to.

- NAME( NAME VARIABLE ) obtains the pointer contents of the name variable.

- NAME( ORDINARY VARIABLE ) creates a pointer to that ordinary variable.
NAME VARIABLES (CONTINUED)

Examples:

Given:

```
DECLARE S SCALAR,
   NS NAME SCALAR,
   NT NAME TASK,
   NA NAME ARRAY(1000) INTEGER;
```

```
STRUCTURE Q:
   1 QS SCALAR,
   1 QN NAME Q-STRUCTURE;
```

```
DECLARE 2Q Q-STRUCTURE;
```

the following are legal:

```
NAME(S)   
NAME(2Q.QS)   }
   reference only
NAME(NS)
NAME(NT)
NAME(NA)
NAME(ZQ.QN)
```

the following are illegal:

```
NAME(1.5)
NAME(S/2)
```
SUBSCRIPTING AND NAME VARIABLES

- SUBSCRIPTING OF DEREFERENCED NAME VARIABLES IS ALLOWED
  E.G., DECLARE ARRAY(3) VECTOR, NV NAME INITIAL(NAME(V));
  \[ V_{1:3} = NV_{2:3} \]
  /* SAME AS \[ V_{2:3} = V_{3:3} */

- SUBSCRIPTING INSIDE A REFERENCE TO THE NAME PSEUDO-FUNCTION
  CAN ONLY APPEAR IN REFERENCE CONTEXT (NOT IN ASSIGNMENT CONTEXT).
  SUBSCRIPTING APPLIES TO THE VARIABLE BEING POINTED TO.

E.G., DECLARE INTEGER
  A ARRAY(10),
  NA NAME ARRAY(10) INITIAL(NAME(A)),
  NI NAME, /* NI IS NAME OF SCALAR*/

\[ \text{NAME}(NI) = \text{NAME}(A_{3:3}); \]
\[ \text{NAME}(NI) = \text{NAME}(NA_{3:3}); \]
NAME VARIABLES (CONTINUED)

COMPONENT SUBSCRIPTING INSIDE NAME( )

- ILLEGAL FOR BIT AND CHARACTER STRINGS
  e.g., DECLARE B BIT(16);  NAME(B5)

- MUST SELECT A SINGLE SCALAR FROM VECTORS AND
  MATRICES (ALSO SINGLE ELEMENT FROM ARRAYS)
  e.g., DECLARE M MATRIX;

\[\begin{array}{cccccccc}
1 & 2 & 3 & 4 & 5 & \ldots & 16 \\
\end{array}\]
NAME VARIABLES (CONTINUED)

STRUCTURE SUBSCRIPTING INSIDE NAME( )

- In assignment context, OK only if NAME( ) is applied to a name variable in a structure with multiple copies - then it selects the appropriate copy of the name variable itself.

E.g. 

```
STRUCTURE S: 1 N NAME SCALAR;
DECLARE S S-STRUCTURE(10);
NAME(N_1) = NAME(N_2);
```

- In reference context, only one can apply:
  - Name variable defined in a structure with copies, subscripting is effective on the name variable itself.
  - Name variable pointing to a structure with copies, subscripting is effective on the variable being pointed to.

```
STRUCTURE S: 1 N NAME-STRUCTURE(10);  // ILLEGAL!
```
INTERACTION WITH STRUCTURES

STRUCTURE A:
1 C SCALAR,
1 B NAME A-STRUCTURE;
DECLARE A-STRUCTURE, Z1, Z2, Z3;
DECLARE Z4 NAME A-STRUCTURE;

Let Z1.B point to Z2, and Z2.B point to Z3, as shown pictorially below:
A pointer value to \( Z3, C \) can be created by the construct:

\[
\text{NAME}(Z3,C)
\]

And also by...

\[
\text{NAME}(Z2,B,C)
\]

Now \( Z1,B \) points to \( Z2 \) so that \( Z3,C \) is accessed through two levels of indirection by \( Z1,B,B,C \). A third way of creating a pointer value to \( Z3,C \) is therefore:

\[
\text{NAME}(Z1,B,B,C)
\]

If furthermore, \( Z4 \) points to \( Z1 \), then

\[
\text{NAME}(Z4,B,B,C)
\]

also has the same effect.
In each of the above cases, the argument of the NAME pseudo-function is Z1.C which is an ordinary data item, even though indirect access is used. Each of the above instances may therefore only be used in a reference context.

The pointer value of Z2.B can itself be set up by using

\[ \text{NAME}(Z2.B) \]
in an appropriate assignment context to be described. The NAME structure terminal z2.B may be indirectly accessed by the qualified form z1.B.B, since z1.B points to z2. Hence, the pointer value of z2.B can also be set up by using:

NAME(z1.B.B)

in assignment context. With z4 again pointing to z1,

NAME(z4.B.B)

has the same effect, since z2.B is again accessed, this time through two levels of indirection.
NAME VARIABLES (CONTINUED)

ARGUMENTS WITH SUBSCRIPTS

Examples:

Given the following declarations:

| DECLARE V VECTOR(3),
| NV NAME VECTOR(3),
| S ARRAY(100) SCALAR,
| NS NAME ARRAY(100) SCALAR,
| M ARRAY(5) MATRIX(3,3),
| NM NAME ARRAY(5) MATRIX(3,3),
| C CHARACTER(80),
| NC NAME CHARACTER(80); |

suppose that NV + V, NS + S, NM + M and NC + C.

The following are legal in contexts causing reference of pointer values:

\[
\begin{align*}
\text{NAME}(V_3) & \quad \text{creates pointer to scalar value which is 3rd element of vector } V \\
\text{NAME}(NV_3) & \quad \text{same as above since } NV + V \\
\text{NAME}(S_5) & \quad \text{creates pointer to 5th array element of array } S
\end{align*}
\]
NAME VARIABLES (CONTINUED)

NAME(N5)   same as above since NS = 5
NAME(M3:1,1) creates pointer to scalar value in row 1,
column 1 of 3rd array element of M
NAME(NM3:1,1) same as above since NM > M
NAME(M4)   creates pointer to 4th array element in M

The following are illegal:

NAME(C1)   subscripting on character strings
    illegal
NAME(NC1)
NAME(V1 TO 2) more than one element of V selected
NAME(M*:1,1) one scalar value selected from more than one
    array element
FURTHER RULES:

1. When a NAME pseudo-function is used to assign pointer values, only structure subscripting effective on the pointer copies is legal.

2. For NAME pseudo-functions in reference context, array and component subscripting is always effective on the ordinary data item specified or indirectly accessed. Structure subscripting is effective in the ordinary data item specified or indirectly accessed, or upon the NAME data item indirectly accessing it, depending on which possesses the multiple copies.
NAME VARIABLES (CONTINUED)

Example:

Given the following declarations

```plaintext
STRUCTURE A:
  1 M ARRAY(5) MATRIX(3,3),
  1 C CHARACTER(80),
  1 V VECTOR(6),
  1 B NAME A-STRUCTURE;
DECLARE Z A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

```plaintext
let Z1.B1 + Z2
Z1.B2 + Z2
Z1.B3 + Z2
Z3 + Z1
```

Illustrations for NAME pseudo-functions in a reference context -

(a) Array and component subscripting:

- `NAME(Z.M[1:3,3])` creates a pointer to the scalar value in row 3, column 3 of the first array element of Z.M
- `NAME(Z.M[1:1,1])` is illegal since the subscript selects a scalar value from more than one array element of Z.M
- `NAME(Z.C[10 TO 15])` is illegal since character strings may not possess component subscripts
- `NAME(Z.V[1])` creates a pointer to the 1st element of vector Z.V
- `NAME(Z.V[1 TO 3])` is illegal since more than one element of Z.V is selected by the subscript
Example:

Given the following declarations

**STRUCTURE A:**

1 M ARRAY(5) MATRIX(3,3),
1 C CHARACTER(00),
1 V VECTOR(6),
1 B NAME A-STRUCTURE;
DECLARE Z A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;

let Z1.B1 = Z2.Z


Z1.B3 = Z2.Z

Z3 = Z1

**Illustrations for NAME pseudo-functions in a reference context:**

(b) Structure subscripting effective upon the data item pointed to or directly specified:

NAME(Z1) creates a pointer to the second copy of Z1 since the subscript acts directly on Z1.

NAME(Z3) since Z3 is a single pointer, pointing to the whole of Z1, the subscript is effective on Z1 rather than Z3; hence a pointer to the second copy of Z1 is again created.
<table>
<thead>
<tr>
<th>NAME</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAME(Z1.M)</td>
<td>creates a pointer to the array of matrices ( M ) in the second copy of ( Z_1 )</td>
</tr>
<tr>
<td>NAME(Z3.M)</td>
<td>as before, the structure subscript is effective on ( Z_1 ) rather than ( Z_3 ); hence as before a pointer to the array of matrices ( M ) in the second copy of ( Z_1 ) is created</td>
</tr>
<tr>
<td>NAME(Z1.M)</td>
<td>is illegal since the subscript selects more than one copy of structure ( Z_1 )</td>
</tr>
<tr>
<td>NAME(Z3.M)</td>
<td>is illegal for the same reason</td>
</tr>
<tr>
<td>NAME(Z1.M)</td>
<td>is illegal since subscripting to select one copy of ( Z_1.M ) must be used</td>
</tr>
</tbody>
</table>
NAME VARIABLES (CONTINUED)

Example:

Given the following declarations

```
STRUCTURE A:
  M ARRAY(5) MATRIX(3,3),
  C CHARACTER(80),
  V VECTOR(6),
  B NAME A-STRUCTURE;
DECLARE Z A-STRUCTURE;
DECLARE A-STRUCTURE(3), Z1, Z2, Z3 NAME;
```

let

```
Z1.B1 = Z2.2
Z1.B2 = Z2.3
Z1.B3 = Z2.1
Z3 = Z1
```

Illustrations for NAME pseudo-functions in a reference context -
NAME VARIABLES (CONTINUED)

(c) Structure subscripting effective on a pointer value:

The following examples use the fact that \( z_1.B_1 \) points to \( z_2 \):

\[
\text{NAME}(z_1.B_1) \quad \text{references the pointer value } z_1.B_1,
\]
i.e., it creates the pointer to \( z_2 \).

\[
\text{NAME}(z_1.B.M_{1,1}) \quad \text{the subscript is effective of } z_1.B,
\]
so that a pointer to the array of matrices in the second copy of \( z_2 \) is created.

\[
\text{NAME}(z_1.B.V_{1,1}) \quad \text{the structure subscript is effective on } z_1.B
\]
as before so that a pointer to the first component of the vector in the second copy of \( z_2 \) is created.

Note that there is no restriction on the selection of one pointer only by a structure subscript effective on pointer data:

\[
\text{NAME}(z_1.B) \quad \text{"simultaneously" references three pointer values}
\]

\[
\text{NAME}(z_1.B.M_{1,1} \text{ TO } 2) \quad \text{"simultaneously" creates two pointers, to matrices } z_2.M_{2,1} \text{ and } z_2.M_{3,1} \text{ respectively}
\]
NAME VARIABLES (CONTINUED)

INITIALIZATION OF NAME VARIABLES

\{
\begin{align*}
\text{INITIAL} & \quad (\text{name reference}) \\
\text{CONSTANT} &
\end{align*}
\}

NAME REFERENCE HAS THE FORM:

\begin{align*}
\text{name(ordinary variable)} \\
\text{name(NULL) or NULL}
\end{align*}

"NULL POINTER" - POINTS TO NOTHING AT ALL. UNINITIALIZED NAME VARIABLES ALSO HAVE THIS VALUE.

ORDINARY VARIABLE MUST BE:

- previously declared
- with datatype matching that of name variable

AND MUST NOT BE:

- subscripted
- dereferenced through another name variable

UNLESS SUBSCRIPTS ARE KNOWN AT COMPILE-TIME!
NAME VARIABLES (CONTINUED)

INITIALIZATION OF NAME VARIABLES

1. DECLARE S SCALAR,
   NS NAME SCALAR INITIAL(NAME(S));

2. NAME VARIABLES IN STRUCTURES WILL BE DISCUSSED LATER.
   STRUCTURE A:
   1 B SCALAR,
   1 C NAME A-STRUCTURE;
   DECLARE A-STRUCTURE,
   Z1,
   Z2 INITIAL(1,5, NAME(Z1));
   DECLARE NB NAME SCALAR INITIAL(NAME(Z1.B));

3. DECLARE A ARRAY(5) INTEGER INITIAL(0),
   NI NAME INTEGER INITIAL(NAME(A3));
NAME VARIABLES (CONTINUED)

Examples:

The following are legal initializations of NAME data items:

```
DECLARE S SCALAR, V ARRAY(4) VECTOR DOUBLE;
DECLARE NS1 NAME SCALAR INITIAL(NAME(S));
DECLARE NV1 NAME ARRAY(4) VECTOR DOUBLE INITIAL(NAME(V));

STRUCTURE A:
1 C SCALAR,
1 B NAME A-STRUCTURE;
DECLARE Z1 A-STRUCTURE;
DECLARE Z2 A-STRUCTURE INITIAL(1.5, NAME(Z1));
DECLARE NA NAME SCALAR INITIAL(NAME(Z1.C));

DECLARE V VECTOR(4);
DECLARE TV NAME SCALAR INITIAL(NAME(V));
```
The following are illegal initializations of NAME data items:

- `DECLARE T SCALAR;`
- `DECLARE NT NAME SCALAR DOUBLE INITIAL(NAME(T1));`  
  - `NT cannot legally point to T`
- `DECLARE NT1 NAME SCALAR INITIAL(NAME(T1));`
- `DECLARE T1 SCALAR;`  
  - `T1 is not previously defined`

**STRUCTURE X:**
- `1 Y SCALAR,`
- `1 Z NAME X-STRUCTURE;`
- `DECLARE XX1 X-STRUCTURE;`
- `DECLARE XX2 X-STRUCTURE INITIAL(1.5, NAME(XX1));`
- `DECLARE NX NAME SCALAR INITIAL(NAME(XX2.Z.Y));`  
  - contains implicit indirection since `XX2.Z` through previous initialization
NAME VARIABLES (CONTINUED)

NULL INITIALIZATION

All NAME data items which are not explicitly initialized, are implicitly initialized with null pointer values. The following examples show the explicit initialization to null pointer values.

Examples:

```
DECLARE LV NAME VECTOR INITIAL(NULL);
STRUCTURE A:
  1 C SCALAR,
  1 B NAME A-STRUCTURE
DECLARE Z A-STRUCTURE(20) INITIAL(20#(7.53, NULL));
```

each copy of B initialized to a null pointer value
NAME ASSIGNMENTS

\[ L_1, L_2, \ldots, L^n = R; \]

Each \( L \) and \( R \) is a NAME PSEUDO-FUNCTION.

E.g., \( \text{name}(\text{NS}_1), \text{name}(\text{NS}_2) = \text{name}(\text{NS}) \);

NAME COMPARISONS

\( L = R \)

\( L \neq R \)

\( L \) and \( R \) are both NAME PSEUDO-FUNCTIONS.

E.g., if \( \text{name}(\text{NS}_1) \neq \text{name}(\text{NS}_2) \) then ...

Equality if both NAME VARIABLES point to the same ORDINARY VARIABLE.
NAME ASSIGNMENTS (CONTINUED)

Examples:

Given the declarations

```plaintext
DECLARE S SCALAR,
    NS NAME SCALAR,
    NSD NAME SCALAR DOUBLE;
DECLARE V VECTOR(3),
    NV NAME VECTOR(3);
STRUCTURE A:
    1 C SCALAR,
    1 B NAME A-STRUCTURE;
DECLARE Z1 A-STRUCTURE,
    Z2 A-STRUCTURE,
    NZ NAME A-STRUCTURE;
```

then

```plaintext
NAME(NSD) = NULL; results in NSD + 0*
NAME(NS) = NAME(S); results in NS + S
NAME(NSD) = NAME(NS); is illegal since NS + S and
    NSD may not legally point to S itself
NAME(NV) = NAME(V); results in NV + V
NAME(NS) = NAME(V); results in NS + V2 - note that
    V2 is a scalar value, which
    is why NS may legally point to it
NAME(NZ) = NAME(Z1); results in NZ + Z1
NAME(NZ.B) = NAME(Z2); results in ZL.B + Z2 because of
    implied indirection in qualified
    reference NZ.B, in which NZ + Z1
NAME(NS) = NAME(NZ.B.C); results in NS + Z2,C because of
    2 levels of implied indirection
    in qualified form NZ.B,C, in which
    NZ + Z1 and ZL.B + Z2
```
NAME VARIABLES (CONTINUED)

MULTIPLE ASSIGNMENTS

Example:

Given

| DECLARE S SCALAR, |
| NS NAME SCALAR, |
| NT NAME SCALAR; |
| STRUCTURE U: |
| 1 US NAME SCALAR, |
| 1 UN NAME U-STRUCTURE; |
| DECLARE Z U-STRUCTURE; |

The following is a legal multiple NAME assignment:

| NAME(NS), NAME(NT), NAME(Z.US) = NAME(S); |
NAME VARIABLES (CONTINUED)

POINTER ARRAYNESS IN NAME ASSIGNMENTS

Examples:

Given

\begin{verbatim}
STRUCTURE A:
  1 B NAME SCALAR,
  1 C SCALAR;
DECLARE Z1 A-STRUCTURE(3),
      Z2 A-STRUCTURE(5);
DECLARE S SCALAR;
\end{verbatim}

then 3 copies of Z1.B exist, and 5 copies of Z2.B exist. Hence in

\begin{verbatim}
NAME(Z1.B) = NAME(S);
\end{verbatim}

the pointer arrayness on the left is (3) whilst the right hand operand has none. The result of
this assignment is:

\begin{verbatim}
Z1.B_1; \rightarrow S
Z1.B_2; \rightarrow S
Z1.B_3; \rightarrow S
\end{verbatim}
NAME COMPARISONS

Examples:

Given

DECLARE S SCALAR;
DECLARE NS NAME SCALAR INITIAL(NAME(S));
DECLARE NT NAME SCALAR INITIAL(NULL);

Then

NAME(NS) = NAME(S) is TRUE;
NAME:NS) = NAME(NULL) is FALSE;
NAME(NT) `= NAME(NULL) is FALSE;
NAME(NT) `= NAME(NS) is TRUE;
NAME VARIABLES (CONTINUED)

POINTER ARRAYNESS IN NAME COMPARISONS

Examples:

Given

| STRUCTURE A:
| 1 D NAME SCALAR,
| 1 C SCALAR;
| DECLARE Z1 A-STRUCTURE(3),
|    Z2 A-STRUCTURE(5);
| DECLARE S SCALAR;

After execution of

| NAME(Z1,D) = NAME(S);

then the result of the comparison

| NAME(Z1,D) = NAME(S) is TRUE

since
| Z1.D1; Z1.D2; Z1.D3;
Further,

\[ \text{NAME}(Z2.B) = \text{NAME}(Z1.B); \]

is illegal since the left and right hand pointer arraynesses are (5) and (3) respectively which do not match. However,

\[ \text{NAME}(Z2.B_{3}) = \text{NAME}(Z1.B); \]
\[ S_{3 \text{ TO } 5}; \]

is legal since the left hand arrayness has been reduced to (3). The result of the assignment is

\[ Z2.B_{3} = Z1.B_{1}; \quad \text{(i.e. they both have the same pointer value)} \]
\[ Z2.B_{4} = Z1.B_{2}; \]
\[ Z2.B_{5} = Z1.B_{3}; \]
NAME VARIABLES (CONTINUED)

Given

\[
\text{STRUCTURE A:}
\]

\[
\begin{align*}
&1 \text{ D NAME SCALAR,} \\
&1 \text{ C SCALAR;} \\
&\text{DECLARE } Z1 \text{ A-STRUCTURE(3),} \\
&Z2 \text{ A-STRUCTURE(5);} \\
&\text{DECLARE } S \text{ SCALAR;}
\end{align*}
\]

After subsequent execution of

\[
\begin{align*}
&\text{NAME}(Z1, \text{D}) = \text{NULL;} \\
&S
\end{align*}
\]

then the result of the comparison

\[
\text{NAME}(Z1, \text{D}) = \text{NAME}(S) \text{ is FALSE}
\]

because \(Z1, \text{D}_1 \rightarrow S\)

\[
\begin{align*}
Z1, \text{D}_2 \rightarrow S \\
Z1, \text{D}_3 \\
\end{align*}
\]

The comparison

\[
\text{NAME}(Z1, \text{D}) = \text{NAME}(Z2, \text{D})
\]

is illegal because the pointer arraynesses of the left and right operands are \(\{3\}\) and \(\{5\}\) respectively, which do not match. However, the comparison

\[
\text{NAME}(Z1, \text{D}) = \text{NAME}(Z2, \text{D}_3 \text{ TO } 5)
\]

is legal since the pointer arrayness of the right hand operand has been reduced to \(\{3\}\).
NAME ARGUMENTS AND PARAMETERS

P: PROCEDURE(NA) ASSIGN(NB);
    DECLARE NA NAME INTEGER,
        NB NAME ARRAY(10) SCALAR;
    CLOSE P;

- ARGUMENT MATCHING NA: NAME PSEUDO-FUNCTION IN REFERENCE CONTEXT (OR NULL).
- ARGUMENT MATCHING NB: NAME PSEUDO-FUNCTION IN ASSIGNMENT CONTEXT.

DECLARE I INTEGER;
DECLARE ARRAY(10) SCALAR, A, N NAME;

CALL P(NAME(I)) ASSIGN(NAME(N));  LEegal
CALL P(NULL) ASSIGN(NAME(N));    LEgal
CALL P(NAME(I)) ASSIGN(NAME(A));  ILLEGAL
CALL P(NULL) ASSIGN(NAME(N));    ILLEGAL

NAME(A) = NB
INPUT ARGUMENTS

The effect of using a pointer value as an input argument of a procedure or function is as if the pointer value were being assigned to the corresponding NAME input parameter. The attributes of the NAME input parameter must therefore allow legal acceptance of that pointer value.

Examples:

```
DECLARE S SCALAR;
DECLARE NS NAME SCALAR;
DECLARE NT NAME TASK;

::

F: FUNCTION(A,B) SCALAR;
  DECLARE A NAME SCALAR,
    B BOOLEAN;

  }
    function body

  CLOSE F;
```
NAME VARIABLES (CONTINUED)

NAME(NS) = NAME(S);
S = F(NAME(S), TRUE);
invocation results in input parameter Λ pointing to S
S = F(NAME(NS), FALSE);
has the same effect; Λ gets same pointer value as NS, i.e. Λ + S
S = F(NAME(NT), TRUE);
is illegal since pointer values legal for NT are not legal for Λ
S = F(NULL, FALSE);
results in Λ + Φ

Note that although ordinary input parameters are prevented from appearing in NAME pseudo-functions, NAME input parameters are only prevented from appearing in NAME pseudo-functions in assignment context.
NAME VARIABLES (CONTINUED)

ASSIGN ARGUMENTS

A pointer value may be passed both into and out of a procedure by the appearance of a NAME pseudo-function in the assign argument list of the procedure's invocation. The class of data items which can be pointed to by the NAME data item appearing in the NAME pseudo-function must be the same as that which can be pointed to by the corresponding NAME assign parameter.
NAME VARIABLES (CONTINUED)

Examples:

DECLARE NS NAME SCALAR;
DECLARE NT NAME TASK;
STRUCTURE A:
  1 B NAME A-STRUCTURE,
  1 C SCALAR;
DECLARE Z A-STRUCTURE;

::

P: PROCEDURE ASSIGN(U,V);
  DECLARE U NAME TASK,
  V NAME A-STRUCTURE:
  \{
    \}
  procedure body
CLOSE P;
::

CALL P ASSIGN(NAME(NT),NAME(Z.B));
  causes passage of pointer values
  between NT and U, and between Z.B
  and V.

CALL P ASSIGN(NAME(NS),NAME(Z));
  \hspace{1em} illegal because % is not a
  NS points to NAME data item
  scalar data items but NT points to
  tasks
NAME VARIABLES (CONTINUED)

POINTER ARRAYNESS IN ARGUMENTS

Examples:

**STRUCTURE A:**
1 B NAME SCALAR;
DECLARE Z A-STRUCTURE(20);
DECLARE S1 ARRAY(20) SCALAR,
S2 ARRAY(10) SCALAR;

P: PROCEDURE(U) ASSIGN(V);
DECLARE U NAME SCALAR,
V NAME SCALAR;

\[
\text{procedure body}
\]
CLOSE P;

F: FUNCTION(W) SCALAR;
DECLARE W NAME SCALAR;

\[
\text{function body}
\]
CLOSE F;
NAME VARIABLES (CONTINUED)

CALL P(NAME(Z,B_1)); ASSIGN(NAME(Z,B));

legal because pointer arrayness subscripted away

S = S_1 + F(NAME(Z,B));

illegal because pointer arrayness exists

legal because pointer arrayness \{20\} matches arrayness \{1:20\} of S_1

The above is equivalent to

\[ S_{1,i} = S_{1,i} + F(NAME(Z,B_i)); \] for \(1 \leq i \leq 20\)

wherein each of 20 invocations of F cause transmission of a different pointer value.
USES OF STRUCTURES AND NAME VARIABLES

1. A PRIORITY-ORDERED QUEUE

STRUCTURE QUEUE:
  1 IDNO INTEGER,
  1 PRTY INTEGER,
  1 LINK NAME QUEUE-STRUCTURE;
DECLARE QUEUE QUEUE-STRUCTURE(10),
  ANCHOR NAME QUEUE-STRUCTURE;
DECLARE INTEGER, I, NEW_ID, NEW_PRTY;

DO FOR 1 = 1 TO 10;
  IDNO, PRTY = I;
  NAME(LINK, I) = NAME(QUEUE, I + 1);
END; 

IDNO, PRTY = 10;
NAME(ANCHOR), NAME(LINK, 10) = NAME(QUEUE, 1);

Problem #1: Queue cannot be unqualified because of nested structure references.
USES OF STRUCTURES AND NAME VARIABLES

DECLARE NAME QUEUE-STRUCTURE, THIS, PREV;

/* FIND IDNO IN QUEUE */
NAME(PREV) = NAME(ANCHOR);
NAME(THIS) = NAME(ANCHOR.LINK);

DO WHILE THIS.IDNO \= NEW_IDNO;
   NAME(PREV) = NAME(THIS);
   NAME(THIS) = NAME(THIS.LINK);
END;

/* SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE */
THIS.PRTY = NEW_PRTY;
NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);

/* FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY */
DO UNTIL PREV.PRTY \= NEW_PRTY, AND NEW_PRTY \= THIS.PRTY;
   NAME(PREV) = NAME(THIS);
   NAME(THIS) = NAME(THIS.LINK);
END;
NAME(PREV.LINK) = NAME(ANCHOR);
NAME(ANCHOR.LINK) = NAME(THIS);
NAME VARIABLES (CONTINUED)

: Mt PRIQO;
1 Mt PROGRAM;
2 Mt STRUCTURE NODE;
2 Mt   IDNO INTEGER,
2 Mt   PRV INTEGER,
2 Mt   LINK NAME NODE-STRUCTURE,
3 Mt DECLARE QUEUE NODE-STRUCTURE(10),
3 Mt ANCHOR NAME NODE-STRUCTURE,
4 Mt DECLARE THIS NAME NODE-STRUCTURE,
4 Mt PREV NAME NODE-STRUCTURE,
5 Mt DECLARE INTEGER;
5 Mt I, NEWID, NEWPRV ;
6 Mt DO FOR I = 1 TO 9 ;
7 Mt QUEUE. IDNO , QUEUE. PRV = I ;
8 Mt + NAME(QUEUE. LINK ) = NAME(QUEUE ) ;
9 Mt END ;
10 Mt QUEUE. IDNO , QUEUE. PRV = 10 ;
11 Mt NAME(ANCHOR), NAME(QUEUE. LINK ) = NAME(QUEUE ) ;
Ct +
Ct 10 10
Ct
Ct
Ct NAME IDNO IN QUEUE
NAME VARIABLES (CONTINUED)

12 M1 NAME(PREV) = NAME(ANCHOR),
   +
13 M1 NAME(THIS) = NAME(ANCHOR.LINK),
14 M1 DO WHILE THIS.IDOH \= NEW.IDH;
   +
15 M1 NAME(PREV) = NAME(THIS),
16 M1 NAME(THIS) = NAME(THIS.LINK),
17 M1 END,

SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE
18 M1 THIS.PRTY = NEW.PRTY.
19 M1 NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK),
   +
20 M1 DO UNTIL PREV.PRTY (= NEW.PRTY AND NEW.PRTY (= THIS.PRTY),
   +
21 M1 NAME(PREV) = NAME(THIS),
22 M1 NAME(THIS) = NAME(THIS.LINK),
23 M1 END,
24 M1 NAME(PREV.LINK) = NAME(ANCHOR),
25 M1 NAME(ANCHOR.LINK) = NAME(THIS),
26 M1 CLOSE.
NAME VARIABLES (CONTINUED)

NAME(ANCHOR), NAME(PREV), NAME(PREV.LINK) = NAME(ANCHOR), NAME(PREV.LINK) = NAME(ANCHOR)

END;

SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE

THIS.PRV = NEw.PRV;

NAME(ANCHOR), NAME(PREV.LINK) = NAME(THIS.LINK);

FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY

DO UNTIL PREV.PRV (= NEw.PRV AND NEw.PRV (= THIS.PRV);

NAME(PREV) = NAME(THIS);

NAME(THIS) = NAME(THIS.LINK);

END;

NAME(PREV.LINK) = NAME(ANCHOR);

NAME(ANCHOR.LINK) = NAME(THIS);

CLOSE.
## NAME VARIABLES (CONTINUED)

<table>
<thead>
<tr>
<th>Name</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00300044</td>
<td>0006</td>
<td>$E_DU$</td>
</tr>
<tr>
<td>00300045</td>
<td>0007</td>
<td>$L_C$</td>
</tr>
<tr>
<td>00300046</td>
<td>0008</td>
<td>$S_T_H$</td>
</tr>
</tbody>
</table>

This continues the description of variables with their corresponding values and descriptions.
USES OF STRUCTURES AND NAME VARIABLES

(2) TREE-STRUCTURED SYMBOL TABLE

STRUCTURE TREE:
  1 SYMBOL CHARACTER(32),
  1 LESS NAME TREE-STRUCTURE,
  1 GTR NAME TREE-STRUCTURE;
DECLARE TREE TREE-STRUCTURE(100),
NEW_SYMBOL CHARACTER(32),
INTEGER INITIAL(0),
WAS_LESS BOOLEAN;
DECLARE NAME TREE-STRUCTURE,
ROOT INITIAL(NULL), OLD_LEAF, THIS;

NAME(THIS) = NAME(ROOT);
DO WHILE NAME(THIS) /= NAME(NULL);
  NAME(OLD_LEAF) = NAME(THIS);
  IF NEW_SYMBOL < THIS_SYMBOL
    THEN DO; WAS_LESS = TRUE; NAME(THIS) = NAME(THIS.LESS); END;
    ELSE DO; WAS_LESS = FALSE, NAME(THIS) = NAME(THIS.GTR); END;
  END;
  I = I+1; SYMBOL_i = NEW_SYMBOL;
  NAME(LESS_i), NAME(GTR_i) = NAME(NULL);
  IF NAME(ROOT) = NAME(NULL) THEN NAME(ROOT) = NAME(TREE_i);
  ELSE IF WAS_LESS THEN NAME(OLD_LEAF.LESS) = NAME(TREE_i);
  ELSE NAME(OLD_LEAF.GTR) = NAME(TREE_i);
NAME VARIABLES (CONTINUED)

1 MT TREE+PROB:
2 MT PROGRAM:
3 MT STRUCTURE NODE:
4 MT 1 SYMBOL CHARACTER(32),
5 MT 1 LESS NAME NODE-STRUCTURE,
6 MT 1 GTR NAME NODE-STRUCTURE;
7 MT DECLARE TREE NODE-STRUCTURE(100),
8 MT NEW SYMBOL CHARACTER(32),
9 MT 1 INTEGER INITIAL(0),
10 MT WAS+LESS BOOLEAN;
11 MT DECLARE ROOT NAME NODE-STRUCTURE INITIAL(NULL);
12 MT DECLARE OLD+LEAF NAME NODE-STRUCTURE;
13 MT DECLARE THIS NAME NODE-STRUCTURE;
14 MT +
15 MT +
16 NAME(This) = NAME(ROOT);
17 MT +
18 MT DO WHILE NAME(This) \= NAME(NULL);
19 MT +
20 MT NAME(OLD+LEAF) = NAME(This);
21 MT IF NEW SYMBOL < THIS SYMBOL THEN
22 MT DO;
23 MT +
24 MT WAS+LESS = TRUE;
25 MT +
26 MT NAME(This) = NAME(This.LESS);
27 MT END;
28 MT ELSE
29 MT DO;
NAME VARIABLES (CONTINUED)

16 MT
17 MT
18 MT
19 MT
20 MT
21 MT
22 MT
23 MT
24 MT
25 MT
26 MT
27 MT

HAS<LESS = FALSE;
+ "NAME\(THIS\) = NAME(\(THIS.\ OTR\))";
END;

\(i = i + 1;\)

\(\text{TREE. Symbol} = \text{NEW\$Symbol};\)
+ \(\text{NAME}(\text{TREE. LESS} ), \text{NAME}(\text{TREE. OTR} ) = \text{NAME}(\text{NULL});\)

\(\text{IF NAME} (\text{ROOT} ) = \text{NAME}(\text{NULL} ) \text{THEN}\)
+ \(\text{IF NAME} (\text{ROOT} ) = \text{NAME}(\text{TREE} ) .\)

ELSE
IF \(\text{HAS<LESS} \text{THEN}\)
\(\text{NAME}(\text{OLD+LEAF. LESS} ) = \text{NAME}(\text{TREE} ) ;\)
ELSE
\(\text{NAME}(\text{OLD+LEAF. OTR} ) = \text{NAME}(\text{TREE} ) ;\)
<table>
<thead>
<tr>
<th>Address</th>
<th>Variable</th>
<th>Value</th>
<th>Instruction</th>
<th>Segment</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000000</td>
<td>ST#7</td>
<td>0015</td>
<td>LH 2,21(1)</td>
<td>CSECT</td>
<td>EQU *</td>
</tr>
<tr>
<td>00000000</td>
<td>0015</td>
<td></td>
<td>LR 4,2</td>
<td></td>
<td>ROOT</td>
</tr>
<tr>
<td>00000000</td>
<td>ST#7</td>
<td>0017</td>
<td>STH 4,23(1)</td>
<td></td>
<td>THIS</td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#3</td>
<td>0017</td>
<td>LH 2,23(1)</td>
<td>EQU</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td>LR 5,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#3</td>
<td>0017</td>
<td>LH 2,1926(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td>CR 5,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#3</td>
<td>0017</td>
<td>BC 4,4+22</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#5</td>
<td>0004</td>
<td>LH 2,23(1)</td>
<td>EQU</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0004</td>
<td></td>
<td>LR 4,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#5</td>
<td>0004</td>
<td>STH 4,22(1)</td>
<td></td>
<td>OLD+LEAF</td>
</tr>
<tr>
<td>00000000</td>
<td>0004</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#9</td>
<td>0021</td>
<td>LH 2,23(1)</td>
<td>EQU</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0021</td>
<td></td>
<td>LR 4,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#9</td>
<td>0021</td>
<td>STH 4,22(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0021</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#11</td>
<td>0003</td>
<td>SB 3(1),1</td>
<td></td>
<td>WAS+LESS</td>
</tr>
<tr>
<td>00000000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#13</td>
<td>0017</td>
<td>LH 2,23(1)</td>
<td>EQU</td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td>LR 4,2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#13</td>
<td>0017</td>
<td>STH 4,23(1)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#14</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#14</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0011</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#15</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#15</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#17</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#17</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#18</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#18</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#7</td>
<td>0026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0026</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>LBL#7</td>
<td>0026</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0026</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#16</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#16</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0003</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#17</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#17</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#18</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>ST#18</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>00000000</td>
<td>0017</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
NAME VARIABLES (CONTINUED)

<table>
<thead>
<tr>
<th>Address</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>00004F</td>
<td>0002</td>
</tr>
<tr>
<td>030A</td>
<td>9F83</td>
</tr>
<tr>
<td>0338</td>
<td>9FF9</td>
</tr>
<tr>
<td>0002</td>
<td>E0F5</td>
</tr>
<tr>
<td>004F</td>
<td>9559</td>
</tr>
<tr>
<td>0059</td>
<td>0011</td>
</tr>
<tr>
<td>001F</td>
<td>0359</td>
</tr>
<tr>
<td>005C</td>
<td></td>
</tr>
<tr>
<td>0052</td>
<td>9F09</td>
</tr>
<tr>
<td>0055</td>
<td>9FF9</td>
</tr>
<tr>
<td>0155</td>
<td>E0F5</td>
</tr>
<tr>
<td>0053</td>
<td>9A59</td>
</tr>
<tr>
<td>0053</td>
<td>BC4A</td>
</tr>
</tbody>
</table>

ST#26

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST#26</td>
<td>EQU</td>
</tr>
<tr>
<td>LH</td>
<td>7,2(1)</td>
</tr>
<tr>
<td>MII</td>
<td>7,1925(1)</td>
</tr>
<tr>
<td>LA</td>
<td>4,5(7,1)</td>
</tr>
<tr>
<td>LH</td>
<td>2,22(1)</td>
</tr>
<tr>
<td>STH</td>
<td>4,17(2)</td>
</tr>
</tbody>
</table>

ST#27

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>ST#27</td>
<td>EQU</td>
</tr>
<tr>
<td>SC</td>
<td>7,7(1)</td>
</tr>
</tbody>
</table>

LBL#13

<table>
<thead>
<tr>
<th>Variable</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>LBL#13</td>
<td>EQU</td>
</tr>
<tr>
<td>LH</td>
<td>7,2(1)</td>
</tr>
<tr>
<td>MII</td>
<td>7,1925(1)</td>
</tr>
<tr>
<td>LA</td>
<td>4,5(7,1)</td>
</tr>
<tr>
<td>LH</td>
<td>2,22(1)</td>
</tr>
<tr>
<td>STH</td>
<td>4,17(2)</td>
</tr>
</tbody>
</table>
NAME VARIABLES (CONTINUED)

1 MT CPL:
2 MT EXTERNAL COMPOOL:
3 MT STRUCTURE, Q DENSE:
4 MT ( Q1 BIT(3),
5 MT 1 B2 BIT(5),
6 MT 1 S SCALAR,
7 MT 1 V VECTOR,
8 MT 1 M MATRIX,
9 MT DECLARE Q Q-STRUCTURE(10)
10 MT CLOSE:

5 MT PROG:
6 MT PROGRAM:
7 MT DECLARE NO NAME Q-STRUCTURE:
8 MT DECLARE ARRAY(10),
9 MT SCAL SCALAR,
10 MT BITS BIT(5),
11 MT DO FOR TEMPORARY I = 1 TO 10,
12 MT SCAL = 5 * V + M,
13 MT END

14 MT BITS = B1 + B2,
15 MT DO FOR TEMPORARY I = 1 TO 10,
16 MT NAME(No) = NAME(0),
17 MT SCAL = NO.5 + NO. V + NO. M,
18 MT BITS = NO. B1 + NO. B2,
19 MT END
NAME VARIABLES (CONTINUED)

%NAMECOPY

%NAMECOPY(α, β);

PERFORMS THE EQUIVALENT OF:

NAME(α) = NAME(β); (*)

EXCEPT

a MUST BE A NAME STRUCTURE AND β IS EITHER A STRUCTURE OR A NAME STRUCTURE.

Q. SINCE THIS IS MORE RESTRICTIVE THAN A REGULAR NAME ASSIGN (LIKE (*)), WHAT IS THE ADVANTAGE?

A. NO TEMPLATE, COPYNESS, OR LENGTH CHECKING IS DONE.

%NAMECOPY ALLOWS UTILIZATION OF A SINGLE MEMORY AREA FOR A NUMBER OF DISTINCT PURPOSES.
NATURALLY, %NAMECOPY MUST BE USED WITH GREAT CAUTION.

EXAMPLE:

STRUCTURE Q:
    1 A SCALAR DOUBLE;
DECLARE Q Q-STRUCTURE(50);
STRUCTURE R:
    1 B SCALAR;
DECLARE R NAME R-STRUCTURE(100);

%NAMECOPY(R,Q); /* THE A AND B ARRAYS NOW ARE PHYSICALLY
                    COINCIDENT */

\[
\begin{align*}
A\textregistered5 & = A\textregistered3; \\
B\textregistered9 & = B\textregistered5; \\
B\textregistered10 & = B\textregistered6;
\end{align*}
\]

BOTH ACCOMPLISH THE SAME THING!
NAME VARIABLES (CONTINUED)

1 NT CPL:
2 NT COMPOOL RIGID:
3 NT DECLARE I ARRAY(10) INTEGER INITIAL(1, 3, 5, 7, 9, 11, 13, 15, 17, 19);
4 NT DECLARE VECT1 VECTOR INITIAL(10, 20, 30);
5 NT DECLARE VECT2 VECTOR INITIAL(40, 50, 60);
6 NT STRUCTURE INT RIGID:
7 NT -1 TYPE INTEGER,
8 NT 1.0P+CODE INTEGER,
9 NT 1. VARENAME1 NAME :INTEGER,
10 NT 1. VARENAME2 NAME INTEGER;
11 NT STRUCTURE VECT RIGID:
12 NT 1. TYPE INTEGER,
13 NT 1.0P+CODE INTEGER,
14 NT 1. VARENAME1 NAME VECTOR,
15 NT 1. VARENAME2 NAME VECTOR;
16 NT DECLARE S1 INT-STRUCTURE INITIAL(1, 1, NAME(I), NAME(I));
17 NT 2 4
18 NT DECLARE S2 INT-STRUCTURE INITIAL(1, 2, NAME(I), NAME(I));
19 NT 5 7
20 NT DECLARE S3 VECT-STRUCTURE INITIAL(2, 1, NAME(VECT1), NAME(VECT2));
21 NT DECLARE S4 INT-STRUCTURE INITIAL(1, 3, NAME(I), NAME(I));
22 NT 1 2
23 NT DECLARE S5 VECT-STRUCTURE INITIAL(2, 2, NAME(VECT1), NAME(VECT2));
24 NT CLOSE.
NAME VARIABLES (CONTINUED)

12 M+ INTERP
13 M+ PROGRAM;
14 M+ DECLARE N+INT NAME INT-STRUCTURE(5);
15 M+ DECLARE N+VECT NAME VECT-STRUCTURE(5);

   E1
16 M+ NAMECOPY(N+INT, S1);
   E1
17 M+ NAMECOPY(N+VECT, S1);
18 M+ DO FOR TEMPORARY I = 1 TO 5;
19 M+ IF N+INT. TYPE = '1' THEN
20 M+ DO CASE N+INT. OP+CODE :
21 M+     \* INTEGER TYPE */
22 M+     \* ADDITION */
23 M+     \* SUBTRACTION */
24 M+     \* MULTIPLICATION */
25 M+     \* VECTOR TYPE */
26 M+     \* DOT PRODUCT */
27 M+     \* CROSS PRODUCT */
28 M+ ELSE
29 M+ \* DO CASE N+VECT. OP+CODE :
30 M+ \* VECTOR TYPE */
31 M+ \* DOT PRODUCT */
32 M+ \* CROSS PRODUCT */
33 M+ \* ADDITION */
34 M+     \* SUBTRACTION */
35 M+     \* MULTIPLICATION */
36 M+     \* VECTOR TYPE */
37 M+     \* DOT PRODUCT */
38 M+     \* CROSS PRODUCT */
39 M+     \* ADDITION */
40 M+     \* SUBTRACTION */
41 M+     \* MULTIPLICATION */
42 M+     \* VECTOR TYPE */
43 M+ END;
44 M+ END;
45 M+ END;
46 M+ \* CROSS PRODUCT */
47 M+ \* ADDITION */
48 M+ \* SUBTRACTION */
49 M+ \* MULTIPLICATION */
50 M+ \* VECTOR TYPE */
51 M+ \* DOT PRODUCT */
52 M+ \* CROSS PRODUCT */
53 M+ \* ADDITION */
54 M+ \* SUBTRACTION */
55 M+ \* MULTIPLICATION */
56 M+ \* VECTOR TYPE */
57 M+ \* DOT PRODUCT */
58 M+ \* CROSS PRODUCT */
59 M+ \* ADDITION */
60 M+ \* SUBTRACTION */
61 M+ \* MULTIPLICATION */
62 M+ \* VECTOR TYPE */
63 M+ \* DOT PRODUCT */
64 M+ \* CROSS PRODUCT */
65 M+ \* ADDITION */
66 M+ \* SUBTRACTION */
67 M+ \* MULTIPLICATION */
68 M+ \* VECTOR TYPE */
69 M+ \* DOT PRODUCT */
70 M+ \* CROSS PRODUCT */
71 M+ \* ADDITION */
72 M+ \* SUBTRACTION */
73 M+ \* MULTIPLICATION */
74 M+ \* VECTOR TYPE */
75 M+ \* DOT PRODUCT */
76 M+ \* CROSS PRODUCT */
77 M+ \* ADDITION */
78 M+ \* SUBTRACTION */
79 M+ \* MULTIPLICATION */
80 M+ \* VECTOR TYPE */
81 M+ \* DOT PRODUCT */
82 M+ \* CROSS PRODUCT */
83 M+ \* ADDITION */
84 M+ \* SUBTRACTION */
85 M+ \* MULTIPLICATION */
86 M+ \* VECTOR TYPE */
87 M+ \* DOT PRODUCT */
88 M+ \* CROSS PRODUCT */
89 M+ \* ADDITION */
90 M+ \* SUBTRACTION */
91 M+ \* MULTIPLICATION */
92 M+ \* VECT
NAME VARIABLES (CONTINUED)

*ARM 1 E. D. MCHN*:
HCP1 HAL/S DRIVER FOR IBM SHUTTLE GPC SIMULATOR "SIM101"
HCP1: LOAD: GO, FND
HCP1: THE FOLLOWING LOAD PARAMETERS ARE IN EFFECT
HCP1: LOADHM= LOADMOD
HCP1: MEMBER= 00
HCP1: REGION=8K BYTES (260 CORE)
HCP1: SIMREGION=4K, HALFWORDS (AP101 CORE)
HCP1: SIMP:9E=4 (CORE RESIDENT 1K PAGES)
HCP1: HAL/S-FC MODULE, LOADED
  10
  3.2000000E+01
  2.0000000E-01
HCP1: HAL/S DRIVER FOR IBM SHUTTLE GPC SIMULATOR "SIM101"
MESSAGE = 63 DURING SIMULATION INTERRUPT #16, OLD PSW=01E74A00 000
HCP1: SVC OLD PSW= 01E74A00 000101FB
HCP1: INSTRUCTION COUNT=336
HCP1: SIMULATION TIME= 7.39999999999999E-04
HCP1: END OF HAL/S-FC PROGRAM
** SUMMARY OF ERRORS**: **
ERROR SYSTEM (STATEMENT)
10: 0
11
-3.0000000E-02 .6.0000000E+02 -3.0000000E+02
RETURN CODE = 0
FOR REAL TIME PROCESS CONTROL

- High-Level Problem-Oriented Real Time Features
- Careful Definition of Semantics -- for Easy User Comprehension
- Guaranteed Orderly Access to Shared Data and Code
- Hierarchical Relationships Among Processes
PROGRAM and TASK Blocks

A: PROGRAM;

B: TASK;
CLOSE B;

C: TASK;
SCHEDULE B ...
CLOSE C;
SCHEDULE C ...
CLOSE A;
A PROCESS IS:

- The Execution of a PROGRAM or TASK Block
- Made ACTIVE through Execution of a SCHEDULE Statement
- Made INACTIVE through Execution of a CLOSE, CANCEL or TERMINATE Statement
HAL/S

Process State Transitions

1. **Inactive**
   - SCHEDULE
   - TERMINATE
   - CANCEL

2. **Waiting**
   - SCHEDULE
   - CANCEL
   - WAIT
   - Real time event

3. **Executing**
   - Priority process becomes ready
   - Other process
     - Ting or inactive

4. **Ready**
   - Elapsed time
Scheduling of PROGRAM and TASK Blocks

PRIMAL PROGRAM

PROGRAM

COMPOOL
(shared data)

TASK BLOCK

TASK BLOCK

INVOCATION OF TASK BLOCK

INVOCATION OF PROGRAM BLOCK

DATA REFERENCES
EACH PROCESS HAS A PRIORITY
(An Integer Value)

- Initially Specified in SCHEDULE Statement:
  
  SCHEDULE ALPHA PRIORITY (50);

- Dynamically Changed by UPDATE PRIORITY Statement:
  
  UPDATE PRIORITY ALPHA TO (1 + 5);
SCHEDULE A AT 1600 PRIORITY (50);

Alternatively,
SCHEDULE process IN Interval PRIORITY (n);
THE WAIT STATEMENT

- Process is Temporarily Placed in the WAITING State

- WAIT Interval ;

- WAIT UNTIL time ;

WAIT UNTIL IGNITION + 5 ;
A: PROGRAM;

B: TASK;
CLOSE B;

C: TASK;
CLOSE C;

CLOSE A;
FURTHER PROCESS DEPENDENCY

A: PROGRAM;

B: TASK;
   SCHEDULE C ...
   DEPENDENT;
   CLOSE B;

C: TASK;
   CLOSE C;

D: TASK;
   CLOSE D;

CLOSE A;

CLOSE B;

CLOSE C;

CLOSE D;

CLOSE A;
CYCLIC SCHEDULING

A Process can be Scheduled to Execute Its PROGRAM or TASK Block Repeatedly, Until it is:

- Terminated by Execution of a TERMINATE Statement

- "Cancelled" through Execution of a CANCEL Statement, or Because a "Cancellation Condition" has been Met
SCHEDULE A AT 1600 PRIORITY (50), REPEAT EVERY 200 UNTIL 3200;
PROCESS SYNCHRONIZATION

EVENT Variables

- Used for Process Synchronization
- Value TRUE or FALSE
- Combined with Operators AND, OR, NOT to Form EVENT Expressions
EVENT Variables May Be "Latched" or "Unlatched"
SET, SIGNAL, and RESET

- Must Be Used to Change Values of EVENT Variables

- Cause Re-evaluation of EVENT Expressions

- SET and RESET for LATCHED Variables

- SIGNAL for LATCHED or UNLATCHED Variables
Effect of SIGNAL

DECLARE EV1 EVENT,
EV2 EVENT LATCHED INITIAL (TRUE);

SIGNAL EV1;

SIGNAL EV2;
<table>
<thead>
<tr>
<th>Statement</th>
<th>Event</th>
<th>Actual Value</th>
<th>Change sensed by RTE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>execution</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>SET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>latched</td>
<td>T —————— T</td>
<td>F + T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F —————— F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T —————— T</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td>RESET</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>latched</td>
<td>T —————— T</td>
<td>none</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F —————— F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T —————— T</td>
<td>T + F</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F —————— F</td>
<td></td>
</tr>
<tr>
<td></td>
<td>SIGNAL</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>latched</td>
<td>T —————— T</td>
<td>F + T</td>
</tr>
<tr>
<td></td>
<td></td>
<td>F —————— F</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>T —————— T</td>
<td>T + F</td>
</tr>
<tr>
<td></td>
<td>unlatched</td>
<td>T —————— T</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>F —————— F</td>
<td></td>
</tr>
</tbody>
</table>
WAIT FOR EVENT-Expression;

WAIT FOR EV1 & EV2;
INITIATION ON AN EVENT CONDITION

SCHEDULE ALPHA ON EV1 & EV2 PRIORITY (50);
SCHEDULE BETA IN 100 PRIORITY (.50), REPEAT AFTER 50 UNTIL EV1 & EV2;
Shared Data Protection for Dynamic Processes

Data must be protected against simultaneous access and modification:

- Data is accessed in an orderly way
- Results are not dependent on timing coincidences

Data is protected by means of:

- Lock groups
- Update blocks
HAL/S

Shared Data Protection for Dynamic Processes

DECLARE LOCK (1), A, B INTEGER, C VECTOR, D MATRIX;
DECLARE LOCK (2), W, X VECTOR, Y MATRIX;

PROCESS A
  UPDATE;
  A = B;
  C = P + Q;
  CLOSE;

PROCESS B
  UPDATE;
  A = W;
  X = D C;
  CLOSE;

PROCESS C
  UPDATE;
  X = Y P;
  CLOSE;

SWAP TO:

DYNAMICS ➔ ACTIVE

<table>
<thead>
<tr>
<th>Swap</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>STALL</td>
<td>RUN</td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td>STALL</td>
</tr>
<tr>
<td>C</td>
<td>RUN</td>
<td>STALL</td>
<td></td>
</tr>
</tbody>
</table>
HAL/S

Shared Data Protection for Dynamic Processes

PROTECTION IS GUARANTEED

- Compiler Allows References to LOCKED Variables Only in UPDATE Blocks

- RTE Reserves LOCK Groups at Entry to UPDATE Block

- Simultaneous Reservation of All Required LOCK Groups Avoids Deadlock
A SHARED BLOCK MAY BE PROTECTED:

- By Problem-Related Synchronization Measures
- By Preventing Concurrent Execution
- By Allowing Concurrent Execution in Independent Environments

THESE PROTECTION METHODS APPLY TO:

- Ordinary PROCEDUREs and FUNCTIONs
- EXCLUSIVE PROCEDUREs and FUNCTIONs
- REENTRANT PROCEDUREs and FUNCTIONs
Action for EXCLUSIVE Blocks

P: PROCEDURE EXCLUSIVE;
Action for REENTRANT Blocks

P: PROCEDURE REENTRANT;
LOCAL DATA IN REENTRANT BLOCKS

- STATIC Data Is Unique to the Block and Common to All Invocations

- AUTOMATIC Data Is Unique to Each Invocation of the Block

- Parameters and Compiler Temporaries are Effectively AUTOMATIC
Conclusions

HAL/S Has Been Designed and Implemented to Enhance Software Reliability

SUMMARY OF REAL TIME FEATURES:

- Program Structure and Name Scope
- Process States, Priority, Real Time, Dependency
- Cyclic Scheduling
- EVENT Expressions, SET, SIGNAL, and RESET
- LOCK Groups and UPDATE Blocks
- EXCLUSIVE and REENTRANT Code Blocks
HAL/S LEVEL:

DECLARE I INTEGER INITIAL(3),
   NI NAME INTEGER INITIAL (NAME(I));

   NI = NI + 1;

   /* SAME AS I = I + 1; */
USES FOR NAME VARIABLES

- Move around a pointer to a block of data (or code) rather than the whole block.

- Accommodate data structures of dynamically varying size in a fixed-size memory.
PROBLEMS WITH UNRESTRICTED Pointers

DECLARE I INTEGER,
    S SCALAR,
    H NAME;
NAME(H) = NAMECI); /* POINT N AT I */
L: N = N + 1; /* WHAT CODE IS COMPILED? */
NAME(N) = NAME(S); /* POINT N AT S */
IF N < S THEN GO TO L; /* CONVERSION REQUIRED? */
SO, A NAME VARIABLE IS DECLARED TO POINT TO VARIABLES
OF A GIVEN DATA TYPE ...

DECLARE NI NAME INTEGER,
   NS NAME SCALAR,
   NV NAME VECTOR(4),
   NA NAME ARRAY(2,2) BOOLEAN,
DECLARE INTEGER DOUBLE,
   I, J, K,
   N NAME;
   /* FACTORED ATTRIBUTES */
HENCE THE COMPILER CAN CHECK FOR TYPE COMPATIBILITY.

DECLARE I INTEGER,
     S SCALAR,
     N NAME INTEGER;

NAME(N) = NAME(I);
/* POINT N AT I */

L: N = N + 1;
/* CODE FOR INTEGER ADDITION */

NAME(N) = NAME(S);
/* ERROR-TYPE MISMATCH */

IF N < S THEN GO TO L;
/* N DEFERENCED AND CONVERTED TO SCALAR */
NAME VARIABLES CAN POINT TO DATA (OR CODE) WITH

THE FOLLOWING ATTRIBUTES:

| SINGLE OR DOUBLE | OR *
|------------------|-----------------
| INTEGER          | CHARACTER(n)   |
| SCALAR           | BIT(n)         |
| VECTOR(n)        | BOOLEAN        |
| MATRIX(n,m)      | EVENT          |
|                  | PROGRAM        |
|                  | TASK           |

DECLARE MATRIX(2,3), A, B, NB NAME INITIAL(NAME(B));
DECLARE MATRIX(3,3), C, D, ND NAME INITIAL(NAME(D));
A = :NB;               /* SAME AS A = B; */
A = ND;                /* ERROR-DIMENSION MISMATCH */
NAME(NB) = NAME(ND);   /* ERROR-DIMENSION MISMATCH */
NB = ND$(2 AT 1,*)];   /* SAME AS B = D$(2 AT 2,*); */
RESTRICTIONS

- **NAME VARIABLES CANNOT POINT TO NAME VARIABLES**

  **BUT**
  **THEY MAY POINT TO STRUCTURES CONTAINING NAME VARIABLES**

- **CANNOT DECLARE AN ARRAY OF NAME VARIABLES**

  **BUT**
  **THEY MAY APPEAR IN A STRUCTURE WITH COPIES**

- **(SUBTLE) A NAME VARIABLE IN A STRUCTURE WITH COPIES CANNOT POINT TO A STRUCTURE WITH COPIES - CONVERSELY, A NAME VARIABLE WHICH POINTS TO A STRUCTURE WITH COPIES CANNOT BE IN A STRUCTURE WITH COPIES.**
OTHER ATTRIBUTES FOR NAME VARIABLES

DENSE
ALIGNED
RIGID
STATIC
AUTOMATIC
(INITIAL())
(CONSTANT())

TO BE DESCRIBED

THESE APPLY TO THE NAME VARIABLE ITSELF, NOT TO THE VARIABLE WHICH IT POINTS TO.
INITIALIZATION OF NAME VARIABLES

\{
  \text{INITIAL}\}
\{
  \text{CONSTANT}\}
\quad \text{(NAME, REFERENCE, NAME REFERENCE, \ldots)}

\text{EACH NAME REFERENCE HAS THE FORM:}
\quad \name(\text{ORDINARY VARIABLE})
\quad \name(\text{NULL}) \text{ OR NULL}

\text{"NULL POINTER" - POINTS TO NOTHING AT ALL, UNINITIALIZED}
\text{NAME VARIABLES ALSO HAVE THIS VALUE.}

\text{ORDINARY VARIABLE MUST BE:}
\quad \text{- PREVIOUSLY DECLARED}
\quad \text{- WITH DATATYPE MATCHING THAT OF NAME VARIABLE}

\text{AND MUST NOT BE:}
\quad \text{- SUBSCRIPTED}
\quad \text{- DEREFERENCED THROUGH ANOTHER NAME VARIABLE}
INITIALIZATION OF NAME-VARIABLES

1. DECLARE S SCALAR,
   NS NAME SCALAR INITIAL(NAME(S));

2. NAME VARIABLES IN STRUCTURES WILL BE DISCUSSED LATER.
   STRUCTURE A:
   1 B SCALAR,
   1 C NAME A-STRUCTURE,
   DECLARE A-STRUCTURE,
   Z1,
   Z2 INITIAL(1.5, NAME(Z1));
   DECLARE NB NAME SCALAR INITIAL(NAME(Z1,B));

3. DECLARE A ARRAY(5) INTEGER INITIAL(0),
   NI NAME INTEGER INITIAL(NAME(A3));
THE NAME PSEUDO-FUNCTION

- ORDINARY REFERENCE TO NAME VARIABLE ACCESSES THE VARIABLE WHICH IT POINTS TO.
- NAME PSEUDO-FUNCTION IS USED TO ACCESS OR CHANGE THE (POINTER) VALUE OF THE NAME VARIABLE ITSELF.

DECLARE SCALAR, S1, S2, NS NAME, NS1 NAME INITIAL(NAME(S2));

S2 = NS1;

NAME(NS) = NAME(NS1);

NAME(NS) = NAME(S2);

ERROR-TYPE MISMATCH!
IN GENERAL:

- **NAME VARIABLE** BY ITSELF DENOTES THE ORDINARY VARIABLE WHICH IT POINTS TO.

- **NAME(NAME VARIABLE)** OBTAINS THE POINTER CONTENTS OF THE NAME VARIABLE.

- **NAME(ORDINARY VARIABLE)** CREATES A POINTER TO THAT ORDINARY VARIABLE.
NAME ASSIGNMENTS

\[ L^1, L^2, \ldots, L^n = R; \]

\{ EACH \( L^i \) AND \( R \) IS A NAME PSEUDO-FUNCTION.\}

E.G. \( \text{NAME}(NS1), \text{NAME}(NS2) = \text{NAME}(NS); \)

NAME COMPARISONS

\[ L = R \]
\[ L \not= R \]

\{ \( L \) AND \( R \) ARE BOTH NAME PSEUDO-FUNCTIONS.\}

E.G. IF \( \text{NAME}(NS1) \not= \text{NAME}(NS2) \) THEN \( \ldots \);

EQUALITY IF BOTH NAME VARIABLES POINT TO THE SAME ORDINARY VARIABLE.

\[ \text{NS1} \quad \text{NS2} \quad \text{NS} \]

\[ \text{NS1} \quad \text{NS2} \]

\[ \text{NS1} \quad \text{NS2} \]
NAME: ARGUMENTS AND PARAMETERS

P: PROCEDURE(NA) ASSIGN(NB);

DECLARE NA NAME INTEGER,
    NB NAME ARRAY(IO) SCALAR;

CLOSE P;

- ARGUMENT MATCHING NA: NAME PSEUDO-FUNCTION IN REFERENCE CONTEXT (OR NULL).
- ARGUMENT MATCHING NB: NAME PSEUDO-FUNCTION IN ASSIGNMENT CONTEXT.

DECLARE I INTEGER;
DECLARE ARRAY(IO) SCALAR, A, N NAME;

CALL P(NAME(I)) ASSIGN(NAME(N));  // LEGAL
CALL P(NULL) ASSIGN(NAME(N));  // LEGAL
CALL P(NAME(I)) ASSIGN(NAME(A));  // ILLEGAL
CALL P(NULL) ASSIGN(NAME(N_));  // ILLEGAL
POINTERS TO CODE BLOCKS

P1: EXTERNAL PROGRAM;
CLOSE P1;
P2: EXTERNAL PROGRAM;
CLOSE P2;
P3: EXTERNAL PROGRAM;
CLOSE P3;

MASTER: PROGRAM;
DECLARE NP NAME PROGRAM,
   I INTEGER;

DO WHILE TRUE;
   DO FOR I = 1 TO 3;
      DO CASE I;
      NAME(NP) = NAME(P1);
      NAME(NP) = NAME(P2);
      NAME(NP) = NAME(P3);
      END;
   END;
   UPDATE PRIORITY NP TO 200;
   WAIT 1;
   UPDATE PRIORITY NP TO 50;
END;

CLOSE MASTER;
NAME VARIABLES IN STRUCTURES

- NAME VARIABLES MAY POINT TO VARIABLES DECLARED WITH STRUCTURE TEMPLATES.

- NAME VARIABLES MAY BE TERMINAL NODES OF A STRUCTURE TEMPLATE DEFINITION.

- A NAME VARIABLE IN A STRUCTURE TEMPLATE MAY POINT TO THE TEMPLATE CURRENTLY BEING DEFINED.

STRUCTURE LIST:

1. VALUE INTEGER,
2. LINK NAME LIST-STRUCTURE;
STRUCTURE LIST:
1 VALUE INTEGER,
1 LINK NAME LIST-STRUCTURE;
DECLARE LIST-STRUCTURE,
Z1, Z2, Z3,
Z0 NAME;

NAME(Z0)

NAME(Z1.LINK) or
NAME(Z0.LINK)

dereferencing

VALUE

Z1.VALUE or
Z0.VALUE

dereferencing

LINK

NAME(Z2.LINK) or
NAME(Z1.LINK.LINK) or
NAME(Z0.LINK.LINK)

dereferencing

VALUE

LINK

NULL

VALUE

Z2.VALUE or
Z1.LINK.VALUE or
Z0.LINK.VALUE

dereferencing

Z3.VALUE or
Z2.LINK.VALUE or
Z1.LINK.LINK.VALUE or
Z0.LINK.LINK.VALUE

dereferencing

Z2.LINK.VALUE or
Z1.LINK.VALUE or
Z0.LINK.VALUE

dereferencing

Z3.VALUE or
Z2.LINK.VALUE or
Z1.LINK.LINK.VALUE or
Z0.LINK.LINK.VALUE

dereferencing
BEST WAY TO THINK OF THIS:

- ANY APPEARANCE OF A NAME VARIABLE IMPLIES DEREFERENCING.
- APPLICATION OF NAME PSEUDO-FUNCTION IMPLIES ONE LEVEL OF "RE-REFERENCING".

THUS

Z0
NAME(Z0)  "BACKS UP" TO Z0 ITSELF
Z1.LINK.LINK
NAME(Z1.LINK.LINK)  "BACKS UP" TO Z2.LINK (OR Z1.LINK.LINK) ITSELF

NOTE THAT Z0.VALUE IS THE SAME AS Z1.VALUE SINCE Z0 IS A NAME VARIABLE BUT Z1 IS NOT.

FURTHER, NAME(Z0.VALUE) IS THE SAME AS NAME(Z1.VALUE) I.E. A POINTER TO THE VALUE FIED IN Z1.
USES OF STRUCTURES AND NAME VARIABLES

① A PRIORITY-ORDERED QUEUE

STRUCTURE QUEUE:
1 IDNO INTEGER,
1 PRTY INTEGER,
1 LINK NAME QUEUE-STRUCTURE;
DECLARE QUEUE QUEUE-STRUCTURE(10),
ANCHOR NAME QUEUE-STRUCTURE;
DECLARE INTEGER, I, NEW_ID, NEW_PRTY;

DO FOR I = 1 TO 9;
IDNO\_I, PRTY\_I = I;
NAME(LINK\_I) = NAME(Queue\_I+1);
END;
IDNO\_10, PRTY\_10 = 10;
NAME(ANCHOR), NAME(LINK\_10) = NAME(Queue\_1);
USES OF STRUCTURES AND NAME VARIABLES

DECLARE NAME QUEUE-STRUCTURE, THIS, PREV

/* FIND IDNO IN QUEUE */
NAME(PREV) = NAME(ANCHOR);
NAME(THIS) = NAME(ANCHOR, LINK);

DO WHILE THIS.IDNO = NEW_IDNO;
    NAME(PREV) = NAME(THIS);
    NAME(THIS) = NAME(THIS, LINK);
END;

/* SET NEW PRIORITY, AND TEMPORARILY UNLINK FROM QUEUE */
THIS.PRTY = NEW_PRTY;
NAME(ANCHOR), NAME(PREV, LINK) = NAME(THIS, LINK);

/* FIND PROPER PLACE TO RE-LINK IN QUEUE ACCORDING TO PRIORITY */
DO UNTIL PREV.PRTY <= NEW_PRTY AND NEW_PRTY <= THIS.PRTY;
    NAME(PREV) = NAME(THIS);
    NAME(THIS) = NAME(THIS, LINK);
END;
NAME(PREV, LINK) = NAME(ANCHOR);
NAME(ANCHOR, LINK) = NAME(THIS);
USES OF STRUCTURES AND NAME VARIABLES

TREE-STRUCTURED SYMBOL TABLE

STRUCTURE TREE:
1 SYMBOL CHARACTER(32),
1 LESS NAME TREE-STRUCTURE,
1 GTR NAME TREE-STRUCTURE;
DECLARE TREE TREE-STRUCTURE(100),
NEW_SYMBOL CHARACTER(32),
1 INTEGER INITIAL(0),
WAS_LESS BOOLEAN;
DECLARE NAME TREE-STRUCTURE,
ROOT INITIAL(NULL), OLD_LEAF, THIS;
NAME(THIS) = NAME(ROOT);
DO WHILE NAME(THIS) /= NAME(NULL);
   NAME(OLD_LEAF) = NAME(THIS);
   IF NEW_SYMBOL < THIS, SYMBOL
      THEN DO; WAS_LESS = TRUE; NAME(THIS) = NAME(THIS, LESS); END;
   ELSE DO; WAS_LESS = FALSE; NAME(THIS) = NAME(THIS, GTR); END;
END;
1 = 1+1; SYMBOL_1 = NEW_SYMBOL;
NAME(LESS_1), NAME(GTR_1) = NAME(NULL);
IF NAME(ROOT) = NAME(NULL) THEN NAME(ROOT) = NAME(TREE_1);
ELSE IF WAS_LESS THEN NAME(OLD_LEAF, LESS) = NAME(TREE_1);
ELSE NAME(OLD_LEAF, GTR) = NAME(TREE_1);
SUBSCRIPTING AND NAME VARIABLES

- **SUBSCRIPTING OF DEREFERENCED NAME VARIABLES IS ALLOWED**
  E.G. DECLARE ARRAY(3) VECTOR, V, NV NAME INITIAL(NAME(V));
  \[ V_{1,3} = NV_{2,3}; \] /* SAME AS \( V_{1,3} = V_{2,3}; */

- **SUBSCRIPTING INSIDE A REFERENCE TO THE NAME PSEUDO-FUNCTION**
  CAN ONLY APPEAR IN REFERENCE CONTEXT (NOT IN ASSIGNMENT CONTEXT).
  SUBSCRIPTING APPLIES TO THE VARIABLE BEING POINTED TO.

E.G. DECLARE INTEGER

A ARRAY(10),
NA NAME ARRAY(10) INITIAL(NAME(A)),
NI NAME; /* NI IS NAME OF SCALAR */

NAME(NI) = NAME(A_3),
NAME(NI) = NAME(NA_3),
COMPONENT SUBSCRIPTING INSIDE NAME( )

- ILLEGAL FOR BIT AND CHARACTER STRINGS
  E.G. DECLARE B BIT(16); NAME(B5)

- MUST SELECT A SINGLE SCALAR FROM VECTORS AND MATRICES (ALSO SINGLE ELEMENT FROM ARRAYS)
  E.G. DECLARE M MATRIX;

![Diagram of component subscripting inside NAME( )]

- [Diagram showing component subscripting with indices and elements]

1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5
1 2 3 4 5

STRUCTURE SUBSCRIPTING INSIDE NAME()

- In assignment context, ok only if NAME() is applied to a name variable in a structure with multiple copies - then it selects the appropriate copy of the name variable itself!

E.g.  

```
STRUCTURE S: 1 N NAME SCALAR;
DECLARE S S-STRUCTURE(10);
NAME(N_{1j}) = NAME(N_{2j});
```

- In reference context, only one can:
  - Name variable defined in a structure with copies, subscripting is effective on the name variable itself.
  - Name variable pointing to a structure with copies, subscripting is effective on the variable being pointed to.

```
STRUCTURE S: 1 N NAME-STRUCTURE(10);  // ILLEGAL!
```