Rules for Arithmetic Assignment

Rules for Arithmetic Assignment

This chapter covers the following topics:

- Field Initialization
- Data Transfer
- Field Truncation and Field Rounding
- Result Format and Length in Arithmetic Operations
- Arithmetic Operations with Floating-Point Numbers
- Arithmetic Operations with Date and Time
- Performance Considerations for Mixed Format Expressions
- Precision of Results for Arithmetic Operations
- Error Conditions in Arithmetic Operations
- Processing of Arrays

Field Initialization

A field (user-defined variable or database field) which is to be used as an operand in an arithmetic operation must be defined with one of the following formats:

Format	
N	Numeric unpacked
P	Packed numeric
I	Integer
F	Floating point
D	Date
T	Time

Note:
For reporting mode: A field which is to be used as an operand in an arithmetic operation must have been previously defined. A user-defined variable or database field used as a result field in an arithmetic operation need not have been previously defined.

All user-defined variables and all database fields defined in a `DEFINE DATA` statement are initialized to the appropriate zero or blank value when the program is invoked for execution.

Data Transfer

Data transfer is performed with a `MOVE` or `COMPUTE` statement. The following table summarizes the data transfer compatibility of the formats an operand may take.

Sending Field Format	**Receiving Field Format**												
	N or P	**A**	**U**	**B*n* (*n*<5)**	**B*n* (*n*>4)**	**I**	**L**	**C**	**D**	**T**	**F**	**G**	**O**
N or P	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
A	-	Y	[13]	[1]	[1]	-	-	-	-	-	-	-	-
U	-	[11]	Y	[12]	[12]	-	-	-	-	-	-	-	-
B*n* (*n*<5)	[4]	[2]	[14]	[5]	[5]	Y	-	-	-	Y	Y	-	-
B*n* (*n*>4)	-	[6]	[15]	[5]	[5]	-	-	-	-	-	-	-	-
I	Y	[2]	[14]	[3]	-	Y	-	-	-	Y	Y	-	-
L	-	[9]	[16]	-	-	-	Y	-	-	-	-	-	-
C	-	-	-	-	-	-	-	Y	-	-	-	-	-
D	Y	[9]	[16]	Y	-	Y	-	-	Y	[7]	Y	-	-
T	Y	[9]	[16]	Y	-	Y	-	-	[8]	Y	Y	-	-
F	Y	[9] [10]	[10] [16]	[3]	-	Y	-	-	-	Y	Y	-	-
G	-	-	-	-	-	-	-	-	-	-	-	Y	-
O	-	-	-	-	-	-	-	-	-	-	-	-	Y

Where:

Y	Indicates data transfer compatibility.
-	Indicates data transfer incompatibility.
[ ]	Numbers in brackets [ ] refer to the corresponding rule for data transfer given below.

Data Conversion

The following rules apply to converting data values:

1. **Alphanumeric to binary:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blank characters depending on the length defined and the number of bytes specified.

2. **(N,P,I) and binary (length 1-4) to alphanumeric:**
   The value will be converted to unpacked form and moved into the alphanumeric field left justified, that is, leading zeros will be suppressed and the field will be filled with trailing blank characters. For negative numeric values, the sign will be converted to the hexadecimal notation `D`*x*. Any decimal

point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.

3. **(N,P,I,F) to binary (1-4 bytes):**
   The numeric value will be converted to binary (4 bytes). Any decimal point in the numeric value will be ignored (the digits of the value before and after the decimal point will be treated as an integer value). The resulting binary number will be positive or a two's complement of the number depending on the sign of the value.

4. **Binary (1-4 bytes) to numeric:**
   The value will be converted and assigned to the numeric value right justified, that is, with leading zeros. (Binary values of the length 1-3 bytes are always assumed to have a positive sign. For binary values of 4 bytes, the leftmost bit determines the sign of the number: 1=negative, 0=positive.) Any decimal point in the receiving numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value.

5. **Binary to binary:**
   The value will be moved from right to left byte by byte. Leading binary zeros will be inserted into the receiving field.

6. **Binary (>4 bytes) to alphanumeric:**
   The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks depending on the length defined and the number of bytes specified.

7. **Date (D) to time (T):**
   If date is moved to time, it is converted to time assuming time 00:00:00:0.

8. **Time (T) to date (D):**
   If time is moved to date, the time information is truncated, leaving only the date information.

9. **L,D,T,F to A:**
   The values are converted to display form and are assigned left justified.

10. **F:**
    If F is assigned to an alphanumeric or Unicode field which is too short, the mantissa is reduced accordingly.

11. **Unicode to alphanumeric:**
    The Unicode value will be converted to alphanumeric character codes according to the default code page (value of the system variable `*CODEPAGE`) using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified.

12. **Unicode to binary:**
    The value will be moved code unit by code unit from left to right. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of bytes specified. The length of the receiving binary field must be even.

13. **Alphanumeric to Unicode:**
    The alphanumeric value will be converted from the default code page to a Unicode value using the International Components for Unicode (ICU) library. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.

14. **(N,P,I) and binary (length 1-4) to Unicode:**
    The value will be converted to unpacked form from which an alphanumeric value will be obtained by suppression of leading zeros. For negative numeric values, the sign will be converted to the hexadecimal notation `Dx`. Any decimal point in the numeric value will be ignored. All digits before and after the decimal point will be treated as one integer value. The resulting value will be converted from alphanumeric to Unicode. The result may be truncated or padded with trailing blank characters, depending on the length defined and the number of code units specified.

15. **Binary (>4 bytes) to Unicode:**
    The value will be moved byte by byte from left to right. The result may be truncated or padded with trailing blanks, depending on the length defined and the number of bytes specified. The length of the sending binary field must be even.

16. **L,D,T,F to U:**
    The values are converted to an alphanumeric display form. The resulting value will be converted from alphanumeric to Unicode and assigned left justified.

If source and target format are identical, the result may be truncated or padded with trailing blank characters (format A and U) or leading binary zeros (format B) depending on the length defined and the number of bytes (format A and B) or code units (format U) specified.

See also *Using Dynamic Variables*.

Field Truncation and Field Rounding

The following rules apply to field truncation and rounding:

- High-order numeric field truncation is allowed only when the digits to be truncated are leading zeros. Digits following an expressed or implied decimal point may be truncated.
- Trailing positions of an alphanumeric field may be truncated.
- If the option `ROUNDED` is specified, the last position of the result will be rounded up if the first truncated decimal position of the value being assigned contains a value greater than or equal to 5. For the result precision of a division, see also *Precision of Results for Arithmetic Operations*.

Result Format and Length in Arithmetic Operations

The following table shows the format and length of the result of an arithmetic operation:

	I1	**I2**	**I4**	**N or P**	**F4**	**F8**
I1	I1	I2	I4	P*	F4	F8
I2	I2	I2	I4	P*	F4	F8
I4	I4	I4	I4	P*	F4	F8
N or P	P*	P*	P*	P*	F4	F8
F4	F4	F4	F4	F4	F4	F8
F8	F8	F8	F8	F8	F8	F8

On a mainframe computer, format/length F8 is used instead of F4 for improved precision of the results of an arithmetic operation.

P* is determined from the integer length and precision of the operands individually for each operation, as shown under *Precision of Results for Arithmetic Operations*.

The following decimal integer lengths and possible values are applicable for format I:

Format/Length	**Decimal Integer Length**	**Possible Values**
I1	3	`-128` to `127`
I2	5	`-32768` to `32767`
I4	10	`-2147483648` to `2147483647`

Arithmetic Operations with Floating-Point Numbers

The following topics are covered below:

- General Considerations
- Precision of Floating-Point Numbers
- Conversion to Floating-Point Representation
- Platform Dependency

General Considerations

Floating-point numbers (format F) are represented as a sum of powers of two (as are integer numbers (format I)), whereas unpacked and packed numbers (formats N and P) are represented as a sum of powers of ten.

In unpacked or packed numbers, the position of the decimal point is fixed. In floating-point numbers, however, the position of the decimal point (as the name indicates) is "floating", that is, its position is not fixed, but depends on the actual value.

Floating-point numbers are essential for the computing of trigonometric functions or mathematical functions such as sinus or logarithm.

Precision of Floating-Point Numbers

Due to the nature of floating-point numbers, their precision is limited:

- For a variable of format/length F4, the precision is limited to approximately 7 digits.
- For a variable of format/length F8, the precision is limited to approximately 15 digits.

Values which have more significant digits cannot be represented exactly as a floating-point number. No matter how many additional digits there are before or after the decimal point, a floating-point number can cover only the leading 7 or 15 digits respectively.

An integer value can only be represented exactly in a variable of format/length F4 if its absolute value does not exceed $2^{23}-1$.

Conversion to Floating-Point Representation

When an alphanumeric, unpacked numeric or packed numeric value is converted to floating-point format (for example, in an assignment operation), the representation has to be changed, that is, a sum of powers of ten has to be converted to a sum of powers of two.

Consequently, only numbers that are representable as a finite sum of powers of two can be represented exactly; all other numbers can only be represented approximately.

Examples:

This number has an exact floating-point representation:

$$1.25 = 2^{0} + 2^{-2}$$

This number is a periodic floating-point number without an exact representation:

$$1.2 = 2^{0} + 2^{-3} + 2^{-4} + 2^{-7} + 2^{-8} + 2^{-11} + 2^{-12} + \ldots$$

Thus, the conversion of alphanumeric, unpacked numeric or packed numeric values to floating-point values, and vice versa, can introduce small errors.

Platform Dependency

Because of different hardware architecture, the representation of floating-point numbers varies according to platforms. This explains why the same application, when run on different platforms, may return slightly different results when floating-point arithmetics are involved. The respective representation also determines the range of possible values for floating-point variables, which is (approximately)

- $\pm 1.17 * 10^{-38}$ to $\pm 3.40 * 10^{38}$ for F4 variables,
- $\pm 2.22 * 10^{-308}$ to $\pm 1.79 * 10^{308}$ for F8 variables.

Note:
The representation used by your pocket calculator may also be different from the one used by your computer - which explains why results for the same computation may differ.

Arithmetic Operations with Date and Time

With formats D (date) and T (time), only addition, subtraction, multiplication and division are allowed. Multiplication and division are allowed on intermediate results of additions and subtractions only.

Date/time values can be added to/subtracted from one another; or integer values (no decimal digits) can be added to/subtracted from date/time values. Such integer values can be contained in fields of formats N, P, I, D, or T.

The intermediate results of such an addition or subtraction may be used as a multiplicand or dividend in a subsequent operation.

An integer value added to/subtracted from a date value is assumed to be in days. An integer value added to/subtracted from a time value is assumed to be in tenths of seconds.

For arithmetic operations with date and time, certain restrictions apply, which are due to the Natural's internal handling of arithmetic operations with date and time, as explained below.

Internally, Natural handles an arithmetic operation with date/time variables as follows:

COMPUTE *result-field = operand1 +/- operand2*

The above statement is resolved as:

1. *intermediate-result = operand1 +/- operand2*
2. *result-field = intermediate-result*

That is, in a first step Natural computes the result of the addition/subtraction, and in a second step assigns this result to the result field.

More complex arithmetic operations are resolved following the same pattern:

COMPUTE *result-field = operand1 +/- operand2 +/- operand3 +/- operand4*

The above statement is resolved as:

1. *intermediate-result1 = operand1 +/- operand2*
2. *intermediate-result2 = intermediate-result1 +/- operand3*
3. *intermediate-result3 = intermediate-result2 +/- operand4*
4. *result-field = intermediate-result3*

The resolution of multiplication and division operations is similar to the resolution for addition and subtraction.

The internal format of such an intermediate result depends on the formats of the operands, as shown in the tables below.

Addition

The following table shows the format of the intermediate result of an addition (*intermediate-result = operand1 + operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	D	Di
D	T	T
D	Di, Ti, N, P, I	D
T	D, T, Di, Ti, N, P, I	T
Di, Ti, N, P, I	D	D
Di, Ti, N, P, I	T	T
Di, N, P, I	Di	Di
Ti, N, P, I	Ti	Ti
Di	Ti, N, P, I	Di
Ti	Di, N, P, I	Ti

Subtraction

The following table shows the format of the intermediate result of a subtraction (*intermediate-result = operand1 - operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	D	Di
D	T	Ti
D	Di, Ti, N, P, I	D
T	D, T	Ti
T	Di, Ti, N, P, I	T
Di, N, P, I	D	Di
Di, N, P, I	T	Ti
Di	Di, Ti, N, P, I	Di
Ti	D, T, Di, Ti, N, P, I	Ti
N, P, I	Di, Ti	P12

Multiplication or Division

The following table shows the format of the intermediate result of a multiplication (*intermediate-result = operand1 * operand2*) or division (*intermediate-result = operand1 / operand2*):

Format of *operand1*	Format of *operand2*	Format of *intermediate-result*
D	D, Di, Ti, N, P, I	Di
D	T	Ti
T	D, T, Di, Ti, N, P, I	Ti
Di	T	Ti
Di	D, Di, Ti, N, P, I	Di
Ti	D	Di
Ti	Di, T, Ti, N, P, I	Ti
N, P, I	D, Di	Di
N, P, I	T, Ti	Ti

Internal Assignments

Di is a value in internal date format; Ti is a value in internal time format; such values can be used in further arithmetic date/time operations, but they cannot be assigned to a result field of format D (see the assignment table below).

In complex arithmetic operations in which an intermediate result of internal format Di or Ti is used as operand in a further addition/subtraction/multiplication/division, its format is assumed to be D or T respectively.

The following table shows which intermediate results can internally be assigned to which result fields (*result-field = intermediate-result*).

Format of *result-field*	Format of *intermediate-result*	Assignment possible
D	D, T	yes
D	Di, Ti, N, P, I	no
T	D, T, Di, Ti, N, P, I	yes
N, P, I	D, T, Di, Ti, N, P, I	yes

A result field of format D or T must not contain a negative value.

Examples 1 and 2 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D)
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Di, and a value of format D*i* cannot be assigned to a result field of format D.

Examples 3 and 4 (invalid):

```
COMPUTE DATE1 (D) = TIME2 (T) - TIME3 (T)
COMPUTE DATE1 (D) = DATE2 (D) - TIME3 (T)
```

These operations are not possible, because the intermediate result of the addition/subtraction would be format Ti, and a value of format Ti cannot be assigned to a result field of format D.

Example 5 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) - DATE3 (D) + TIME3 (T)
```

This operation is possible. First, `DATE3` is subtracted from `DATE2`, giving an intermediate result of format Di; then, this intermediate result is added to `TIME3`, giving an intermediate result of format T; finally, this second intermediate result is assigned to the result field `DATE1`.

Examples 6 and 7 (invalid):

```
COMPUTE DATE1 (D) = DATE2 (D) + DATE3 (D) * 2
COMPUTE TIME1 (T) = TIME2 (T) - TIME3 (T) / 3
```

These operations are not possible, because the attempted multiplication/division is performed with date/time fields and not with intermediate results.

Example 8 (valid):

```
COMPUTE DATE1 (D) = DATE2 (D) + (DATE3(D) - DATE4 (D)) * 2
```

This operation is possible. First, `DATE4` is subtracted from `DATE3` giving an intermediate result of format Di; then, this intermediate result is multiplied by two giving an intermediate result of format Di; this intermediate result is added to `DATE2` giving an intermediate result of format D; finally, this third intermediate result is assigned to the result field `DATE1`.

If a format T value is assigned to a format D field, you must ensure that the time value contains a valid date component.

Performance Considerations for Mixed Format Expressions

When doing arithmetic operations, the choice of field formats has considerable impact on performance:

For business arithmetic, only fields of format I (integer) should be used, if possible.

For scientific arithmetic, fields of format F (floating point) should be used, if possible.

In expressions where formats are mixed between numeric (N, P) and floating point (F), a conversion to floating point format is performed. This conversion results in considerable CPU load. Therefore it is recommended to avoid mixed format expressions in arithmetic operations.

Precision of Results for Arithmetic Operations

Operation	Digits Before Decimal Point	Digits After Decimal Point
Addition/Subtraction	Fi + 1 or Si + 1 (whichever is greater)	Fd or Sd (whichever is greater)
Multiplication	Fi + Si + 2	Fd + Sd (maximum 7)
Division	Fi + Sd	(see below)
Exponentiation	15 - Fd (See *Exception* below)	Fd
Square Root	Fi	Fd

- where:

F	First operand
S	Second operand
R	Result
i	Digits before decimal point
d	Digits after decimal point

Exception:

If the exponent has one or more digits after the decimal point, the exponentiation is internally carried out in floating point format and the result will also have floating point format. See *Arithmetic Operations with Floating-Point Numbers* for further information.

Digits after Decimal Point for Division Results

The precision of the result of a division depends whether a result field is available or not:

- If a result field is available, the precision is: Rd or Fd (whichever is greater) *.
- If no result field is available, the precision is: Fd or Sd (whichever is greater) *.

* If the `ROUNDED` option is used, the precision of the result is internally increased by one digit before the result is actually rounded.

A result field is available (or assumed to be available) in a `COMPUTE` and `DIVIDE` statement, and in a logical condition in which the division is placed after the comparison operator (for example: `IF #A = #B / #C THEN ...`).

A result field is not (or not assumed to be) available in a logical condition in which the division is placed before the comparison operator (for example: `IF #B / #C = #A THEN ...`).

Exception:

If both dividend and divisor are of integer format and at least one of them is a variable, the division result is always of integer format (regardless of the precision of the result field and of whether the ROUNDED option is used or not).

Precision of Results for Arithmetic Expressions

The precision of arithmetic expressions, for example: `#A / (#B * #C) + #D * (#E - #F + #G)`, is derived by evaluating the results of the arithmetic operations in their processing order. For further information on arithmetic expressions, see *arithmetic-expression* in the COMPUTE statement description.

Error Conditions in Arithmetic Operations

In an addition, subtraction, multiplication or division, an error occurs if the total number of digits (before and after the decimal point) of the result is greater than 31.

In an exponentiation, an error occurs in any of the following situations:

- if the base is of packed format and either the result has over 16 digits or any intermediate result has over 15 digits;
- if the base is of floating-point format and the result is greater than approximately $7 * 10^{75}$.

Processing of Arrays

Generally, the following rules apply:

- All scalar operations may be applied to array elements which consist of a single occurrence.
- If a variable is defined with a constant value (for example, `#FIELD (I2) CONSTANT <8>`), the value will be assigned to the variable at compilation, and the variable will be treated as a constant. This means that if such a variable is used in an array index, the dimension concerned has a *definite* number of occurrences.
- If an assignment/comparison operation involves two arrays with a different number of dimensions, the "missing" dimension in the array with fewer dimensions is assumed to be (1:1).

  Example: If `#ARRAY1 (1:2)` is assigned to `#ARRAY2 (1:2,1:2)`, `#ARRAY1` is assumed to be `#ARRAY1 (1:1,1:2)`.

The following topics are covered below:

- Definitions of Array Dimensions
- Assignment Operations with Arrays
- Comparison Operations with Arrays

- Arithmetic Operations with Arrays

Definitions of Array Dimensions

The first, second and third dimensions of an array are defined as follows:

Number of Dimensions	Properties
3	#a3(3rd dim., 2nd dim., 1st dim.)
2	#a2(2nd dim., 1st dim.)
1	#a1(1st dim.)

Assignment Operations with Arrays

If an array range is assigned to another array range, the assignment is performed element by element.

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE
*
MOVE #ARRAY(2:4) TO #ARRAY(3:5)
/* is identical to
/* MOVE #ARRAY(2) TO #ARRAY(3)
/* MOVE #ARRAY(3) TO #ARRAY(4)
/* MOVE #ARRAY(4) TO #ARRAY(5)
/*
/* #ARRAY contains  10,20,20,20,20
```

If a single occurrence is assigned to an array range, each element of the range is filled with the value of the single occurrence. (For a mathematical function, each element of the range is filled with the result of the function.)

Before an assignment operation is executed, the individual dimensions of the arrays involved are compared with one another to check if they meet one of the conditions listed below. The dimensions are compared independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The assignment of values from one array to another is only allowed under one of the following conditions:

- The number of occurrences is the same for both dimensions compared.
- The number of occurrences is indefinite for both dimensions compared.
- The dimension that is assigned to another dimension consists of a single occurrence.

Example - Array Assignments:

The following program shows which array assignment operations are possible.

```
DEFINE DATA LOCAL
1 A1   (N1/1:8)
1 B1   (N1/1:8)
1 A2   (N1/1:8,1:8)
1 B2   (N1/1:8,1:8)
1 A3   (N1/1:8,1:8,1:8)
1 I    (I2)          INIT <4>
1 J    (I2)          INIT <8>
1 K    (I2)          CONST <8>
END-DEFINE
*
COMPUTE A1(1:3) = B1(6:8)                                  /* allowed
COMPUTE A1(1:I) = B1(1:I)                                  /* allowed
COMPUTE A1(*)   = B1(1:8)                                  /* allowed
COMPUTE A1(2:3) = B1(I:I+1)                                /* allowed
COMPUTE A1(1)   = B1(I)                                    /* allowed
COMPUTE A1(1:I) = B1(3)                                    /* allowed
COMPUTE A1(I:J) = B1(I+2)                                  /* allowed
COMPUTE A1(1:I) = B1(5:J)                                  /* allowed
COMPUTE A1(1:I) = B1(2)                                    /* allowed
COMPUTE A1(1:2) = B1(1:J)                                  /* NOT ALLOWED (NAT0631)
COMPUTE A1(*)   = B1(1:J)                                  /* NOT ALLOWED (NAT0631)
COMPUTE A1(*)   = B1(1:K)                                  /* allowed
COMPUTE A1(1:J) = B1(1:K)                                  /* NOT ALLOWED (NAT0631)
*
COMPUTE A1(*)       = B2(1,*)                              /* allowed
COMPUTE A1(1:3)     = B2(1,I:I+2)                          /* allowed
COMPUTE A1(1:3)     = B2(1:3,1)                            /* NOT ALLOWED (NAT0631)
*
COMPUTE A2(1,1:3)   = B1(6:8)                              /* allowed
COMPUTE A2(*,1:I)   = B1(5:J)                              /* allowed
COMPUTE A2(*,1)     = B1(*)                                /* NOT ALLOWED (NAT0631)
COMPUTE A2(1:I,1)   = B1(1:J)                              /* NOT ALLOWED (NAT0631)
COMPUTE A2(1:I,1:J) = B1(1:J)                              /* allowed
*
COMPUTE A2(1,I)     = B2(1,1)                              /* allowed
COMPUTE A2(1:I,1)   = B2(1:I,2)                            /* allowed
COMPUTE A2(1:2,1:8) = B2(I:I+1,*)                          /* allowed
*
COMPUTE A3(1,1,1:I)   = B1(1)                              /* allowed
COMPUTE A3(1,1,1:J)   = B1(*)                              /* NOT ALLOWED (NAT0631)
 COMPUTE A3(1,1,1:I)   = B1(1:I)                           /* allowed
COMPUTE A3(1,1:2,1:I) = B2(1,1:I)                          /* allowed
COMPUTE A3(1,1,1:I)   = B2(1:2,1:I)                        /* NOT ALLOWED (NAT0631)
END
```

Comparison Operations with Arrays

Generally, the following applies: if arrays with multiple dimensions are compared, the individual dimensions are handled independently of one another; that is, the 1st dimension of the one array is compared with the 1st dimension of the other array, the 2nd dimension of the one array is compared with the 2nd dimension of the other array, and the 3rd dimension of the one array is compared with the 3rd dimension of the other array.

The comparison of two array dimensions is only allowed under one of the following conditions:

- The array dimensions compared with one another have the same number of occurrences.
- The array dimensions compared with one another have an indefinite number of occurrences.
- All array dimensions of one of the arrays involved are single occurrences.

Example - Array Comparisons:

The following program shows which array comparison operations are possible:

```
DEFINE DATA LOCAL
1 A3  (N1/1:8,1:8,1:8)
1 A2  (N1/1:8,1:8)

1 A1  (N1/1:8)
1 I   (I2)   INIT <4>
1 J   (I2)   INIT <8>
1 K   (I2)   CONST <8>
END-DEFINE
*
IF A2(1,1)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(*)              THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(I -3:I+4)       THEN IGNORE END-IF /* allowed
IF A2(1,5:J)   = A1(1:I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A1(1:I)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,*)     = A1(1:K)            THEN IGNORE END-IF /* allowed
*
IF A2(1,1)     = A2(1,1)            THEN IGNORE END-IF /* allowed
IF A2(1,1)     = A2(1,I)            THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(1,1:8)          THEN IGNORE END-IF /* allowed
IF A2(1,*)     = A2(I,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I+1:J)        THEN IGNORE END-IF /* allowed
IF A2(1,1:I)   = A2(1,I:I+1)        THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(*,1)     = A2(1,*)            THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
IF A2(1,1:I)   = A1(2,1:K)          THEN IGNORE END-IF /* NOT ALLOWED(NAT0629)
*
IF A3(1,1,*)   = A2(1,*)            THEN IGNORE END-IF /* allowed
IF A3(1,1,*)   = A2(1,I -3:I+4)     THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,1:I+1)        THEN IGNORE END-IF /* allowed
IF A3(1,*,I:J) = A2(*,I:J)          THEN IGNORE END-IF /* allowed
END
```

When you compare two array ranges, note that the following two expressions lead to different results:

```
#ARRAY1(*) NOT EQUAL #ARRAY2(*)
NOT #ARRAY1(*) = #ARRAY2(*)
```

Example:

- **Condition A:**

  ```
  IF #ARRAY1(1:2) NOT EQUAL #ARRAY2(1:2)
  ```

This is equivalent to:

```
IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) AND (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
```

Condition A is therefore true if the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *and* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

- **Condition B:**

  ```
  IF NOT #ARRAY1(1:2) = #ARRAY2(1:2)
  ```

  This is equivalent to:

  ```
  IF NOT (#ARRAY1(1)= #ARRAY2(1) AND #ARRAY1(2) = #ARRAY2(2))
  ```

  This in turn is equivalent to:

  ```
  IF (#ARRAY1(1) NOT EQUAL #ARRAY2(1)) OR (#ARRAY1(2) NOT EQUAL #ARRAY2(2))
  ```

  Condition B is therefore true if *either* the first occurrence of #ARRAY1 does not equal the first occurrence of #ARRAY2 *or* the second occurrence of #ARRAY1 does not equal the second occurrence of #ARRAY2.

Arithmetic Operations with Arrays

A general rule about arithmetic operations with arrays is that the number of occurrences of the corresponding dimensions must be equal.

The following illustrates this rule:

```
#c(2:3,2:4) := #a(3:4,1:3) + #b(3:5)
```

In other words:

Array	Dimension Number	Number of Occurrences	Range
#c	2nd	2	2:3
#c	1st	3	2:4
#a	2nd	2	3:4
#a	1st	3	1:3
#b	1st	3	3:5

The operation is performed element by element.

Note:
An arithmetic operation of a different number of dimensions is allowed.

For the example above, the following operations are executed:

```
#c(2,2) := #a(3,1) + #b(3)

#c(2,3) := #a(3,2) + #b(4)

#c(2,4) := #a(3,3) + #b(5)

#c(3,2) := #a(4,1) + #b(3)

#c(3,3) := #a(4,2) + #b(4)

#c(3,4) := #a(4,3) + #b(5)
```

Below is a list of examples of how array ranges may be used in the following ways in arithmetic operations (in `COMPUTE`, `ADD` or `MULTIPLY` statements). In examples 1-4, the number of occurrences of the corresponding dimensions must be equal.

1. *range* + *range* = *range*.

   The addition is performed element by element.

2. *range* * *range* = *range*.

   The multiplication is performed element by element.

3. *scalar* + *range* = *range*.

   The scalar is added to each element of the range.

4. *range* * *scalar* = *range*.

   Each element of the range is multiplied by the scalar.

5. *range* + *scalar* = *scalar*.

   Each element of the range is added to the scalar and the result is assigned to the scalar.

6. *scalar* * *range* = *scalar2*.

   The scalar is multiplied by each element of the array and the result is assigned to *scalar2*.

Since intermediate results will be generated for arithmetic operations as shown in the above examples, the result of overlapping index ranges is computed element by element in an intermediate result array and finally the intermediate result array is assigned to the result field.

Example:

```
DEFINE DATA LOCAL
1 #ARRAY(I4/1:5) INIT <10,20,30,40,50>
END-DEFINE

#ARRAY(3:5) := #ARRAY(2:4) + 1

/* A temporary array for the
/* intermediate result values is
/* generated implicitly: #temp(1:3).
/* The following operations are
/* performed internally:
/* #temp(1) := #ARRAY(2) + 1
```

```
/* #temp(2) := #ARRAY(3) + 1
/* #temp(3) := #ARRAY(4) + 1
/* #ARRAY(3:5) := #temp(1:3)
/*
/* #ARRAY contains  10,20,21,31,41
```