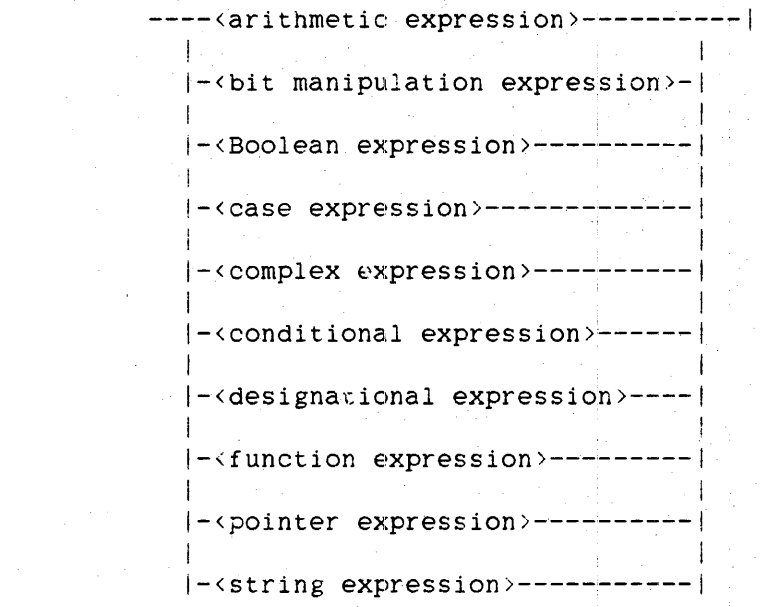
6 EXPRESSIONS

6.1 EXPRESSIONS: CONCEPTS AND TYPES

An expression describes how a value can be obtained by applying specified operations to designated operands or primaries.

Syntax

<expression>



Semantics

Evaluation of an arithmetic expression returns a numerical value. Evaluation of a Boolean expression returns a Boolean value (TRUE or FALSE). Evaluation of a complex expression returns a complex value--a value with a real numerical part and an imaginary numerical part. Evaluation of a designational expression returns a label. Evaluation of a pointer expression returns a value that can be used to reference a character position in an array row. Evaluation of a string expression returns a value that is an EBCDIC, ASCII, or hexadecimal string.

A bit manipulation expression operates on bits within words and makes it possible to "build" the contents of a word from groups of bits contained in other words.

Case expressions and conditional expressions allow one of several alternative expressions to be chosen for evaluation based on a selection value.

A function expression is a call on a typed procedure. The procedure can be declared in the program or it can be an intrinsic--a typed procedure that is a predefined part of the ALGOL language. Intrinsic functions exist that are predefined arithmetic expressions, predefined Boolean expressions, predefined complex expressions, predefined pointer expressions, and predefined string expressions. For example, SQRT(<arithmetic expression>) is a function expression that returns the square root of the value of <arithmetic expression>. Because SQRT returns a numeric value, it is an arithmetic function--a predefined arithmetic expression.

NOTE

Expressions that are very large or deeply nested can cause the compiler to get a stack overflow fault. The fault can be avoided by breaking very large or deeply nested expressions into several separate expressions or by increasing the maximum stack size by using the task attribute STACKLIMIT.

Examples

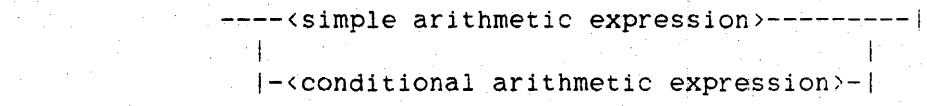
```
X*Y                 A = B               CASE N OF (1,2,4,8)

SWLBL[SWX]          SQRT(X)             IF BOOL THEN A ELSE B

POINTER(A)          S1 || S2            0 & N [3:4]

RSLT.[19:20]        CSIN(C)
```

ARITHMETIC EXPRESSION

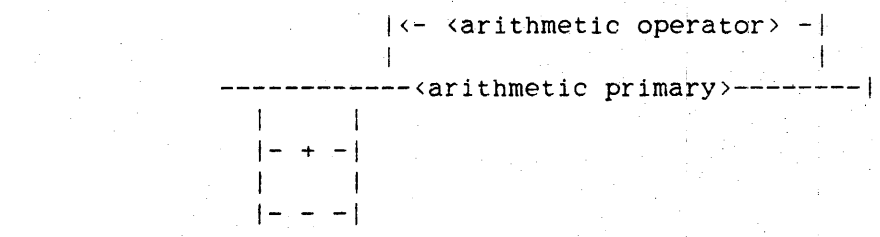
Arithmetic expressions are expressions that return numerical values by performing specified operations on designated arithmetic primaries.

Syntax

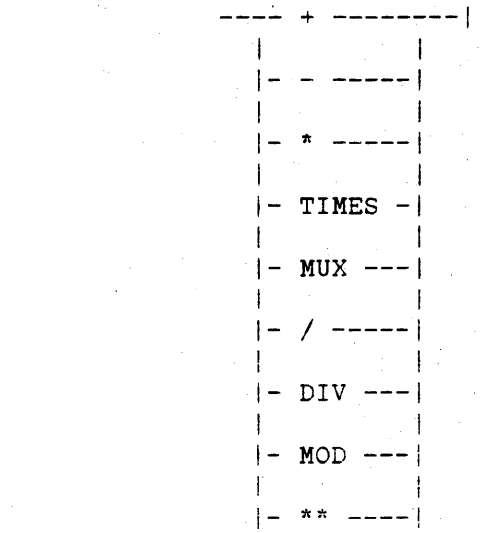
<arithmetic expression>



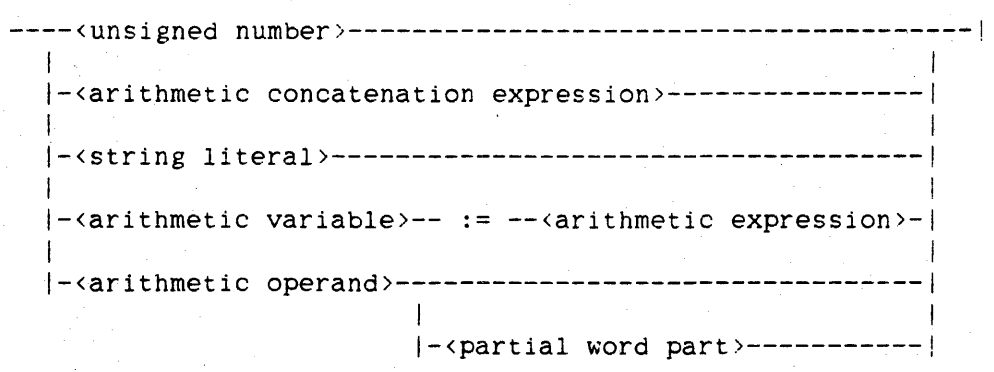
<simple arithmetic expression>



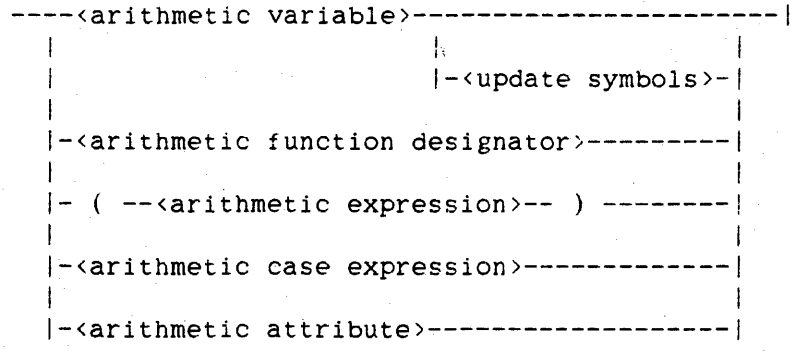
<arithmetic operator>



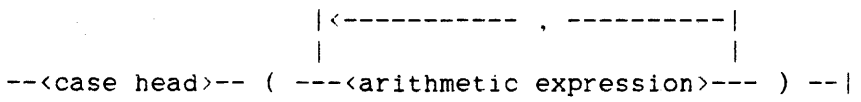
<arithmetic primary>



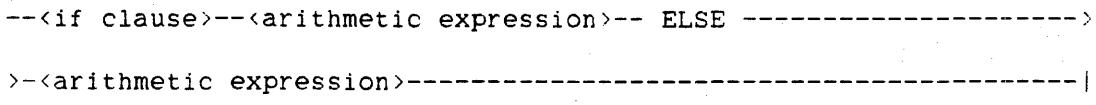
<arithmetic operand>



<arithmetic case expression>



<conditional arithmetic expression>



<constant arithmetic expression>

An <arithmetic expression> that can be fully evaluated at compile time.

See also

Examples

Valid	Invalid
SUM/N	L*-A
(A+B)/(C-D)	*ENTIER(60)
2*(X+Y)	4(AC)
(A+B)/(C-D)	X*-3
COS(A+B)+C	3X + 4Y + Z
Y*3	A(X + 5)
+8	A+X*(B:=X*(C+X(D+X*E))))
(-B+SQRT(D))/(A+A)	P*[X + Y + Z]
-T*3	X + Y*-X + Z**2
A+X*(B:=X*(C+X*(D+X*E)))	
THETA	

Semantics

The evaluation of a conditional arithmetic expression is described in "Conditional Expression."

Arithmetic Primaries

The items on which arithmetic operators act are called arithmetic primaries.

A variable or function designator used as an arithmetic primary in an arithmetic expression must be of an arithmetic type--INTEGER. REAL, or DOUBLE.

An attribute used as an arithmetic primary must have a type that is INTEGER, REAL, or DOUBLE. For information on file attribute and direct array attribute types, see the "I/O Subsystem Reference Manual." The arithmetic-valued task attributes are described under "Arithmetic Assignment."

The length of a string literal used as an arithmetic primary must not exceed 48 bits. A string literal used as an arithmetic primary is interpreted as either type REAL or type INTEGER, depending on its value.

The arithmetic concatenation expression is described in "Concatenation Expression."

The partial word part is described in "Partial Word Expression."

The evaluation of an arithmetic case expression is described in "Case Expression."

See also
Arithmetic Assignment 225

Examples

Valid	Invalid
5.678	X := * := Y
X := * + 3	+ DC8
(14 + 3.142)	B - A
MABEL	-(A + B)
R & 3 [1:2]	TRUE
Y.[30:4]	
"ABCD"	
SQRT(X)	
CASE I OF (5,15,17)	
FYLE.MAXRECSIZE	

Arithmetic Operators

The operators +, -, *, and / have the conventional mathematical meanings of addition, subtraction, multiplication, and division, respectively. No two operators can be adjacent, and implied multiplication is not allowed.

The TIMES and * operators denote multiplication.

The DIV operator denotes integer division. It has the following mathematical meaning:

```
Y DIV Z = SIGN(Y/Z) * ENTIER(ABS(Y/Z))
```

The MOD operator denotes remainder division. For Z greater than or equal to 1, MOD has the following meaning:

```
Y MOD Z = Y - (Z * (Y DIV Z))
```

For Z less than 1, the MOD operator produces undefined results.

The MUX operator multiplies either single- or double-precision arithmetic primaries, and yields a double-precision result.

The ** operator denotes exponentiation. The semantics of the exponentiation operator depend on the types and values of the primaries involved. Table 6-1 explains the various meanings of Y**Z.

Table 6-1. Exponentiation
Meaning of Y**Z

	Z: Type Integer			Z: Type Real		
	Z > 0	Z = 0	Z < 0	Z > 0	Z = 0	Z < 0
Y>0	Note 1	1	Note 2	Note 3	1	Note 3
Y<0	Note 1	1	Note 2	Note 4	1	Note 4
Y=0	0	Note 4	Note 4	0	Note 4	Note 4

Note 1: Y**Z = Y*Y*Y ... *Y (Z times)
Note 2: Y**Z = Reciprocal of Y*Y*Y ...*Y (ABS(Z) times)
Note 3: Y**Z = EXP(Z*LN(Y))
Note 4: Value of the expression is undefined.

Precedence of Arithmetic Operators

The sequence in which the operations of an arithmetic expression are performed is determined by the precedence of the operators involved. The order of precedence is as follows:

1. ** (highest precedence)

2. *, /, MOD, DIV, MUX, TIMES

3. +, -

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. An expression to the right of an assignment operator is evaluated before the assignment.

2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

Parentheses can be used in normal mathematical fashion to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. For example, in the expression

```
(X+1)/Y
```

the addition is performed before the division because of the parentheses. In the expression

```
X+1/Y
```

1 is first divided by Y and then the result is added to X.

Table 6-2 illustrates how mathematical notation can be translated to an ALGOL arithmetic expression.

Table 6-2. Mathematical Notation

Mathematical Expression	Equivalent ALGOL Expression
A X B	A * B
$A + \frac{B}{2}$	A + B/2
$\frac{X + 1}{Y}$	(X + 1)/Y
$\frac{D + E^2}{2A}$	(D + E**2)/(2 * A)
$4(X + Y)^3$	4 * (X + Y) ** 3
$\frac{M - N}{(M + N)^{P + 5 X 10^{-6}}}$	(M - N)/(M + N) ** (P + 5@-6)

Precision of Arithmetic Expressions

The value of an arithmetic expression can be expressed in single or double precision, depending on the precision of its constituents or, in the case of MUX, on the operator involved. The value of an arithmetic expression is double precision if any variable, function, or number of which it is composed is of type DOUBLE, or if two primaries are combined by the double-precision operator MUX. The MUX operator allows a double-precision result to be obtained from the multiplication of two single-precision arithmetic primaries.

The value of a case expression is double precision if any expression in its arithmetic expression list is of type DOUBLE. Likewise, the value of a conditional arithmetic expression is double precision if either arithmetic expression is double precision. In either case, single-precision arithmetic expressions are converted to double precision, when necessary.

Types of Resulting Values

The type of the value resulting from an arithmetic operation depends on the arithmetic operator and the types of the primaries being combined, except when the resulting value is undefined. Table 6-3 describes the types of quantities that result from various combinations of arithmetic primaries.

Table 6-3. Types of Values Resulting from Arithmetic Operations

Operand on Left	Operand on Right	+ - *	/	DIV	MOD	**	MUX
INTEGER	INTEGER	Note 3	REAL	INTEGER	INTEGER	Note 1	DOUBLE
INTEGER	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
INTEGER	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	Note 2	DOUBLE
REAL	INTEGER	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	REAL	REAL	REAL	INTEGER	REAL	Note 2	DOUBLE
REAL	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE
DOUBLE	(any)	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE	DOUBLE

Note 1: If the operand on the right is negative or the absolute value of the result is greater than or equal to 2**39, REAL; otherwise, INTEGER

Note 2: If the operand on the right is zero, INTEGER; otherwise, REAL

Note 3: If the absolute value of the result is less than 2**39, INTEGER; otherwise, REAL

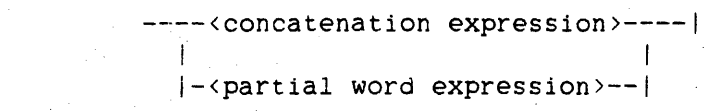
The type of an arithmetic case expression or a conditional arithmetic expression is DOUBLE if any of its constituent expressions are of type DOUBLE. (For more information, refer to "Precision of Arithmetic Expressions" above.) If the conditional arithmetic expression or arithmetic case expression contains only expressions of type INTEGER and REAL, its type is REAL. A conditional arithmetic expression or arithmetic case expression is of type INTEGER only if all its constituent expressions are of type INTEGER.

BIT MANIPULATION EXPRESSION

Bit manipulation expressions provide a means of isolating a field of one or more bits from a word, and allow words to be constructed from fields of one or more bits from other words.

Syntax

<bit manipulation expression>

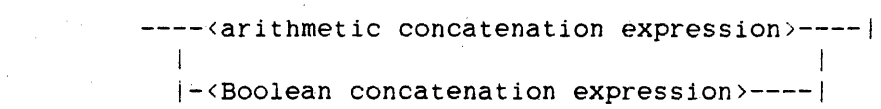


Concatenation Expression

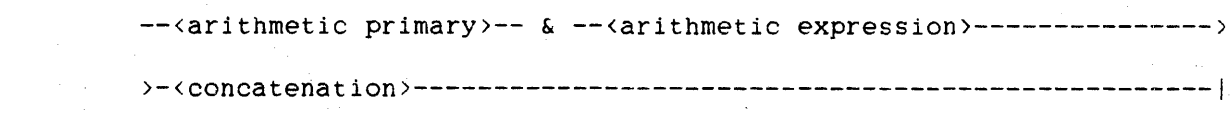
The concatenation expression forms a primary from selected parts of two or more primaries.

Syntax

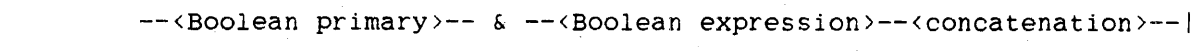
<concatenation expression>

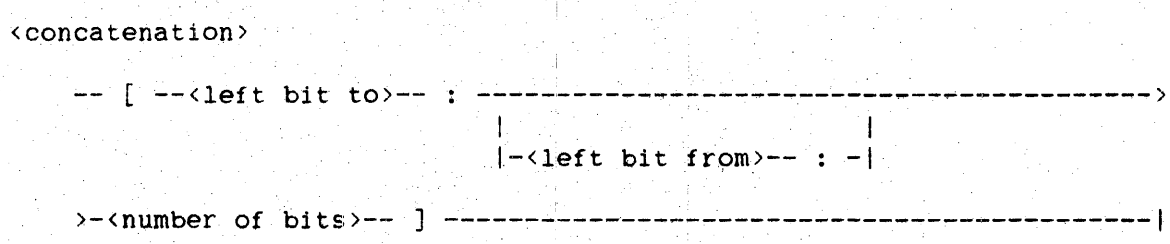


<arithmetic concatenation expression>



<Boolean concatenation expression>





See also
<arithmetic primary> 476
<Boolean primary> 492

Semantics

A concatenation primary is formed by taking a specified part of the bit pattern of the value of an expression and copying it into the specified portion of a primary. The rest of the destination primary is not changed by this operation.

Note that only arithmetic expressions can be concatenated with arithmetic primaries, and only Boolean expressions can be concatenated with Boolean primaries.

Because the concatenation expression is a primary, and the syntax for a concatenation expression is of the form

```
<primary> & <expression> <concatenation>
```

concatenation expressions of the following form are allowed:

```
<primary> & <expression> <concatenation>
          & <expression> <concatenation>
                         .
                         .
                         .
          & <expression> <concatenation>
```

If, as in the example above, more than one concatenation term is used in a concatenation expression, then these terms are evaluated from left to right.

<concatenation>

The <concatenation> construct describes the location in the expression of the field to be copied and the location in the destination primary where the field is to be copied.

The <left bit to> element defines the leftmost bit location of the field in the destination word. The <left bit from> element defines the leftmost bit location of the field in the source word. The <number of bits> element specifies the length of the field to be copied from the source to the destination.

If the "[<left bit to>:<left bit from>:<number of bits>]" form is used, the field of bits to be copied starts at <left bit from> in the source word and is <number of bits> long.

If the "[<left bit to>:<number of bits>]" form is used, then the field of bits to be copied from the source word starts at bit number (<number of bits>-1) and extends through bit zero. That is, the source field is assumed to be the low-order <number of bits> bits in the source word.

The values of <left bit to> and <left bit from> must lie within the range 0 through 47, where bit 0 is the rightmost, or least significant, bit in the word.

The value of <number of bits> must lie within the range 0 through 48. If the value of <number of bits> exceeds the number of bits to the right of the starting bit in either the source or destination words, these fields wrap around and are continued at bit 47, the leftmost bit, of the same word.

If, through the programmer's use of variables, the ranges for <left bit to>, <left bit from>, or <number of bits> are exceeded, then the program is discontinued with a fault.

Because a concatenation expression is a primary, when it appears as an operand in a larger expression, the concatenation expression is evaluated before other any operation is executed. For example, the expression

```
2**4 & 1 [0:0:1]
```

is evaluated as

```
2**(4 & 1 [0:0:1]) = 2**5 = 32
```

and NOT as

```
(2**4) & 1 [0:0:1] = 16 & 1 [0:0:1] = 17
```

For the same reason, the Boolean concatenation expression

```
3 < 7 & (5 > 8) [47:47:20]
```

is evaluated as

```
3 < (7 & (5 > 8) [47:47:20])
```

which is equivalent to

```
3 < (7 & FALSE [47:47:20])
```

This results in an error. because the Boolean expression "FALSE" cannot be concatenated with the arithmetic primary "7".

Examples

Given real variables X, Y, and Z with the following values:

```
X = 32767 = 4"000000007FFF"
Y =  1024 = 4"000000000400"
Z =     1 = 4"000000000001"
```

the following are examples of arithmetic concatenation expressions and their values:

Expression	Value
X & Y [47:11:4]	4"400000007FFF"
X & Y [47:12]	4"400000007FFF"
Y & X [39:20]	4"0007FFF00400"
Y & Z [46:1]	4"400000000400"
0 & Y [11:11]	4"000000000800"
0 & X [23:48]	4"007FFF000000"
X & Y [39:12] & Z [47:1]	4"804000007FFF"
Y & X [19:15:8]	4"00000007F400"

Assume the elements of the real array INFO contain information about the data in a file. Each element of INFO contains a record number in the field [19:20], and the length of the data in that record is in field [39:20]. That is, each element of INFO stores the location and length of a record in the data file. Let N be a variable that contains a record number and let L be a variable that contains the length of that record. The following declarations and assignment statement could be used to store a value into an element of INFO.

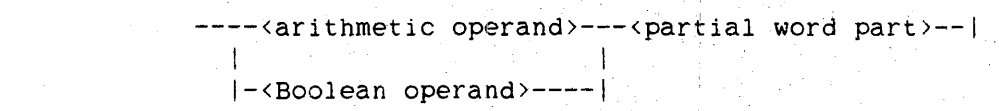
```
DEFINE
   REC_NUMF = [19:20]#,
   LENGTHF  = [39:20]#;
          .
          .
          .
INFO[I] := 0 & N REC_NUMF
             & L LENGTHF;
```

Partial Word Expression

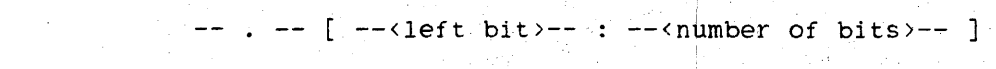
A partial word expression isolates the value of a field of one or more bits of a specified word.

Syntax

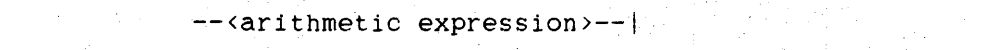
<partial word expression>



<partial word part>



<left bit>



See also
<arithmetic operand>. 476
<Boolean operand> 492
<number of bits>. 485

Semantics

The <partial word part> construct describes the location in the operand of the field to be isolated. The isolated field is copied to the low order (rightmost) field of a word of all zeros.

The <left bit> element defines the leftmost bit location of the field in the source word. The value of <left bit> must lie within the range 0 through 47, where bit 0 is the rightmost, or least significant, bit in the word.

The <number of bits> element specifies the length of the field. The value of <number of bits> must lie within the range 0 through 48. If the value of <number of bits> exceeds the number of bits to the right of <left bit> in the source word, the field wraps around and is continued at bit 47, the leftmost bit, of the same word.

If, through the programmer's use of variables, these ranges are exceeded, the program is discontinued with a fault.

Examples

Given real variables X, Y, and Z with the following values:

```
X = 32767 = 4"000000007FFF"
Y =  1024 = 4"000000000400"
Z =     2 = 4"000000000002"
```

the following are examples of arithmetic partial word expressions and their values:

Expression	Value
X.[5:6]	4"00000000003F"
Y.[11:4]	4"000000000004"
Z.[19:48]	4"000040000000"
X.[23:24]	4"000000007FFF"
X.[23:20]	4"0000000007FF"

Using the INFO array example from "Concatenation Expression" above, the following assignments could be used to extract information from INFO.

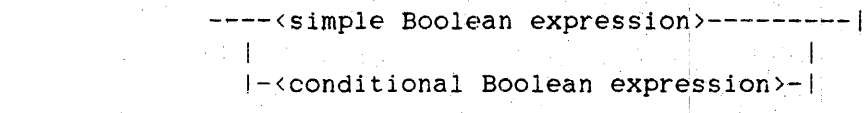
```
N := INFO[I].REC_NUMF;
L := INFO[I].LENGTHF:
```

BOOLEAN EXPRESSION

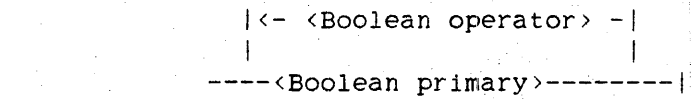
Boolean expressions are expressions that return logical values by applying specified operations to designated Boolean primaries.

Syntax

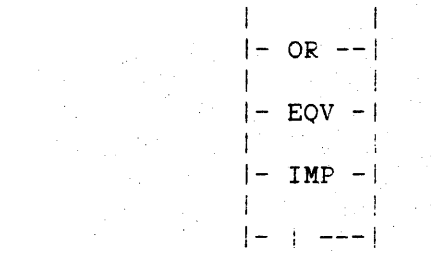
<Boolean expression>



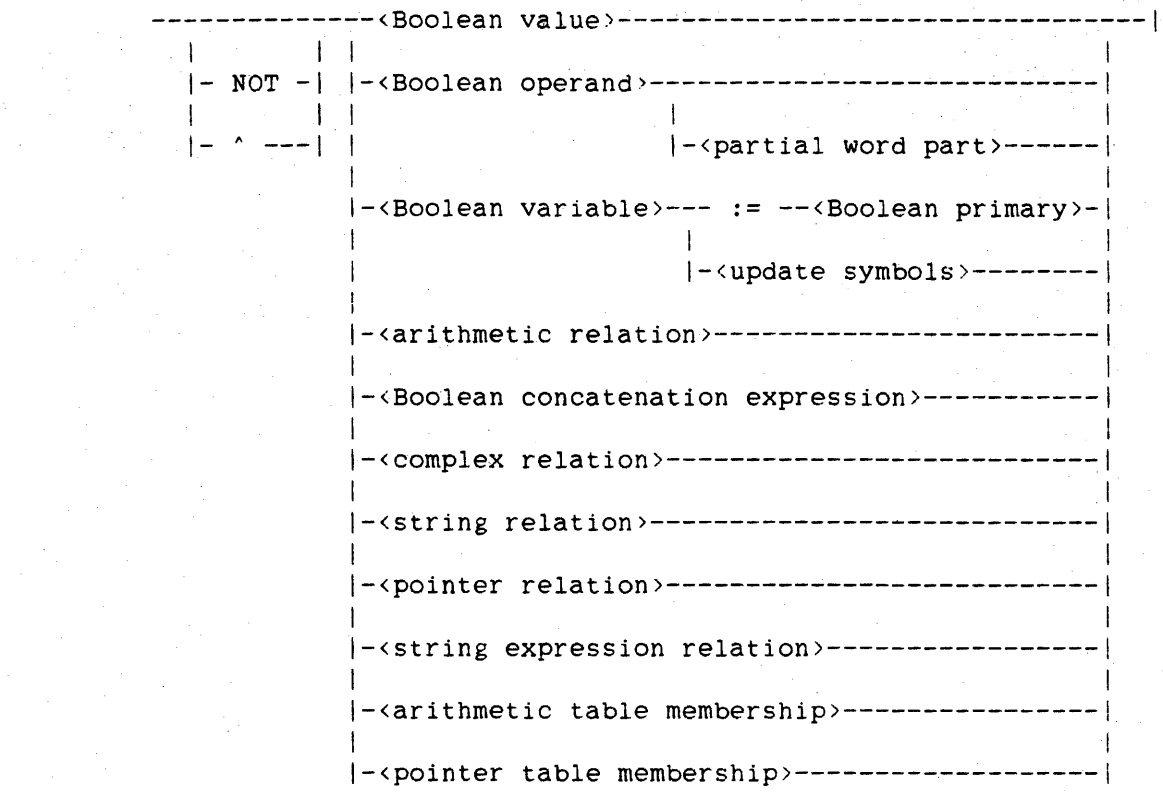
<simple Boolean expression>



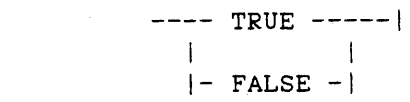
<Boolean operator>



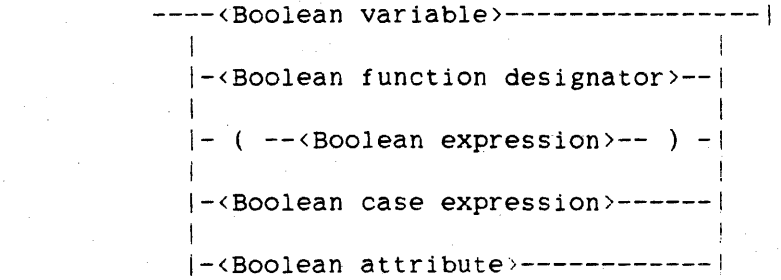
<Boolean primary>



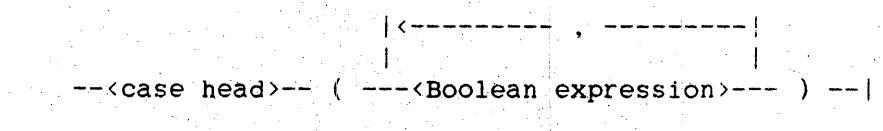
<Boolean value>



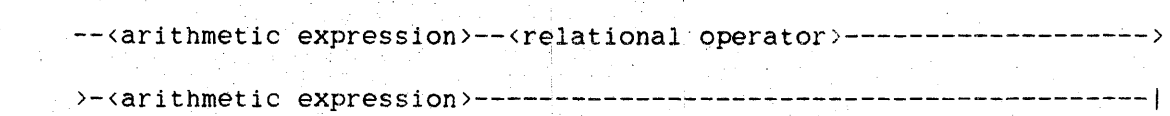
<Boolean operand>



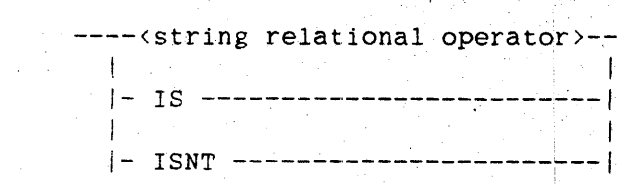
<Boolean case expression>



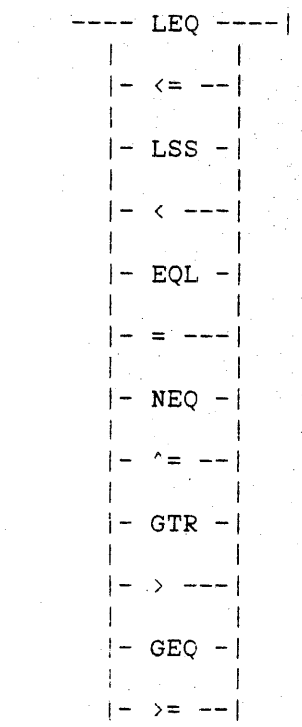
<arithmetic relation>

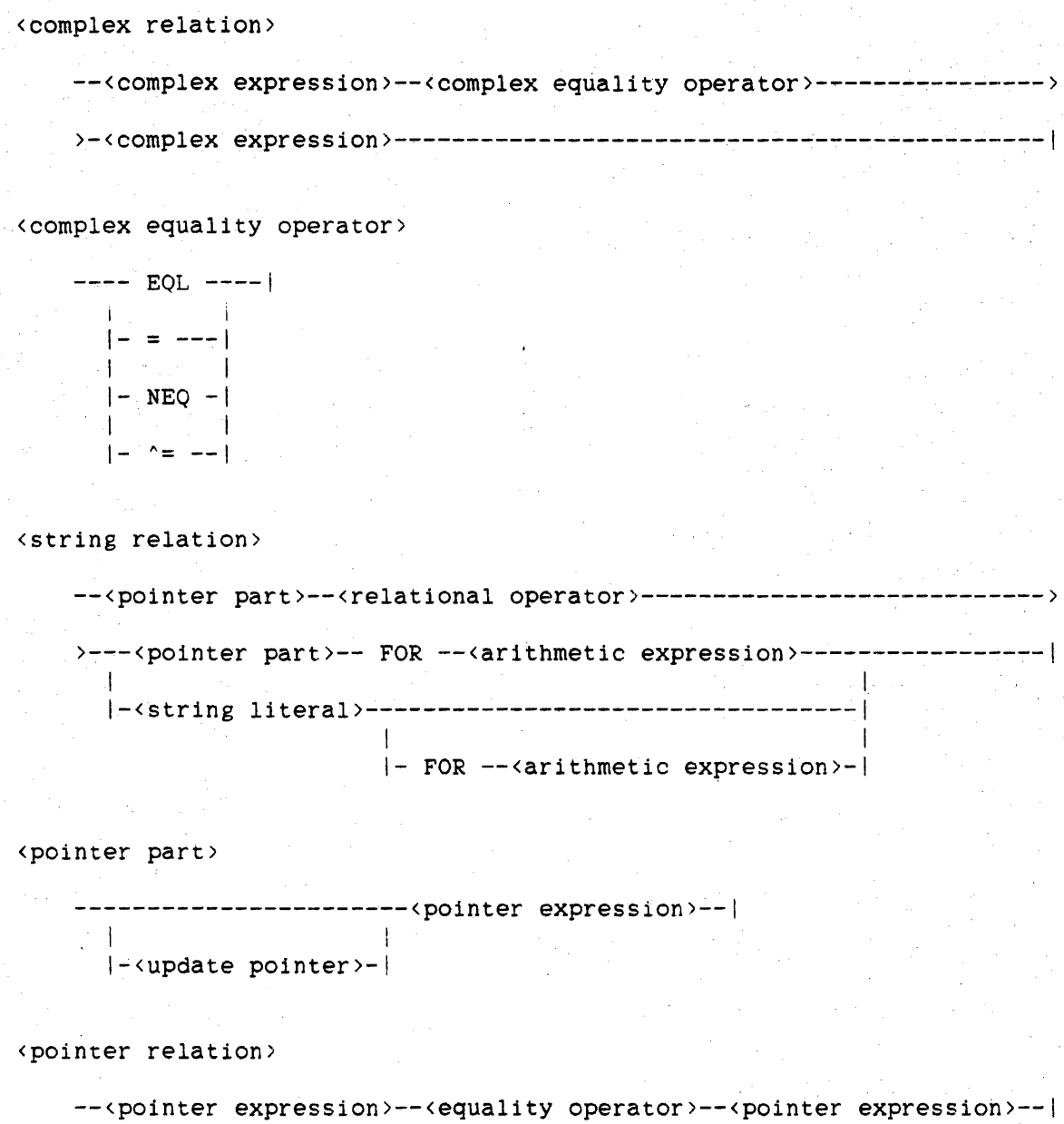


<relational operator>

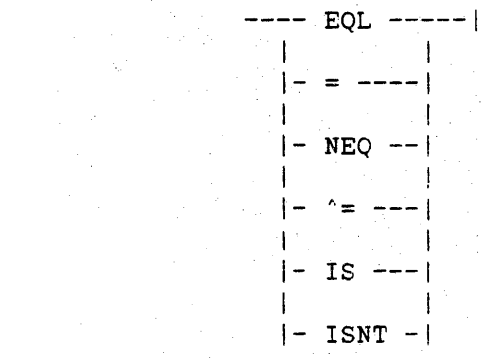


<string relational operator>

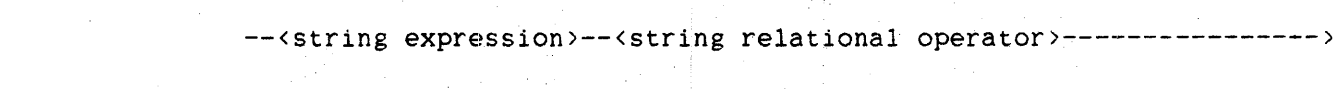



<complex relation>
--<complex expression>--<complex equality operator>----------->
>-<complex expression>---|
<complex equality operator>
---- EQL ----|
|- = ---|
|- NEQ -|
|- ^= --|
<string relation>
--<pointer part>--<relational operator>--------------------------->
>---<pointer part>-- FOR --<arithmetic expression>----------------|
|-<string literal>-----------------------------------|
|- FOR --<arithmetic expression>-|
<pointer part>
----------<pointer expression>--|
|-<update pointer>-|
<pointer relation>
--<pointer expression>--<equality operator>--<pointer expression>--|

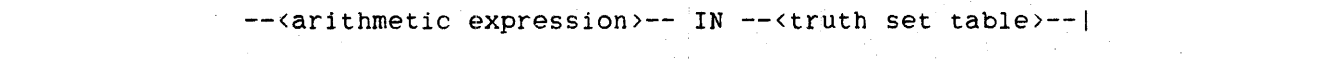
<equality operator>



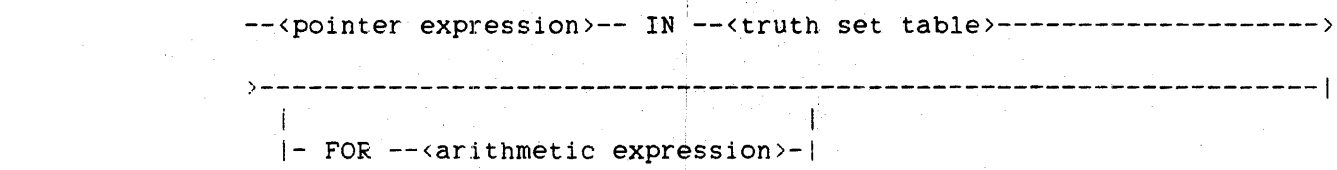
<string expression relation>



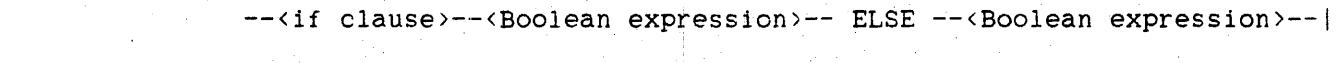
<arithmetic table membership>



<pointer table membership>



<conditional Boolean expression>



See also

Examples

Valid	Invalid
B AND TRUE	1+A AND Z>0
1-A > B*(-E)	NOT NOT A
(X=Y OR W=K)	1-W*2
X=0 AND Y^=0	(B*2-4XAXC)
A>1 AND (B=0 OR C<D)	
(A=B OR C=D) AND (X<2 OR Y<2)	
A=B	
X EQV Y	
NOT B & TRUE [5:1]	

Semantics

The evaluation of a conditional Boolean expression is described in "Conditional Expression."

Operators in Boolean Expressions

The following table lists the operators that can be used in Boolean expressions, along with their meanings. When two operators are listed on the same line, they are equivalent to each other.

Operator		Meaning
NOT	^	logical NOT
AND		logical AND
OR	|	logical inclusive OR
IMP		logical implication
EQV		logical equivalence
IS		identical to
ISNT		not identical to
EQL	=	equal to
NEQ	^=	not equal to
GTR	>	greater than
GEQ	>=	greater than or equal to
LSS	<	less than
LEQ	<=	less than or equal to

Logical Operators

The values returned by the logical operators are defined in Table 6-4.

Table 6-4. Results of Logical Operators

Operand A	Operand B	NOT A	A AND B	A OR B	A IMP B	A EQV B
TRUE	TRUE	FALSE	TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	FALSE	TRUE	FALSE	FALSE
FALSE	TRUE	TRUE	FALSE	TRUE	TRUE	FALSE
FALSE	FALSE	TRUE	FALSE	FALSE	TRUE	TRUE

The Boolean operations defined above are performed on all 48 bits of the Boolean primaries on a bit-by-bit basis. For example, the constant TRUE (4"000000000001") does not have the same bit pattern as the Boolean expression NOT FALSE, because NOT complements all 48 bits of the constant FALSE (4"000000000000"), generating 4"FFFFFFFFFFFF". Even so, the constant TRUE and the Boolean expression NOT FALSE have the same Boolean value of TRUE because the Boolean value of a Boolean primary is based upon the value of the low-order bit (bit zero) of the Boolean primary (0 is FALSE; 1 is TRUE).

NOTE

Exception: When NOT operates on an arithmetic relation, the low-order bit (bit zero) is complemented; however, the other 47 bits are not necessarily complemented. For example, if X = Y evaluates to TRUE, then NOT(X = Y) evaluates to FALSE, not necessarily to NOT TRUE as would be expected.

IS and ISNT Operators

The IS relational operator performs a bit-for-bit compare on its two operands. It returns TRUE if the corresponding bits of each operand are the same. The ISNT operator is the negation of the IS operator.

If the IS operator is used to compare a double-precision arithmetic quantity to a single-precision (integer or real) arithmetic quantity, the result is always FALSE.

The IS operator differs from the EQL or = operator, which does an arithmetic compare on its operands. Two operands can have the same arithmetic value with different bit patterns. Thus, the following pairs yield TRUE when compared with the EQL or = operator, but yield FALSE when compared with the IS operator:

1. +0 and -0
2. A normalized number and the same number not in normalized format
3. A number with an exponent of +0 and the same number with an exponent of -0
4. A number with bit 47 = 0 and the same number with bit 47 = 1

Relational Operators

The action of the relational operators GTR, >, GEQ, >=, LSS, <, LEQ, <=, EQL, =, NEQ, and ^= depends on the kinds of items being compared. For more information, refer to "<arithmetic relation>," "<complex relation>," "<string relation>," "<pointer relation>," and "<string expression relation>" in this section.

Precedence in Boolean Expressions

The components of Boolean expressions are evaluated in the following order:

1. All arithmetic, complex, pointer, and string expressions are evaluated.
2. All relations, table memberships, and assignments are evaluated.
3. Logical operators are then applied.

The order of precedence of the logical operators is as follows:

1. NOT (highest precedence)
2. AND
3. OR, |
4. IMP
5. EQV

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. A primary to the right of an assignment operator is evaluated before the assignment.
2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

Parentheses can be used to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. For example, in the expression

```
X AND (Y OR Z)
```

the OR is performed before the AND because of the parentheses. In the expression

```
X AND Y OR Z
```

the AND is performed before the OR.

Boolean Primaries

The Boolean concatenation expression is described in "Concatenation Expression."

The partial word part is described in "Partial Word Expression."

The evaluation of a Boolean case expression is described in "Case Expression."

<Boolean value>

The Boolean value TRUE is represented internally as a 48-bit word containing 4"000000000001", and the Boolean value FALSE is represented internally as a 48-bit word containing 4"000000000000".

<arithmetic relation>

An arithmetic relation performs an arithmetic comparison of the values of two arithmetic expressions. The value of the relation is either TRUE or FALSE.

<complex relation>

A complex relation performs a comparison between the values of two complex expressions. Because one of the forms of a complex expression is an arithmetic expression, the complex relation can also be used to compare a complex value and an arithmetic value. In a complex relation, the only allowed relational operators are =, EQL, ^=, and NEQ.

<string relation>

The string relation syntax causes a comparison to be performed between two character strings referenced by two pointer expressions, or between the character string referenced by a pointer expression and a string literal. The character strings are compared according to the EBCDIC collating sequence. The arithmetic expression specifies the number of characters to be compared (the repeat count). If a string literal follows the relational operator and a repeat count has been specified, then the string literal is concatenated with itself, if necessary, to form a 48-bit literal. The comparison is repeated until the repeat count is exhausted. If no repeat count is specified, the string characters are compared once.

<pointer relation>

A pointer relation determines whether two pointer expressions refer to the same character position in the same array row. If the character sizes of the two pointer expressions are unequal, the comparison always yields the value FALSE.

A pointer relation should not be confused with a string relation. A string relation compares the character strings referenced by pointer expressions. A pointer relation compares the pointer expressions themselves. For example, if pointers P1 and P2 are initialized as follows:

```
POINTER P1,P2;
REAL ARRAY A,B[0:1];
P1 := POINTER(A,8);
P2 := POINTER(B,8);
REPLACE P1 BY "A";
REPLACE P2 BY "A";
```

then the string relation

```
P1 EQL P2 FOR 1
```

would have the value TRUE, but the pointer relation

```
P1 EQL P2
```

would have the value FALSE, because P1 refers to array A and P2 refers to array B.

<string expression relation>

The string expression relation compares two string expressions according to the collating sequence of the character type of the string expressions. Only string expressions of the same character type can be compared.

Two strings are equal only if the lengths of the two strings are equal and if every character in one string is equal to the corresponding character in the other string. Two null strings are equal.

One string is strictly greater than a second string only if at least one of the following conditions is true:

1. The leftmost character in the first string that is not equal to the corresponding character in the second string compares as greater than the corresponding character in the second string.

2. The length of the first string is greater than the length of the second string, and the two strings compare as equal for the length of the second string.

Whenever two string literals can be compared as two arithmetic primaries, they are so compared. For example, the Boolean expression

```
"B" > "AA"
```

yields a value of FALSE, because the two quoted strings are treated as two arithmetic primaries. However, if the two string literals "B" and "AA" were assigned to two string variables T and S, respectively, then the Boolean expression

```
T > S
```

yields a value of TRUE, because the first character of T is greater than the first character of S.

For example, if a string S1 is assigned the value "AAB12+", then all of the following comparisons are TRUE:

```
S1 EQL TAKE(S1,6)
S1 NEQ HEAD(S1,ALPHA)
S1 GTR HEAD(S1,ALPHA)
S1 LSS DROP(S1,3)
S1 LSS DROP(S1,1)
```

Table Membership

The table membership constructs allow testing to determine whether a character is a member of a truth set. The character can be either a character in a string literal or a character in an array row referenced by a pointer expression. In a pointer table membership primary, the "FOR ‹arithmetic expression›" part applies the membership test to the first ‹arithmetic expression› characters to which the pointer expression points.

The ‹subscripted variable› form of the ‹truth set table› construct allows several truth set tables to be contained in one array row. The value of the subscript indicates the beginning of the desired table

within the array row. For a description of truth sets, refer to "TRUTHSET Declaration."

Examples

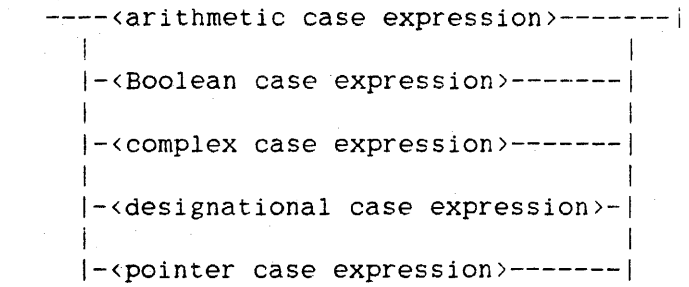
Valid	Invalid
TRUE	SQRT(X)
BOOL	5.67
B.[11:1]	-X
B := TRUE	R + TRUE
B1 := * AND B2	CSIN(C1) > CSIN(C2)
CSIN(C1) = CSIN(C2)	P1 > P2
X > Y	S1 IS "ABCD"
NOT B	
R = 3	
PTR = "ABCD"	
P1 < P2 FOR 20	
S1 || S2 = S3	
HAPPENED(E)	
CASE I OF (B1, TRUE, X=Y)	
NOT FYLE.OPEN	
P1 = P2	
BOOL & TRUE [19:1]	
X IN ALPHA8	
P1 IN ALPHA8 FOR 6	

CASE EXPRESSION

Case expressions provide a means of selecting from a list of expressions one expression for evaluation.

Syntax

<case expression>



See also
<arithmetic case expression> 476
<Boolean case expression> 493
<complex case expression> 507
<designational case expression> 512
<pointer case expression> 519

Semantics

In a case expression, the list of expressions must be composed of expressions of the same kind; for example, all arithmetic expressions or all Boolean expressions. The expression to be evaluated is selected as follows:

1. The arithmetic expression in the case head is evaluated and integerized by rounding, if necessary.

2. This value is used as an index into the expression list. The component expressions of the expression list are numbered sequentially from 0 through N-1, where N is the number of expressions in the list.

3. The expression selected by the index is evaluated, and its value is the value of the case expression.

If the value of the index lies outside the range 0 through N-1, the program is discontinued with a fault.

The type of an arithmetic case expression is DOUBLE if any of its constituent expressions is of type DOUBLE; in this case, any constituent expression that is not of type DOUBLE is extended to double precision. The type of an arithmetic case expression is INTEGER if and only if all of its constituent expressions are of type INTEGER. Otherwise, an arithmetic case expression is of type REAL.

Examples

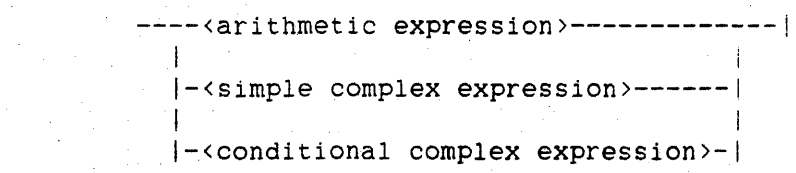
```
CASE N OF (2, 20, 100, 37)

CASE X.[27:2] OF (TRUE, FALSE, TRUE, TRUE)

CASE I OF (C1, C2, COMPLEX(X,Y))

CASE TSTS[INDEX] OF (LBL1, LBL2, AGAIN, NEXT, MORE)

CASE CHAR.SZF OF (PTR, PTS, POINTER(A), PTEMP, POLO)
```

COMPLEX EXPRESSION

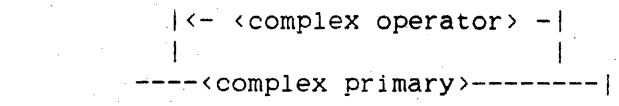
Complex expressions are expressions that return complex values (arithmetic values that consist of a real part and an imaginary part) by applying specified operations to designated complex primaries.

Syntax

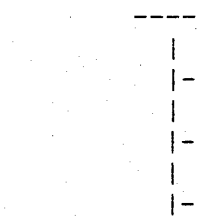
<complex expression>



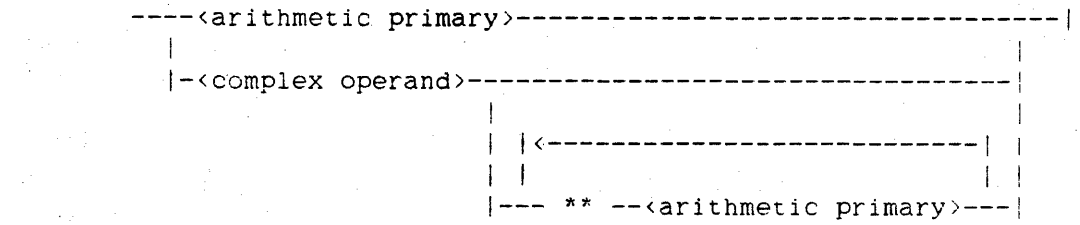
<simple complex expression>



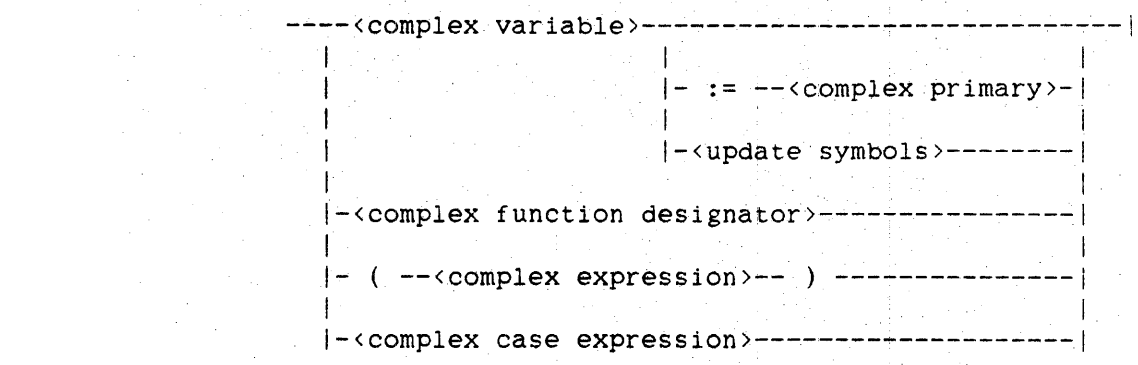
<complex operator>



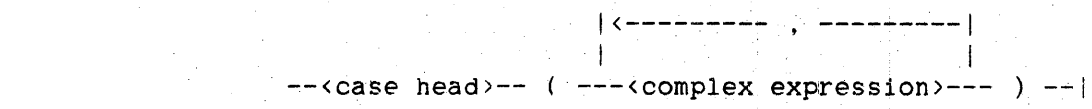
<complex primary>



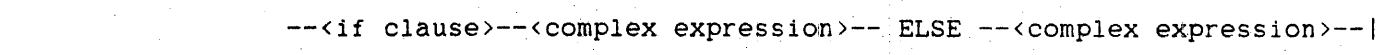
<complex operand>



<complex case expression>



<conditional complex expression>



See also

Semantics

The imaginary part of a complex value can be equal to zero. Because of this, an arithmetic expression is, whenever necessary, considered to be the real part of a complex expression with a zero imaginary part. No automatic type conversion from complex to arithmetic exists.

The sequence in which the operations of a complex expression are performed is determined by the precedence of the operators involved. The order of precedence is as follows:

1. *, / (highest precedence)

2. +, -

Operators with the same precedence are applied in their order of appearance in an expression, from left to right.

The precedence of the assignment operator (:=) is as follows:

1. A primary to the right of an assignment operator is evaluated before the assignment.

2. The assignment is done before the evaluation of an expression involving the variable that is the target of the assignment.

Parentheses can be used in normal mathematical fashion to override the defined order of precedence. An expression in parentheses is evaluated by itself, and the resulting value is subsequently combined with the other elements of the expression. For example, in the expression

```
(C2 + C1)/C2
```

the addition is performed before the division because of the parentheses. In the expression

```
C2 + C1/C2
```

C1 is first divided by C2 and then the result is added to C2.

The evaluation of a conditional complex expression is described in "Conditional Expression."

The evaluation of a complex case expression is described in "Case Expression."

Examples

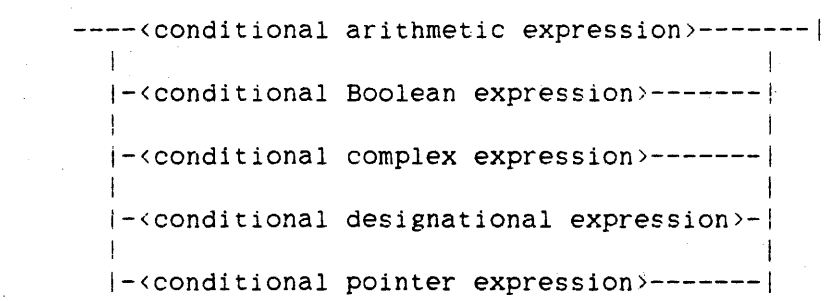
```
C1

C1+3

C1 ** X

C1 := COMPLEX(X,Y)

C1 := * - C2
```

```
CABS(C1)

(COMPLEX(R,S) * CCON)

CASE I OF (C1, C2, CLN(C2))

IF BOOL THEN C1 ELSE CONJUGATE(C1)
```

CONDITIONAL EXPRESSION

A conditional expression is one that returns one of two possible values, depending upon a specified condition.

Syntax



See also
<conditional arithmetic expression> 476
<conditional Boolean expression>. 495
<conditional complex expression>. 507
<conditional designational expression>. 512
<conditional pointer expression>. 520

Semantics

Conditional expressions are of the form

IF <Boolean expression> THEN <expression> ELSE <expression>

Either the first or the second expression is selected for evaluation, depending on the value of the Boolean expression. The two alternative expressions must be of the same kind: for example, two arithmetic expressions or two Boolean expressions.

The selection process proceeds as follows:

- The Boolean expression following "IF" is evaluated.
- If the resulting value is TRUE, the expression following "THEN" is evaluated, and the expression following "ELSE" is ignored.
- If the resulting value is FALSE, the expression following "THEN" is ignored, and the expression following "ELSE" is evaluated.

If either of the two expressions is itself a conditional expression, the process is repeated until an unconditional expression is selected for evaluation.

The type of a conditional arithmetic expression is DOUBLE if either of its constituent expressions is of type DOUBLE; in this case, a constituent expression that is not of type DOUBLE is extended to double precision. The type of a conditional arithmetic expression is INTEGER if and only if both of its constituent expressions are of type INTEGER. Otherwise, a conditional arithmetic expression is of type REAL

Examples

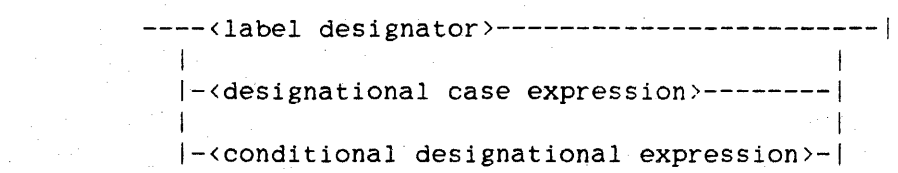
```
IF BOOL THEN 47 ELSE 95

IF A = B THEN BOOL ELSE FALSE

IF NOT BOOL THEN C1 ELSE C2

IF ALLDONE THEN EOJLBL ELSE NEXTLBL

IF CHAR.SZF = 8 THEN PTRINEBCDIC ELSE PTRINHEX
```

DESIGNATIONAL EXPRESSION

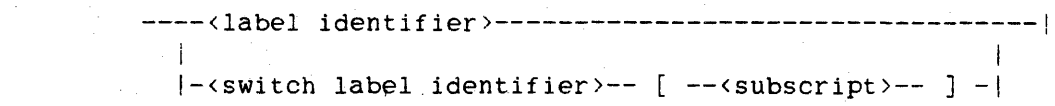
Designational expressions are expressions that return a value that is a label.

Syntax

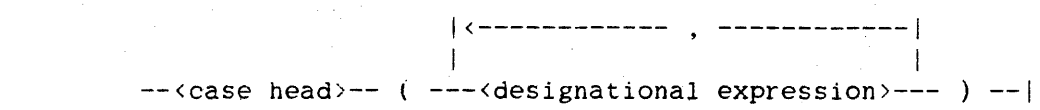
<designational expression>



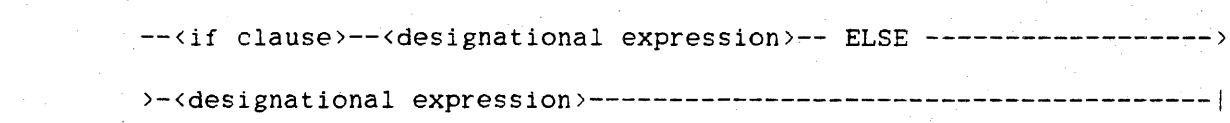
<label designator>



<designational case expression>



<conditional designational expression>



See also

Semantics

If a designational expression is a label identifier, then the value of the expression is that label.

If a designational expression is a subscripted switch label identifier, then the numerical value of the subscript designates one of the elements in the switch label list. The value of the subscript is rounded, if necessary, to an integer. This value is used as an index into the switch label list. The entries of the list are numbered sequentially from 1 through N, where N is the number of entries in the list. The entry corresponding to the value of the subscript is selected. If the value of the subscript is outside the range of the switch label list, program control continues to the next statement without any error indication. (For more information about the switch label list, refer to "SWITCH LABEL Declaration.")

The evaluation of a designational case expression is described in "Case Expression."

The evaluation of a conditional designational expression is described in "Conditional Expression."

Examples

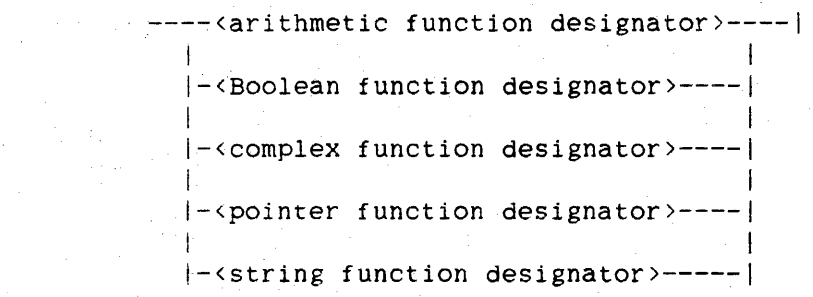
```
ENDLABEL

CHOOSELABEL[I+2]

CASE X OF (GOTDATA, GOTERR, GOTREAL, GOTCHANGE, ESCAPE)

IF K = 1 THEN SELECT[2] ELSE START
```

FUNCTION EXPRESSION

A function expression is an expression that returns a single value that is the result of invoking a procedure. The procedure can be declared in the program, or it can be an intrinsic procedure.

Syntax

<function expression>



Semantics

There are two kinds of functions: predefined functions, called intrinsic functions, which are part of the ALGOL language, and programmer-defined functions, which are typed procedures that are declared in the program.

The intrinsic functions are described later in this chapter under the heading "Intrinsic Functions."

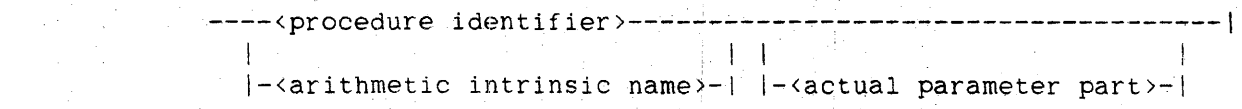
See also
Intrinsic Functions 528

Arithmetic Function Designator

An arithmetic function designator specifies a function that returns an arithmetic value--a value of type INTEGER, REAL, or DOUBLE.

Syntax

<arithmetic function designator>



<arithmetic intrinsic name>

Any of the names listed under the heading "Arithmetic Intrinsic Names" in this chapter.

See also
<actual parameter part> 346
Arithmetic Intrinsic Names. 528
<procedure identifier>. 165

Semantics

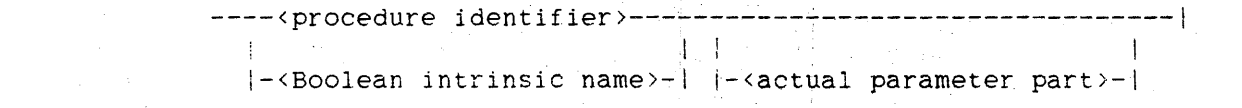
The procedure specified by the procedure identifier must be of type INTEGER, REAL. or DOUBLE.

Boolean Function Designator

A Boolean function designator specifies a function that returns a Boolean value--a value of TRUE or FALSE.

Syntax

<Boolean function designator>



<Boolean intrinsic name>

Any of the names listed under the heading "Boolean Intrinsic Names" in this chapter.

See also
<actual parameter part> 346
Boolean Intrinsic Names 531
<procedure identifier>. 165

Semantics

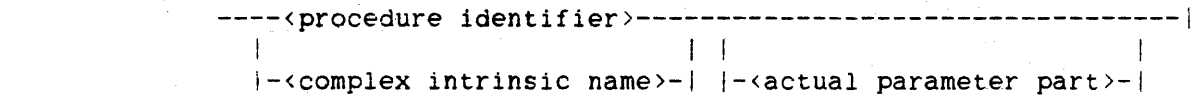
The procedure specified by the procedure identifier must be of type BOOLEAN.

Complex Function Designator

A complex function designator specifies a function that returns a complex value--a value with a real part and an imaginary part.

Syntax

<complex function designator>



<complex intrinsic name>

Any of the names listed under the heading "Complex Intrinsic Names" in this chapter.

See also
<actual parameter part> 346
Complex Intrinsic Names 531
<procedure identifier>. 165

Semantics

The procedure specified by the procedure identifier must be of type COMPLEX.

Pointer Function Designator

A pointer function designator specifies a function that returns a pointer value--a value that can be used to refer to a character position in an array row.

Syntax

<pointer function designator>

--<pointer intrinsic name>--<actual parameter part>--|

<pointer intrinsic name>

Any of the names listed under the heading "Pointer Intrinsic Names" in this chapter.

See also
<actual parameter part> 346
Pointer Intrinsic Names 531

Semantics

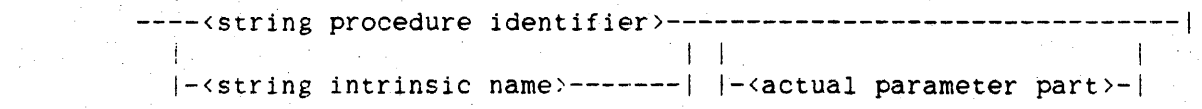
Unlike the other function designators, the syntax for pointer function designator does not allow a procedure identifier as part of the syntax. There is no PROCEDURE declaration that allows declaring a procedure of type POINTER.

String Function Designator

A string function designator specifies a function that returns a string value--a hexadecimal string, an ASCII string, or an EBCDIC string.

Syntax

<string function designator>



<string intrinsic name>

Any of the names listed under the heading "String Intrinsic Names" in this chapter.

See also
<actual parameter part> 346
String Intrinsic Names. 531
<string procedure identifier> 165

Semantics

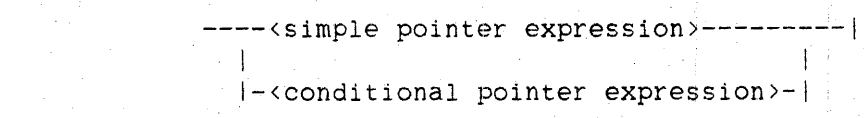
A string procedure identifier designates a procedure declared with a type of HEX STRING, ASCII STRING, or EBCDIC STRING.

POINTER EXPRESSION

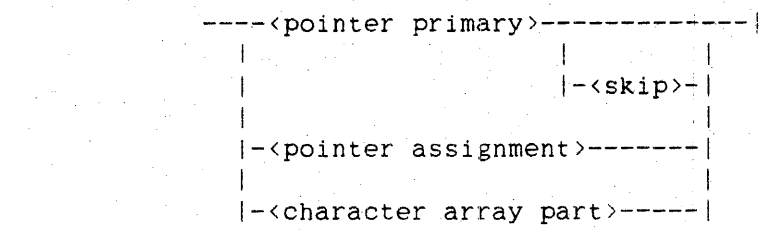
A pointer expression is an expression that returns a value that is a pointer, which can be used to reference a character position in an array row.

Syntax

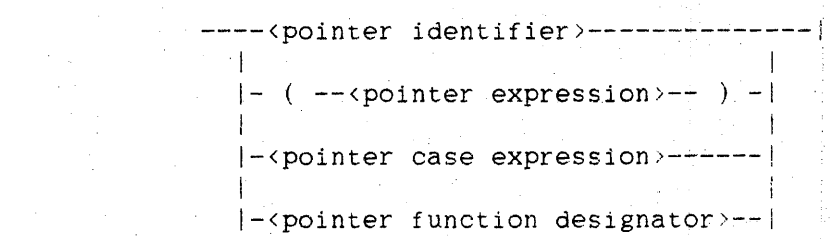
<pointer expression>



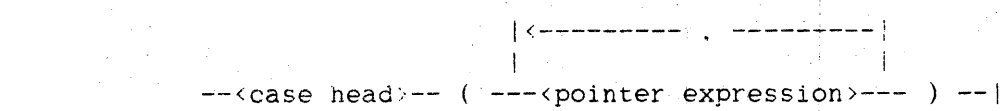
<simple pointer expression>



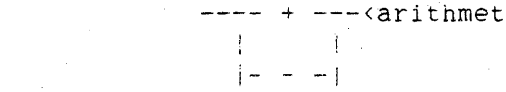
<pointer primary>



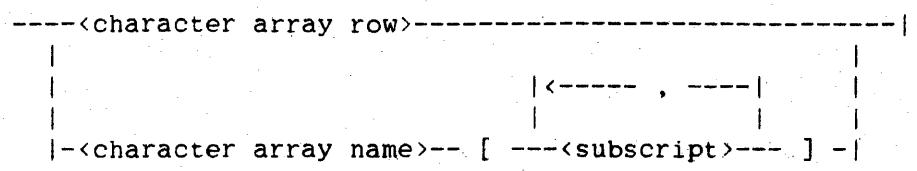
<pointer case expression>



<skip>



<character array part>



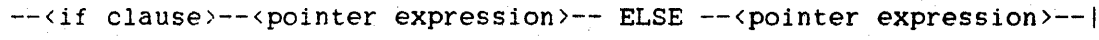
<character array row>

An <array row> whose identifier is declared with a <character type>.

<character array name>

An <array name> whose identifier is declared with a <character type>.

<conditional pointer expression>



See also

Semantics

A pointer must be initialized before it can be used; otherwise, a run-time error occurs. A pointer can be initialized in the following ways:

1. By a pointer assignment
2. By appearing as an update pointer in any of the following:
   - a REPLACE statement
   - a SCAN statement
   - a string relation in a Boolean expression
   - the DOUBLE function
   - the INTEGER function

The evaluation of a conditional pointer expression is described in "Conditional Expression."

<pointer primary>

A one-dimensional array designator or a fully subscripted variable can be interpreted as a pointer primary whenever context determines that no conflict exists with other valid constructs (for example. when a pointer expression is required). This syntax can be used for such constructs as

```
REPLACE A BY B FOR 10 WORDS
```

where A and B are one-dimensional arrays.

The evaluation of a pointer case expression is described in "Case Expression."

<skip>

If the <skip> construct is used, the value of the arithmetic primary determines the adjustment to the value of the pointer primary. If N is the value of the arithmetic primary, the pointer is adjusted as follows:

- If N is less than or equal to zero. the pointer is not adjusted.

- If N is greater than zero, then the pointer is adjusted N characters to the right if the <skip> construct specifies "+", or N characters to the left if it specifies "-". Skipping to the right is defined as incrementing the value of the character index. Skipping to the left is defined as decrementing this value.

If the adjustment to the value of the pointer primary is given by the value of an arithmetic expression, note that the arithmetic expression must be enclosed in parentheses. For example, the expression

```
PTR + X*Y
```

is invalid and will get a compile-time error. It is correctly written as

```
PTR + (X*Y)
```

Pragmatics

The use of a pointer expression to skip up and down an array for more than a few words is expensive. Each word of the array is accessed in order to ensure that no memory-protected words are encountered. For pointer moves of more than a few words, it is faster to re-index the array and use the POINTER function for word arrays, or to re-index the array for character arrays.

Examples

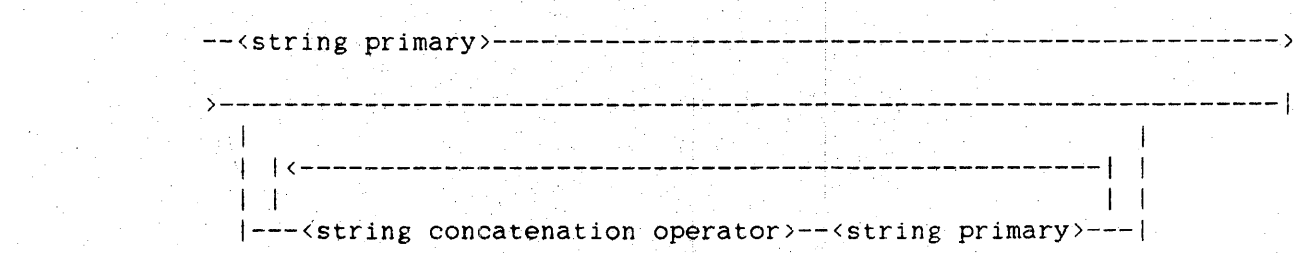
```
PTR

PTS+15

PTR := POINTER(A)

(PTEMP + (X*Y))

HEXARAY

HEXARAY[N]

CASE VAL OF (PTR,PTS,PTEMP,PSORCE)

POINTER(INFO,8)

READLOCK(PTR,POLD)

IF BOOL THEN P1 ELSE POINTER(A)
```

STRING EXPRESSION

A string expression is an expression that returns a value that is a hexadecimal string, EBCDIC string, or ASCII string.

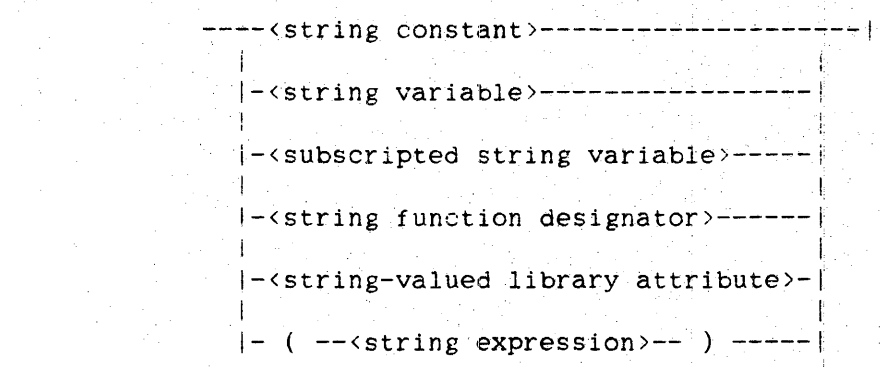
Syntax

<string expression>

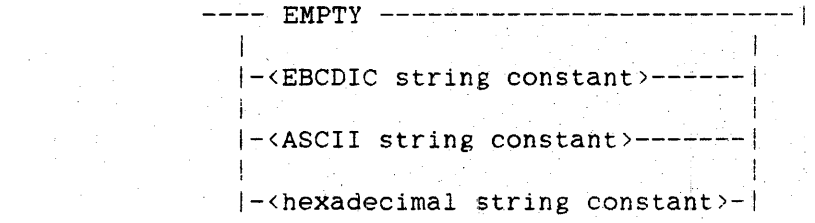


<string concatenation operator>

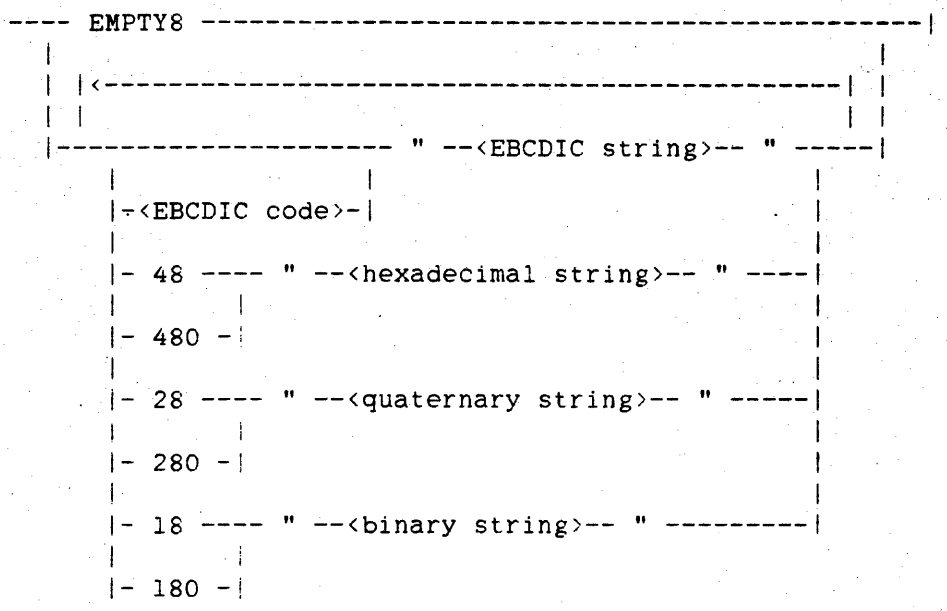
<string primary>



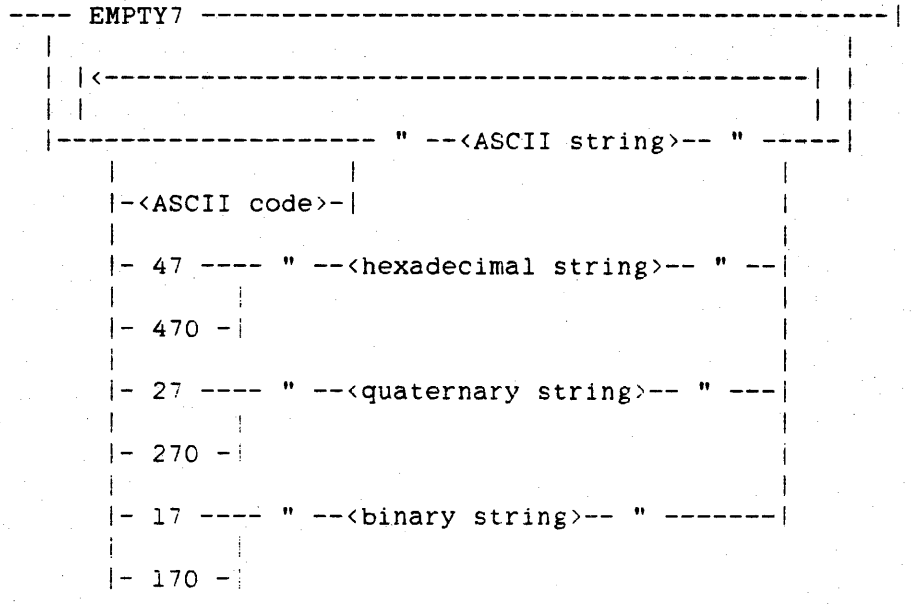
<string constant>



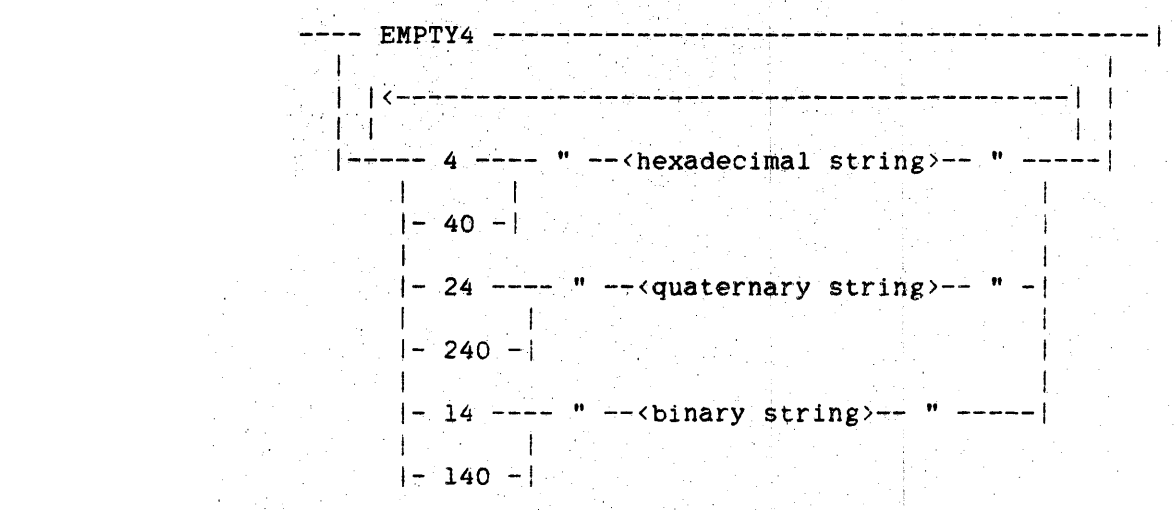
<EBCDIC string constant>



<ASCII string constant>



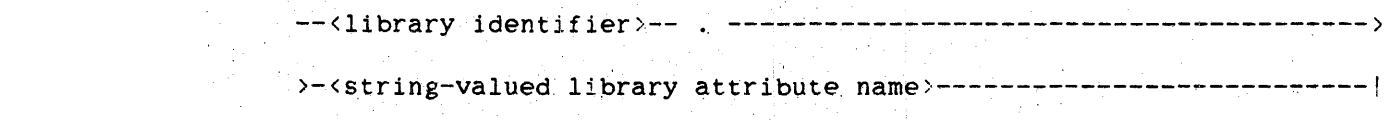
‹hexadecimal string constant›



‹string variable›

‹subscripted string variable›

‹string-valued library attribute›



‹constant string expression›

A ‹string expression› that can be fully evaluated at compile time.

See also

Semantics

In the syntax for EBCDIC string constant, ASCII string constant, and hexadecimal string constant, the string codes determine the interpretation of the characters between the quotation marks (") and has no effect on the justification of a string. A string is always left-justified; therefore, any "0" in a string code is ignored.

The <EBCDIC code> construct is optional in an EBCDIC string constant containing an EBCDIC string only if the default character type is EBCDIC. The <ASCII code> construct is optional in an ASCII string constant containing an ASCII string only if the default character type is ASCII. For more information, refer to "String Code" in the chapter "Language Components" and "Default Character Type" in the appendix "Data Representation."

The reserved words EMPTY8, EMPTY7, and EMPTY4 represent null strings of the character types EBCDIC, ASCII, and hexadecimal, respectively. The reserved word EMPTY represents a null string of the default character type.

For more information about string-valued library attributes, refer to "Library Attributes" in the "Interface to the Library Facility" chapter.

String Concatenation

The operators CAT and || are used to concatenate two strings. The concatenation of two strings yields a new string whose length is the sum of the lengths of the two original strings, and whose value is formed by joining a copy of the second string immediately onto the end of a copy of the first string.

Only strings of the same character type can be concatenated.

If more than one string primary is used in a concatenation operation, they are evaluated from left to right.

Pragmatics

No more than 256 characters can appear between one pair of quotation marks in a string constant; however, as many as 4095 characters can appear in an EBCDIC string constant, ASCII string constant, or hexadecimal string constant.

See also
Default Character Type. 817
Library Attributes. 665
String code 36

Examples

```
8"ABCD123"              % result = ABCD123

""WHY"48"6F""""         % result = "WHY?"

EMPTY8                  % result =

S2 || S3

"AC" || S2 || "123"

S1 || TRANSLATE(S2,HEXTOEBCDIC)

HEAD(S,ALPHA)

TAIL(S,NOT "-")

REPEAT("ABC",3)

STRING(256,*)

TAKE(S,2)

DROP(TAKE(S,4),2)

TRANSLATE(S,HEXTOEBCDIC)
```

6.2 INTRINSIC FUNCTIONS

Intrinsic functions are typed procedures that are predefined in the ALGOL language; that is, intrinsic functions can be used without being declared.

INTRINSIC NAMES BY TYPE RETURNED

The intrinsic functions of ALGOL return values of type INTEGER, REAL, DOUBLE, BOOLEAN, COMPLEX, POINTER, and STRING.

Arithmetic Intrinsic Names

The following intrinsic functions return arithmetic values (values of type INTEGER, REAL, and DOUBLE).

Alphabetical Listing of Arithmetic Intrinsic Functions

The type of the value each function returns, INTEGER, REAL, or DOUBLE, is indicated by I, R, or D, respectively, following the name of the function.

ABS (R)
ARCCOS (R)
ARCSIN (R)
ARCTAN (R)
ARCTAN2 (R)
ARRAYSEARCH (I)
ATANH (R)
CABS (R)
CHECKSUM (R)
CLOSE (I)
COMPILETIME (R)
COS (R)
COSH (R)
COTAN (R)
DABS (D)
DAND (D)
DARCCOS (D)
DARCSIN (D)
DARCTAN (D)
DARCTAN2 (D)
DCOS (D)
DCOSH (D)
DECIMAL (D)
DELINKLIBRARY (I)
DELTA (I)
DEQV (D)
DERF (D)
DERFC (D)
DEXP (D)
DGAMMA (D)
DIMP (D)
DINTEGER (D)
DLGAMMA (D)
DLN (D)
DLOG (D)
DMAX (D)
DMIN (D)
DNABS (D)
DNOT (D)
DOR (D)
DOUBLE (D)
DSCALELEFT (D)
DSCALERIGHT (D)
DSCALERIGHTT (D)
DSIN (D)
DSINH (D)
DSQRT (D)
DTAN (D)
DTANH (D)
ENTIER (I)
ERF (R)
ERFC (R)
EXP (R)
FIRST (R)
FIRSTONE (I)
FIRSTWORD (R)
GAMMA (R)
IMAG (R)
INTEGER (I)
INTEGERT (I)
LENGTH (I)
LINENUMBER (I)
LINKLIBRARY (I)
LISTLOOKUP (I)
LN (R)
LNGAMMA (R)
LOG (R)
MASKSEARCH (I)
MAX (R)
MESSAGESEARCHER (I)
MIN (R)
NABS (R)
NORMALIZE (R)
OFFSET (I)
ONES (I)
OPEN (I)
POTC (D)
POTH (D)
POTL (D)
PROCESSID (I)
RANDOM (R)
READLOCK (R)
REAL (R)
REMAININGCHARS (I)
SCALELEFT (I)
SCALERIGHT (I)
SCALERIGHTF (R)
SCALERIGHTT (I)
SECONDWORD (R)
SETACTUALNAME (I)
SIGN (I)
SIN (R)
SINGLE (R)
SINH (R)
SIZE (I)
SQRT (R)
TAN (R)
TANH (R)
TIME (R)
VALUE (I)
WAIT (I)
WAITANDRESET (I)

Listing of Arithmetic Intrinsic Functions by Type Returned

Intrinsic Functions Returning Values of Type INTEGER

ARRAYSEARCH
CLOSE
DELINKLIBRARY
DELTA
ENTIER
FIRSTONE
INTEGER
INTEGERT
LENGTH
LINENUMBER
LINKLIBRARY
LISTLOOKUP
MASKSEARCH
MESSAGESEARCHER
OFFSET
ONES
OPEN
PROCESSID
REMAININGCHARS
SCALELEFT
SCALERIGHT
SCALERIGHTT
SETACTUALNAME
SIGN
SIZE
VALUE
WAIT
WAITANDRESET

Intrinsic Functions Returning Values of Type REAL

ABS
ARCCOS
ARCSIN
ARCTAN
ARCTAN2
ATANH
CABS
CHECKSUM
COMPILETIME
COS
COSH
COTAN
ERF
ERFC
EXP
FIRST
FIRSTWORD
GAMMA
IMAG
LN
LNGAMMA
LOG
MAX
MIN
NABS
NORMALIZE
RANDOM
READLOCK
REAL
SCALERIGHTF
SECONDWORD
SIN
SINGLE
SINH
SQRT
TAN
TANH
TIME

Intrinsic Functions Returning Values of Type DOUBLE

DABS
DAND
DARCCOS
DARCSIN
DARCTAN
DARCTAN2
DCOS
DCOSH
DECIMAL
DEQV
DERF
DERFC
DEXP
DGAMMA
DIMP
DINTEGER
DLGAMMA
DLN
DLOG
DMAX
DMIN
DNABS
DNOT
DOR
DOUBLE
DSCALELEFT
DSCALERIGHT
DSCALERIGHTT
DSIN
DSINH
DSQRT
DTAN
DTANH
POTC
POTH
POTL

Boolean Intrinsic Names

The following intrinsic functions return values of type BOOLEAN.

ACCEPT	READ
AVAILABLE	READLOCK
BOOLEAN	REMOVEFILE
CHANGEFILE	SEEK
CHECKPOINT	SPACE
FIX	WAIT
FREE	WRITE
HAPPENED	

Complex Intrinsic Names

The following intrinsic functions return values of type COMPLEX.

CCOS	CONJUGATE
CEXP	CSIN
CLN	CSQRT
COMPLEX	

Pointer Intrinsic Names

The following intrinsic functions return values of type POINTER.

POINTER
READLOCK

String Intrinsic Names

The following intrinsic functions return values of type STRING.

DROP	STRING7
HEAD	STRING8
REPEAT	TAIL
STRING	TAKE
STRING4	TRANSLATE

INTRINSIC FUNCTION DESCRIPTIONS

For arithmetic intrinsic functions, all arithmetic parameters are assumed to be call-by-value. For a further description of some of the arithmetic intrinsic functions, refer to the chapter "Mathematical Functions" in the "System Software Utilities Reference Manual."

<abs function>

```
-- ABS -- ( --<arithmetic expression>-- ) --|
```

The ABS function returns, as a real value, the absolute value of the specified arithmetic expression.

<accept statement>

The ACCEPT statement returns a Boolean value. For more information, refer to "ACCEPT Statement."

<arccos function>

```
-- ARCCOS -- ( --<arithmetic expression>-- ) --|
```

The ARCCOS function returns, as a real value, the principal value of the arccosine (in radians) of the specified arithmetic expression, where -1 < <arithmetic expression> < 1. If the value of the arithmetic expression is not in this range, a run-time error occurs.

<arcsin function>

```
-- ARCSIN -- ( --<arithmetic expression>-- ) --|
```

The ARCSIN function returns, as a real value, the principal value of the arcsine (in radians) of the specified arithmetic expression, where -1 < <arithmetic expression> < 1. If the value of the arithmetic expression is not in this range, a run-time error occurs.

<arctan function>

```
-- ARCTAN -- ( --<arithmetic expression>-- ) --|
```

The ARCTAN function returns, as a real value, the principal value of the arctangent (in radians) of the specified arithmetic expression.

<arctan2 function>

```
-- ARCTAN2 -- ( --<arithmetic expression>-- , ---------------------->
>-<arithmetic expression>-- ) ------------------------------------|
```

The ARCTAN2 function returns, as a real value, the arctangent (in radians) of

first <arithmetic expression> / second <arithmetic expression>

The returned value is adjusted to fall in the range (-pi to +pi) by the following formula:

Let X be the value of the first arithmetic expression and Y be the value of the second arithmetic expression.

If Y > 0,

ARCTAN2(X,Y) = ARCTAN(X/Y)

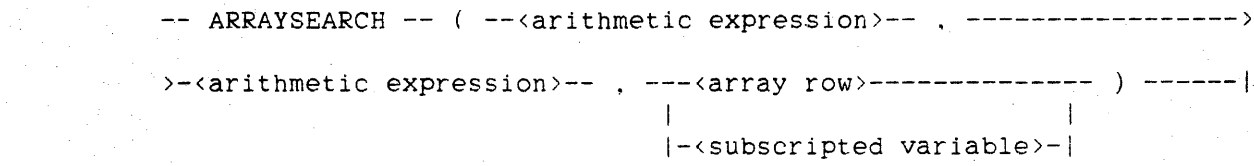
If Y = 0,

ARCTAN2(X,Y) = SIGN(X) * pi / 2

If Y < 0,

ARCTAN2(X,Y) = ARCTAN(X/Y) + SIGN(X) * pi

<arraysearch function>



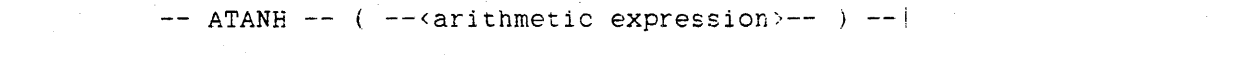
See also
<array row> 43
<subscripted variable>. 225

The ARRAYSEARCH function searches for a specific value within an array. The first arithmetic expression is the value being searched for (the target value). The second arithmetic expression is the mask to be used in the search. The third parameter is a row or subscripted variable of an array of type INTEGER, REAL, BOOLEAN, or COMPLEX.

If the third parameter is an array row, the search begins with the last element of the specified array row; otherwise, the search begins with the element specified by the subscripted variable. Each element, in turn, is retrieved, logically ANDed with the value of the mask, and compared, using the IS operator, with the target value, which has also been logically ANDed with the mask. If the comparison yields TRUE, the function returns an integer value equal to the difference between the subscript of the element where the value is found and the subscript of the first element in the array. If the comparison yields FALSE, the subscript of the element to be retrieved is decremented by one, and the search continues until either a match is found or the first element of the array has been examined. If no match is found, -1 is returned.

The ARRAYSEARCH function can be used on paged (segmented) arrays and unpaged (unsegmented) arrays.

<atanh function>



The ATANH function returns, as a real value, the hyperbolic arctangent of the specified arithmetic expression.

<available function>

```
-- AVAILABLE -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

The AVAILABLE function is a Boolean function that returns TRUE if the available state of the specified event is TRUE (available) and returns FALSE if the available state is FALSE (not available).

<Boolean function>

```
-- BOOLEAN -- ( --<arithmetic expression>-- ) --|
```

The BOOLEAN function returns the value of the arithmetic expression as a Boolean value. If the arithmetic expression is double precision, its value is first truncated to single precision.

<cabs function>

```
-- CABS -- ( --<complex expression>-- ) --|
```

The CABS function returns, as a real value, the absolute value of the specified complex expression.

<ccos function>

```
-- CCOS -- ( --<complex expression>-- ) --|
```

The CCOS function returns, as a complex value, the complex cosine of the specified complex expression.

<cexp function>

```
-- CEXP -- ( --<complex expression>-- ) --|
```

The CEXP function returns, as a complex value,

```
e ** <complex expression>
```

where e is the base of the natural logarithms.

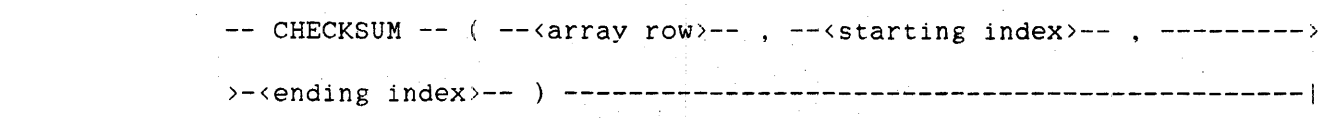
<changefile statement>

The CHANGEFILE statement returns a Boolean value. For more information, refer to "CHANGEFILE Statement."

<checkpoint statement>

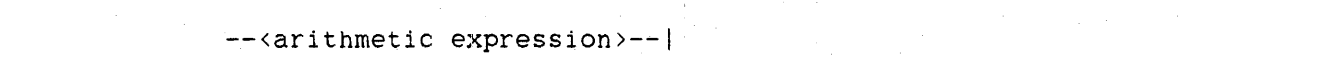
The CHECKPOINT statement returns a Boolean value. For more information, refer to "CHECKPOINT Statement."

<checksum function>



<starting index>

<ending index>



See also
<array row> 43

The CHECKSUM function returns, as a real value, a hash function of all bits in the words of the array row beginning with the word indexed by <starting index> and up to, but not including, the word indexed by <ending index>. Both <starting index> and <ending index> are rounded to an integer value before they are used to index the array row. The value of this function can be used to verify the integrity of data being transferred or stored. The array row must be single precision and unpaged (unsegmented).

<cln function>

The CLN function returns, as a complex value, the natural logarithm of the specified complex expression.

<close statement>

The CLOSE statement returns an integer value. For more information, refer to "CLOSE Statement."

<compiletime function>

```
-- COMPILETIME -- ( --<constant arithmetic expression>-- ) --|
```

See also
<constant arithmetic expression>. 476

The COMPILETIME function obtains various system time values at compile time, for use by the object program. The form of the value returned by the COMPILETIME function is the same as that returned by the TIME function for the same argument. The returned value is computed by the compiler using the TIME function at compile time. For more information, refer to the TIME function in this section.

COMPILETIME(20) returns, in integer form, the program version number as set by the most recent VERSION compiler control option.

COMPILETIME(21) returns, in integer form, the program cycle number as set by the most recent VERSION compiler control option.

COMPILETIME(22) returns, in integer form, the program patch number as set by the most recent VERSION compiler control option.

<complex function>

```
-- COMPLEX -- ( --<arithmetic expression>-- , ---------------------->

>-<arithmetic expression>-- ) ------------------------------------|
```

The COMPLEX function returns, as a complex value,

first <arithmetic expression> + i * second <arithmetic expression>

where i is the square root of -1. The arithmetic expressions are first rounded to single precision, if necessary.

<conjugate function>

-- CONJUGATE -- (--<complex expression>--) --|

The CONJUGATE function returns, as a complex value, the complex conjugate of the specified complex expression.

<cos function>

-- COS -- (--<arithmetic expression>--) --|

The COS function returns, as a real value, the cosine of an angle of <arithmetic expression> radians.

<cosh function>

-- COSH -- (--<arithmetic expression>--) --|

The COSH function returns, as a real value, the hyperbolic cosine of the specified arithmetic expression.

<cotan function>

-- COTAN -- (--<arithmetic expression>--) --|

The COTAN function returns, as a real value, the cotangent of an angle of <arithmetic expression> radians.

<csin function>

-- CSIN -- (--<complex expression>--) --|

The CSIN function returns, as a complex value, the complex sine of the specified complex expression.

<csqrt function>

```
-- CSQRT -- ( --<complex expression>-- ) --|
```

The CSQRT function returns, as a complex value, the square root of the specified complex expression.

<dabs function>

```
-- DABS -- ( --<arithmetic expression>-- ) --|
```

The DABS function returns, as a double-precision value, the absolute value of the specified arithmetic expression.

<dand function>

```
-- DAND -- ( --<arithmetic expression>-- , ------------------------>

>-<arithmetic expression>-- ) -----------------------------------|
```

The DAND function returns, as a double-precision value,

first <arithmetic expression> AND second <arithmetic expression>

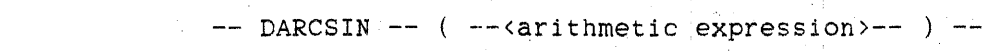
The arithmetic expressions are first extended to double precision, if necessary, and the AND operation is performed on all 96 bits.

<darccos function>

```
-- DARCCOS -- ( --<arithmetic expression>-- ) --|
```

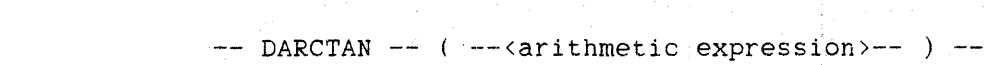
The DARCCOS function returns, as a double-precision value, the principal value of the arccosine (in radians) of the specified arithmetic expression, where -1 < <arithmetic expression> < 1. If the value of the arithmetic expression is not in this range, a run-time error occurs.

<darcsin function>



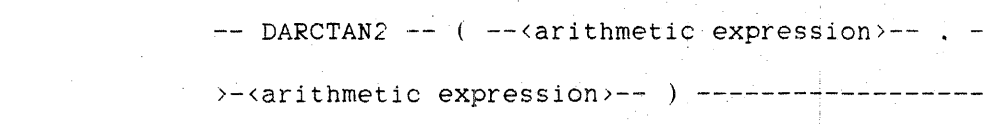
The DARCSIN function returns, as a double-precision value, the principal value of the arcsine (in radians) of the specified arithmetic expression, where -1 < <arithmetic expression> < 1. If the value of the arithmetic expression is not in this range, a run-time error occurs.

<darctan function>



The DARCTAN function returns, as a double-precision value, the principal value of the arctangent (in radians) of the specified arithmetic expression.

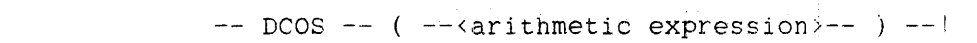
<darctan2 function>



The DARCTAN2 function returns, as a double-precision value, the principal value of the arctangent (in radians) of

first <arithmetic expression> / second <arithmetic expression>

<dcos function>



The DCOS function returns, as a double-precision value, the cosine of an angle of <arithmetic expression> radians.

<dcosh function>

```
-- DCOSH -- ( --<arithmetic expression>-- ) --|
```

The DCOSH function returns, as a double-precision value, the hyperbolic cosine of the specified arithmetic expression.

<decimal function>

```
-- DECIMAL -- ( --<string expression>-- ) --|
```

The DECIMAL function returns the double-precision value represented by the string expression.

The string expression must yield a valid <number> on evaluation (see "Number" in the chapter "Language Components"). For example, the assignment

```
D := DECIMAL(STR);
```

is valid for the strings

```
STR = "+5497823"
STR = "1.75@-46"
STR = "-4.31468"
STR = "@2"
STR = "+549"||"7823"
```

However, for the strings

```
STR = "50 00."
STR = "1,505,278.00"
STR = "1@2.5"
STR = ".573"||"5.82"
```

the program is given a run-time error.

<delinklibrary function>

```
-- DELINKLIBRARY -- ( --<library identifier>-- ) --|
```

See also
<library identifier>. 129

The DELINKLIBRARY function delinks the program from the librar specified by the library identifier. The DELINKLIBRARY function affects only the linkage between the program and the indicated library; other programs using the library program are not affected. This function returns an integer value that indicates success or failure and the reason for failure. The values returned by the function can be interpreted as follows:

Value	Meaning
1	The library has been delinked from the program.
0	The library was not linked to the program.
-1	The library structure could not be accessed; a system fault has occurred.

<delta function>

```
-- DELTA -- ( --<pointer expression>-- , --<pointer expression>---->
>- ) ------------------------------------------------------------|
```

The DELTA function returns, as an integer value, the number of characters between the character position referenced by the first pointer expression and the character position referenced by the second pointer expression. The value is calculated as follows: the character position referenced by the first pointer expression is subtracted from the character position referenced by the second pointer expression.

For a function that returns the number of characters between the character position referenced by a pointer expression and the beginning of the array row, see the OFFSET function in this section. For a function that returns the number of characters between the character position referenced by a pointer expression and the end of the array row, see the REMAININGCHARS function in this section.

<deqv function>

```
-- DEQV -- ( --<arithmetic expression>-- , ----------------------->

>-<arithmetic expression>-- ) ------------------------------------|
```

The DEQV function returns, as a double-precision value,

first <arithmetic expression> EQV second <arithmetic expression>

Both arithmetic expressions are first extended to double precision, if necessary, and the EQV operation is performed on all 96 bits.

<derf function>

```
-- DERF -- ( --<arithmetic expression>-- ) --|
```

The DERF function returns, as a double-precision value, the value of the standard error function at the specified arithmetic expression. For any valid N, DERF(-N) = -DERF(N).

<derfc function>

```
-- DERFC -- ( --<arithmetic expression>-- ) --|
```

The DERFC function returns, as a double-precision value, the complement of the value of the standard error function at the specified arithmetic expression. For any valid N, DERFC(N) = 1 - DERF(N).

<dexp function>

```
-- DEXP -- ( --<arithmetic expression>-- ) --|
```

The DEXP function returns, as a double-precision value,

e ** <arithmetic expression>

where e is the base of the natural logarithms.

<dgamma function>

```
-- DGAMMA -- ( --<arithmetic expression>-- ) --|
```

The DGAMMA function returns, as a double-precision value, the value of the gamma function at the specified arithmetic expression.

<dimp function>

```
-- DIMP -- ( --<arithmetic expression>-- , ----------------------->

>-<arithmetic expression>-- ) ------------------------------------|
```

The DIMP function returns, as a double-precision value,

first <arithmetic expression> IMP second <arithmetic expression>

Both arithmetic expressions are first extended to double precision, if necessary, and the IMP operation is performed on all 96 bits.

<dinteger function>

```
-- DINTEGER -- ( --<arithmetic expression>-- ) --|
```

The DINTEGER function returns the value of the arithmetic expression as a double-precision integer value. Specifically, the function returns

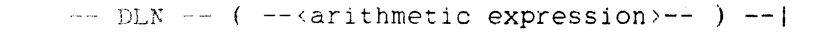
DOUBLE(ENTIER(<arithmetic expression> + 0.5))

<dlgamma function>

```
-- DLGAMMA -- ( --<arithmetic expression>-- ) --|
```

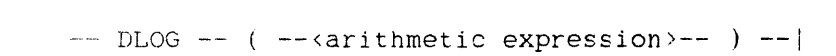
The DLGAMMA function returns, as a double-precision value, the natural logarithm of the gamma function at the specified arithmetic expression.

<dln function>



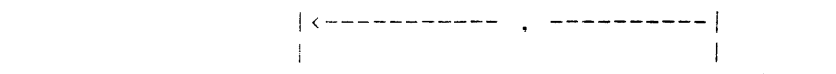
The DLN function returns, as a double-precision value, the natural logarithm of the specified arithmetic expression.

<dlog function>



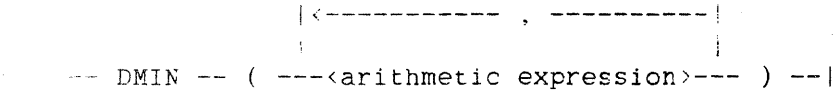
The DLOG function returns, as a double-precision value, the base-10 logarithm of the specified arithmetic expression.

<dmax function>



The DMAX function returns, as a double-precision value, the maximum of the values of all the specified arithmetic expressions.

<dmin function>



The DMIN function returns, as a double-precision value, the minimum of the values of all the specified arithmetic expressions.

<dnabs function>

```
-- DNABS -- ( --<arithmetic expression>-- ) --|
```

The DNABS function returns, as a double-precision value, the negative of the absolute value of the specified arithmetic expression.

<dnot function>

```
-- DNOT -- ( --<arithmetic expression>-- ) --|
```

The DNOT function returns, as a double-precision value, the logical complement of the value of the arithmetic expression. The arithmetic expression is first extended to double precision, if necessary, and the NOT operation is performed on all 96 bits.

<dor function>

```
-- DOR -- ( --<arithmetic expression>-- , ------------------------>

>-<arithmetic expression>-- ) ------------------------------------|
```

The DOR function returns, as a double-precision value,

first <arithmetic expression> OR second <arithmetic expression>

Both arithmetic expressions are first extended to double precision, if necessary, and the OR operation is performed on all 96 bits.

<double function>

The DOUBLE function has three forms, each of which returns a double-precision value.

```
-- DOUBLE -- ( --<arithmetic expression>-- ) --|
```

This form of the DOUBLE function returns the value of the arithmetic expression extended to a double-precision value.

```
-- DOUBLE -- ( --<arithmetic expression>-- , --<arithmetic expression>->

>- ) ----------------------------------------------------------------|
```

This form of the DOUBLE function returns a double-precision value in which the first word is equal to the value of the first arithmetic expression and the second word is equal to the value of the second arithmetic expression. The arithmetic expressions are first truncated to single precision, if necessary.

```
-- DOUBLE -- ( ----------------------<pointer expression>-- , --------->
                |                    |
                |-<update pointer>-|

>-<arithmetic expression>-- ) ---------------------------------------|
```

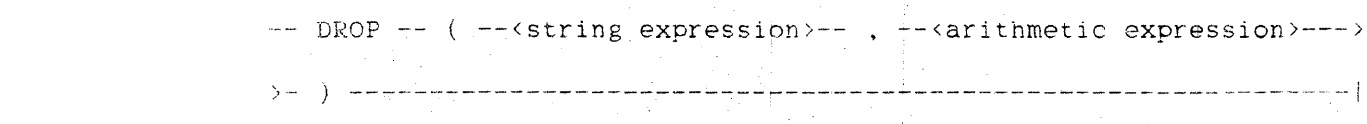
See also
<update pointer>. 379

This form of the DOUBLE function returns, as a double-precision value, the decimal value represented by the string of characters starting with the character pointed to by the pointer expression. The arithmetic expression specifies the length of the string and must have a value, when rounded to an integer, less than 24.

A zone field configuration of 1"1101" for 8-bit characters or 1"10" for 6-bit characters in the least significant character position causes the result of the function to be negative. With 4-bit characters, a 1"1101" in the most significant character position results in a negative value.

The state of the pointer expression when the count is exhausted can be preserved by using the <update pointer> construct.

<drop function>



The DROP function returns a string with a value equal to the value of the string expression with the first <arithmetic expression> characters deleted. The value of the arithmetic expression is rounded to an integer, if necessary. An error occurs if the rounded value of the arithmetic expression is greater than the number of characters in the string expression or less than zero. If the rounded value of the arithmetic expression is zero, the result is the same as the value of the string expression. If the rounded value of the arithmetic expression is equal to the length of the string expression, the result is the null string.

The DROP function and the TAKE function are complementary functions. This means that for any string expression S and any arithmetic expression A in the range 0 <= A <= LENGTH(S), the following relation is always TRUE:

S = TAKE(S,A) CAT DROP(S,A)

See the TAKE function in this section.

Examples

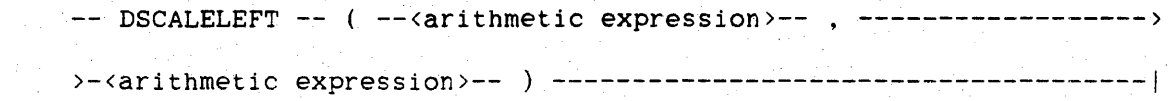
In the examples below, string S has a length of 6 and contains 8"ABCDEF".

DROP(S,2) = 8"CDEF"

DROP(TAKE(S,4),2) = 8"CD"

DROP(S,6) = the null string

<dscaleleft function>

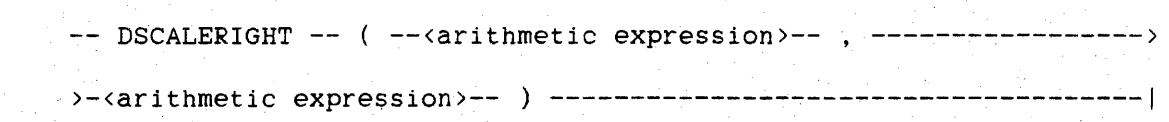


The DSCALELEFT function returns, as a double-precision value,

first <arithmetic expression>*(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. The DSCALELEFT function is undefined when the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<dscaleright function>



The DSCALERIGHT function returns, as a double-precision value, the rounded result of

first <arithmetic expression>/(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<dscalerightt function>

```
-- DSCALERIGHTT -- ( --<arithmetic expression>-- , ---------------->
>-<arithmetic expression>-- ) ----------------------------------|
```

The DSCALERIGHTT function returns, as a double-precision value, the truncated result of

first <arithmetic expression>/(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<dsin function>

```
-- DSIN -- ( --<arithmetic expression>-- ) --|
```

The DSIN function returns, as a double-precision value, the sine of an angle of <arithmetic expression> radians.

<dsinh function>

```
-- DSINH -- ( --<arithmetic expression>-- ) --|
```

The DSINH function returns, as a double-precision value, the hyperbolic sine of the specified arithmetic expression.

<dsqrt function>

```
-- DSQRT -- ( --<arithmetic expression>-- ) --|
```

The DSQRT function returns, as a double-precision value, the square root of the specified arithmetic expression. The value of the arithmetic expression must be greater than or equal to zero.

<dtan function>

```
-- DTAN -- ( --<arithmetic expression>-- ) --|
```

The DTAN function returns, as a double-precision value, the tangent of an angle of <arithmetic expression> radians.

<dtanh function>

```
-- DTANH -- ( --<arithmetic expression>-- ) --|
```

The DTANH function returns, as a double-precision value, the hyperbolic tangent of the specified arithmetic expression.

<entier function>

```
-- ENTIER -- ( --<arithmetic expression>-- ) --|
```

The ENTIER function returns the largest integer not greater than the value of the arithmetic expression.

Because of the limitations of finite representation arithmetic, the ENTIER function erroneously returns zero, rather than -1, for negative numbers of small magnitude. For a single-precision expression, the threshold is -0.5 * 8**-13 (approximately -9.09E-13). This number correctly yields -1, whereas the next smaller (in magnitude) single-precision number incorrectly yields zero. For a double-precision expression, the threshold can differ on various systems (because of different algorithms for double-precision rounding), but it is in the vicinity of -8**-26 (approximately -3.31E-24).

The ENTIER function returns an incorrect result for single-precision, negative integers less than or equal to -(8**32). The result of the function for an integer expression should be equal to the value of the expression. However, ENTIER(-68719476736) returns a value of -68719476737.

The ENTIER function is not a simple truncation function, as illustrated by the examples. For a simple truncation function, see the INTEGERT function in this section.

Examples

```
ENTIER(2.6) = 2

ENTIER(3.1) = 3

ENTIER(-0.01) = -1

ENTIER(-3.4) = -4

ENTIER(-1.8) = -2
```

<erf function>

```
    -- ERF -- ( --<arithmetic expression>-- ) --|
```

The ERF function returns, as a real value, the value of the standard error function at the specified arithmetic expression. For any valid N, ERF(-N) = -ERF(N).

<erfc function>

```
    -- ERFC -- ( --<arithmetic expression>-- ) --|
```

The ERFC function returns, as a real value, the complement of the value of the standard error function at the specified arithmetic expression. For any valid N, ERFC(N) = 1 - ERF(N).

<exp function>

```
    -- EXP -- ( --<arithmetic expression>-- ) --|
```

The EXP function returns, as a real value,

```
    e ** <arithmetic expression>
```

where e is the base of the natural logarithms.

<first function>

```
-- FIRST -- ( --<string expression>-- ) --|
```

The FIRST function returns, as a real value, the ordinal position in the EBCDIC, ASCII, or hexadecimal collating sequence of the first character in the string expression. This function returns an ordinal position in the EBCDIC collating sequence if the string expression is EBCDIC, an ordinal position in the ASCII collating sequence if the string expression is ASCII, and an ordinal position in the hexadecimal collating sequence if the string expression is hexadecimal. A run-time error occurs if the string expression is the null string.

Examples

```
FIRST("ABC")    = 193 (4"C1")

FIRST(7"NNXX") =  78 (4"4E")

FIRST(4"F1F2") =  15 (4"0F")
```

<firstone function>

```
-- FIRSTONE -- ( --<arithmetic expression>-- ) --|
```

The FIRSTONE function returns, as an integer value, the bit number plus 1 of the leftmost nonzero bit in the value of the arithmetic expression. The FIRSTONE function returns zero if all the bits are zero. If the expression is double precision, only the first word of its value is examined.

<firstword function>

```
-- FIRSTWORD -- ( ------------------------------------------------>

>-<arithmetic expression>-------------------------- ) -------------|
                          |                        |
                          |- . --<real variable>-|
```

See also

<real variable> 447

The FIRSTWORD function returns. as a real value, the first word of the double-precision arithmetic expression. The arithmetic expression is first extended to double precision, if necessary. If the real variable parameter is specified, the second word of the double-precision arithmetic expression is stored in the variable.

<fix statement>

The FIX statement returns a Boolean value. For more information, refer to "FIX Statement."

<free statement>

The FREE statement returns a Boolean value. For more information, refer to "FREE Statement."

<gamma function>

```
-- GAMMA -- ( --<arithmetic expression>-- ) --|
```

The GAMMA function returns, as a real value, the value of the gamma function at the specified arithmetic expression.

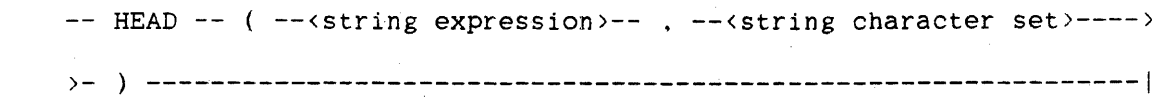
<happened function>

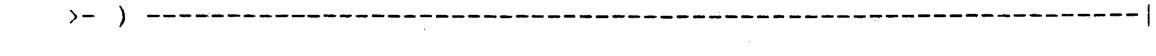
```
-- HAPPENED -- ( --<event designator>-- ) --|
```

See also
<event designator>. 78

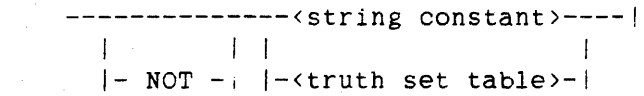
The HAPPENED function is a Boolean function that returns TRUE if the happened state of the specified event is TRUE (happened) and returns FALSE if the happened state is FALSE (not happened).

<head function>





<string character set>



See also
<string constant> 523
<truth set table> 382

The HEAD function returns a string whose value consists of the leftmost characters of the string expression up to, but not including, the first character that is not a member of the string character set. If the first character of the string expression is not a member of the string character set, the null string is returned. If all characters of the string expression are members of the string character set, the entire string expression is returned.

The string character set must be of the same character type as the string expression. If a truth set table is used, it must not be composed of characters of different character types. "NOT" indicates a string character set made up of all characters that are not specified in the string constant or truth set table, but that are of the same character type as the string expression.

The HEAD function and the TAIL function are complementary functions. This means that for any string expression S and any string character set C, the following relation is always TRUE:

```
S = HEAD(S,C) CAT TAIL(S,C)
```

See the TAIL function in this section.

Examples

In the examples below, S is a string of length 9 that contains 8"ABC/1-2+3".

HEAD(S,NOT "/") = 8"ABC"

HEAD(S,ALPHA) = 8"ABC"

HEAD(S,"123") = the null string

<imag function>

-- IMAG -- (--<complex expression>--) --|

The IMAG function returns, as a real value, the imaginary part of the specified complex expression.

For a function that returns the real part of a complex expression, see the REAL function in this section.

<integer function>

The INTEGER function has two forms, each of which returns an integer value.

-- INTEGER -- (--<arithmetic expression>--) --|

This form of the INTEGER function returns the value of the arithmetic expression integerized with rounding. Specifically, it returns

ENTIER(<arithmetic expression> + 0.5) ?

```
-- INTEGER -- ( ----------------------<pointer expression>-- , -------->
                |                     |
                |-<update pointer>-|

>-<arithmetic expression>-- ) ----------------------------------------|
```

See also
<update pointer>. 379

This form of the INTEGER function returns, as an integer value, the decimal value represented by the string of characters starting with the character pointed to by the pointer expression. The absolute value of the integer value to be returned must be less than or equal to 549755813887.

The arithmetic expression specifies the length of the string of characters and must have a value, when rounded to an integer, less than 24. If the rounded value of the arithmetic expression is 24 or greater, the program is terminated with a fault. If the length is zero, a result of zero is returned.

A zone field configuration of 1"1101" for 8-bit characters or 1"10" for 6-bit characters in the least significant character position causes the result of the function to be negative. With 4-bit characters, a 1"1101" in the most significant character position results in a negative value.

The state of the pointer expression when the count is exhausted can be preserved by using the <update pointer> construct.

<integert function>

```
-- INTEGERT -- ( --<arithmetic expression>-- ) --|
```

The INTEGERT function returns the value of the arithmetic expression integerized with truncation.

The INTEGERT function does not necessarily return the same value as the ENTIER function. For example, INTEGERT(-1.2) = -1, but ENTIER(-1.2) = -2.

<length function>

```
-- LENGTH -- ( --<string expression>-- ) --|
```

The LENGTH function returns, as an integer value, the number of characters in the string that results from evaluation of the string expression. The null string has a length of zero.

<linenumber function>

```
-- LINENUMBER --|
```

The LINENUMBER function returns, as an integer value, the sequence number of the source file record on which it appears.

<linklibrary function>

```
-- LINKLIBRARY -- ( --<library identifier>-- ) --|
```

See also
<library identifier>. 129

The LINKLIBRARY function determines whether or not the program is currently linked to or is capable of being linked to the library program specified by the library identifier. Results indicating a successful linkage are returned by the LINKLIBRARY function if the program is presently linked to the library or if the program is capable of being linked, in which case the LINKLIBRARY function performs the linkage. If the program cannot be linked to the library, the function returns a result indicating the reason for the failure. In either case, the program continues to execute: that is, use of the LINKLIBRARY function prevents termination or suspension of a program upon an unsuccessful attempt to link to a library.

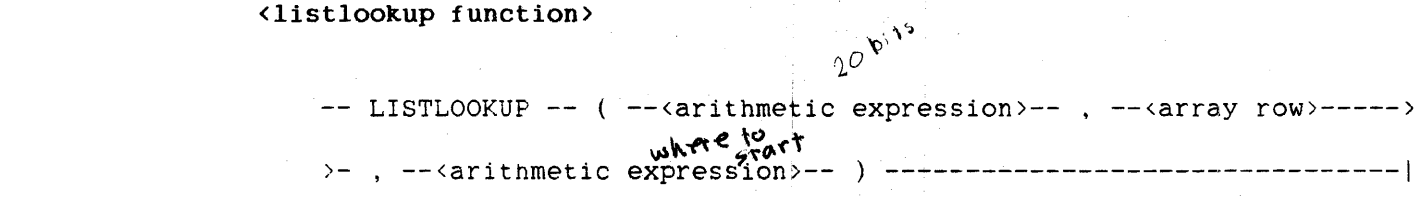
During the linkage process, an attempt is made to link to every entry point exported from the library whose name matches an entry point declared in the program. Only those names that match are checked for correct function type, number of parameters, and parameter types. Therefore, the LINKLIBRARY function does not check that every entry point declared in the program is also exported from the library.

The values returned by the LINKLIBRARY function can be interpreted as follows:

Value	Meaning
2	Successful linkage was made to the library, but not all entry points were provided.
1	Successful linkage was made to the library, and all entry points were provided.
0	The program was already linked to the specified library at the time of the LINKLIBRARY call.
-1	The library code file is missing.
-2	The family size of a process running in Swapper was exceeded on the attempt to link to the library.
-3	A by-calling procedure specified more than one library task.
-4	The library is not a system library.
-5	The number or types of parameters provided by the library do not correspond to those declared in the program.
-6	The library was terminated, canceled, or thawed before being frozen; therefore, the library was not successfully initiated.
-7	A circular chain of library linkages was detected. A library cannot reference itself, either directly or indirectly, through a chain of library references.
-8	A by-calling procedure never specified a library task.
-9	A bad task was passed by a by-calling library procedure.
-10	A library feature that was used is not implemented.
-11	The library template contained an illegal provision type.
-12	The library template level is obsolete. This program must be recompiled.
-13	The library directory level is incompatible with the library template level. The older program must be recompiled.
-14	The program cannot link to a system library.
-15	A task array for a by-calling library was not declared in the library's stack.

Value	Meaning
-16	This program is not authorized to use this library procedure.
-17	The library is not visible to the program attempting to link to the library.

<listlookup function>



See also
<array row> 43

The LISTLOOKUP function causes a linked list of words to be searched and returns, as an integer value, an index into the list as follows:

1. The array row is indexed by the value of the second arithmetic expression and the word is extracted. Each word contains a value (in field [47:28]) and a link (in field [19:20]) to the next word of the linked list.

2. If the value in the extracted word is greater than or equal to the value of the first arithmetic expression, the operation stops, and the index of the word whose link points to the extracted word is returned by the function.

3. If the value in the extracted word is less than the value of the first arithmetic expression, the link of the extracted word is used as an index into the array row, a new word is extracted, and the process is repeated.

A word with a link of zero terminates the search. The value of a word is tested only if the link field is nonzero. If the linked list is exhausted (if a word with a link of zero is encountered), a value of -1 is returned by the LISTLOOKUP function.

<ln function>

```
-- LN -- ( --<arithmetic expression>-- ) --|
```

The LN function returns, as a real value, the natural logarithm of the specified arithmetic expression.

<lngamma function>

```
-- LNGAMMA -- ( --<arithmetic expression>-- ) --|
```

The LNGAMMA function returns, as a real value, the natural logarithm of the gamma function at the specified arithmetic expression.

<log function>

```
-- LOG -- ( --<arithmetic expression>-- ) --|
```

The LOG function returns, as a real value, the base-10 logarithm of the specified arithmetic expression.

<masksearch function>

```
-- MASKSEARCH -- ( --<arithmetic expression>-- , ------------------>

>-<arithmetic expression>-- , ---<array row>-------------- ) ------|
                                |                          |
                                |-<subscripted variable>-|
```

See also
<array row> 43
<subscripted variable>. 225

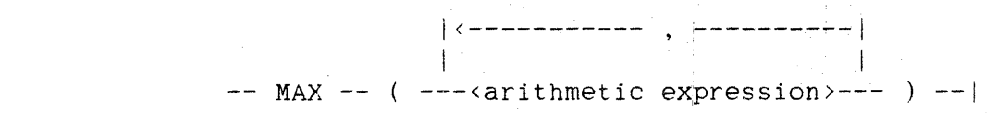
The MASKSEARCH function performs the same operations as the ARRAYSEARCH function, except that it is intended for use only with unpaged (unsegmented) arrays.

If the third parameter to the MASKSEARCH function is an array row or a subscripted variable of a paged (segmented) array, then a warning message is given at run time. This warning message is given only the first time the statement containing the MASKSEARCH function is executed.

If the third parameter is an array row of a paged array, execution of the MASKSEARCH function causes a fault.

If the third parameter is a subscripted variable of a paged array, then no fault is generated at run time, but the results can be unexpected. The MASKSEARCH function searches only the 256-word segment containing the specified array element. If the target value is found in that array segment, the index relative to the beginning of the segment is returned. If the target is not found in that segment, -1 is returned.

<max function>

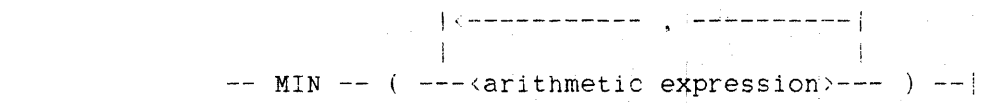


The MAX function returns, as a real value, the maximum of the values of all the specified arithmetic expressions.

<messagesearcher statement>

The MESSAGESEARCHER statement returns an integer value. For more information, refer to "MESSAGESEARCHER Statement."

<min function>



The MIN function returns, as a real value, the minimum of the values of all the specified arithmetic expressions.

<nabs function>

```
-- NABS -- ( --<arithmetic expression>-- ) --|
```

The NABS function returns, as a real value, the negative of the absolute value of the specified arithmetic expression.

<normalize function>

```
-- NORMALIZE -- ( --<arithmetic expression>-- ) --|
```

The NORMALIZE function returns, as a real value, the value of the arithmetic expression, normalized and rounded to a single-precision operand. (For more information on normalized format, refer to "Real Operand" in the appendix "Data Representation.")

See also
Real Operand. 820

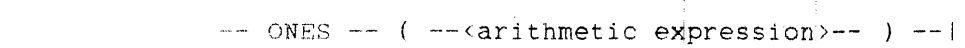
<offset function>

```
-- OFFSET -- ( --<pointer expression>-- ) --|
```

The OFFSET function returns, as an integer value, the number of characters between the character position referenced by the pointer expression and the beginning of the array row, not including the character at which the pointer expression is pointing. The function value is in terms of the character size of the pointer expression. If it is a word pointer, the offset is given in terms of 8-bit characters. If the pointer expression points to a paged (segmented) array, the OFFSET function returns the total offset from the beginning of the first segment of the row.

For a function that returns the number of characters between the character position referenced by a pointer expression and the end of the array row, see the REMAININGCHARS function in this section.

<ones function>

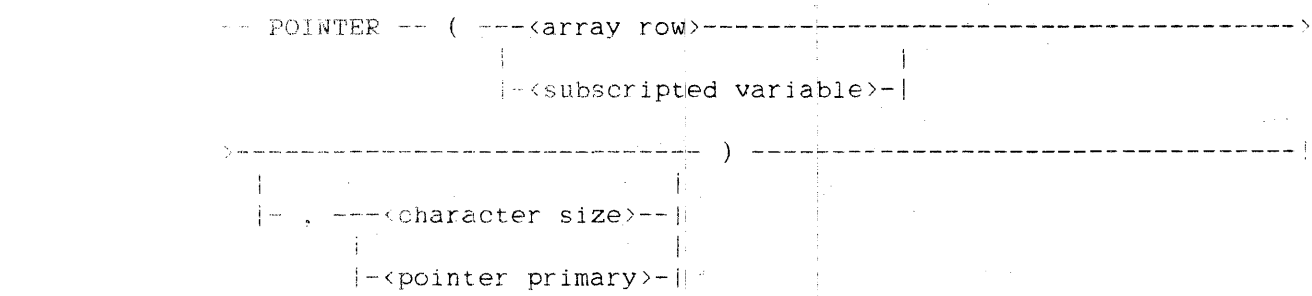


The ONES function returns, as an integer value, the number of nonzero bits in the value of the arithmetic expression. If the arithmetic expression is double precision, all 96 bits of its value are examined.

<OPEN statement>

The OPEN statement returns an integer value. For more information, refer to "OPEN statement."

<pointer function>



<character size>

See also

The POINTER function generates a pointer to the specified location.

If the first parameter is an array row, the pointer references the first character of the first word of the specified array row. If the first parameter is a subscripted variable, the pointer references the first character of the array element specified by the subscripted variable.

If the second parameter is not given, the character size of the pointer is six or eight bits, depending on the default character type. (For more information on the default character type, refer to "Default Character Type" in the appendix "Data Representation.")

If the second parameter is a character size of 4, 6, or 8, the character size of the pointer being generated is four bits, six bits, or eight bits, respectively. If the second parameter is a character size of 7, the character size of the pointer is eight bits. If the second parameter is a character size of 0, the pointer is word-oriented, rather than character-oriented: it is single precision if the array it points to is single precision (INTEGER, REAL, or BOOLEAN) and double precision if the array is double precision (DOUBLE or COMPLEX).

If a pointer primary is given as the second parameter, the character size of that pointer primary is used for the character size of the pointer being generated.

See also

Default Character Type. 817

NOTE

The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as a 6-bit pointer, will cause the program to get a compile-time warning message.

Examples

```
BEGIN
   ARRAY A1,A2[0:9];
   POINTER P1,P2,P3,P4,P5;
   P1 := POINTER(A1);
   P2 := POINTER(A1[9]);
   P3 := POINTER(A1,4);
   P4 := POINTER(A2[6],P3);
   $ SET BCL
   P5 := POINTER(A1):        % LAST STATEMENT
END.
```

Execution of this program has the following results:

P1 This pointer has a character size of eight bits (the default character type is EBCDIC when P1 is assigned) and points to the first 8-bit character of the first element of array A1.

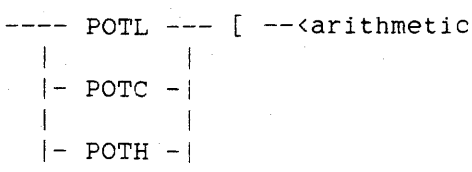
P2 This pointer has a character size of eight bits and points to the first 8-bit character of element 9 of array A1.

P3 This pointer has a character size of four bits and points to the first 4-bit character of the first element of array A1.

P4 This pointer has a character size of four bits and points to the first 4-bit character of element 6 of array A2.

P5 This pointer has a character size of six bits (when P5 is assigned, the default character type is BCL) and points to the first 6-bit character of the first element of array A1.

<pot function>



The POT (Power of Ten) functions, POTL (low), POTC (center), and POTH (high), together provide the value

```
10 ** <arithmetic expression>
```

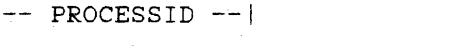
from three tables that are double-precision, read-only arrays. Each of the POT functions is defined only for integerized values of the arithmetic expression falling in the range 0 through 29604. The complete value of 10 ** <arithmetic expression> can be computed using the following arithmetic expression:

```
POTL[<arithmetic expression>.[5:6]]

    * POTC[<arithmetic expression>.[11:6]]

        * POTH[<arithmetic expression>.[14:3]]
```

Examples

```
RESULT := POTL[X]; % WHERE X < 64

DEFINE POT(T) = (POTL[T.[5.6]] * POTC[T.[11:6]] * POTH[T.[14:3]])#:
ONEDIVTENTOI := 1 / POT(I);
```

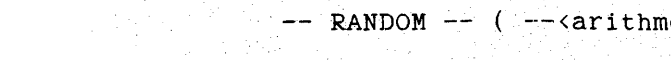
<processid function>



The PROCESSID function returns a positive integer value that uniquely identifies the process executing the function.

The PROCESSID function returns a value that remains unique to that process for the duration of its execution. However, the value is not guaranteed to be the same on every invocation of the PROCESSID function. For example, upon restarting from a checkpoint, the value of the PROCESSID function can have changed. Also, after a process terminates, its PROCESSID value can be assigned to a new process.

<random function>



See also
<arithmetic variable> 225

The RANDOM function returns, as a real value, a random number that is greater than or equal to 0 and less than 1. The arithmetic variable is a call-by-name parameter, and its value is changed each time the function is referenced. A compile-time or run-time error occurs if the parameter is not a single-precision arithmetic variable.

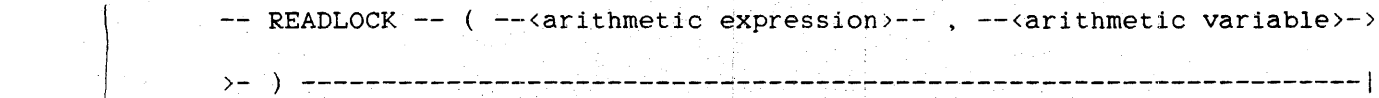
<read statement>

The READ statement returns a Boolean value. For more information, refer to "READ Statement."

<readlock function>

Taking only one memory cycle, the READLOCK function stores the value of the specified expression in the designated variable and returns the previous contents of the variable.

The READLOCK function has three forms. One form returns a real value, one form returns a Boolean value, and one form returns a pointer.



See also
<arithmetic variable> 225

This form of the READLOCK function stores the value of the arithmetic expression in the arithmetic variable and returns the previous contents of the variable as a real value. Both the variable and the expression must be single precision.

```
-- READLOCK -- ( --<Boolean expression>-- , --<Boolean variable>-- ) --|
```

See also
<Boolean variable>. 234

This form of the READLOCK function stores the value of the Boolean expression in the Boolean variable and returns the previous contents of the variable as a Boolean value.

```
-- READLOCK -- ( --<pointer expression>-- , --<pointer variable>-- )   |
```

See also
<pointer variable>. 241

This form of the READLOCK function stores the value of the pointer expression in the pointer variable and returns the previous contents of the variable as a pointer value.

<real function>

The REAL function has four forms, each of which returns a real value.

```
-- REAL -- ( --<arithmetic expression>-- ) --|
```

This form of the REAL function returns the value of the arithmetic expression rounded to a single-precision, real value.

```
-- REAL -- ( --<Boolean expression>-- ) --|
```

This form of the REAL function returns the value of the Boolean expression as a real value. All bits of the Boolean expression are used.

```
-- REAL -- ( --<complex expression>-- ) --|
```

This form of the REAL function returns, as a real value, the real part of the specified complex expression.

For a function that returns the imaginary part of a complex expression, see the IMAG function in this section.

```
-- REAL -- ( --<pointer expression>-- , --<arithmetic expression>-- ) -|
```

This form of the REAL function returns, as a real value, the string of <arithmetic expression> characters starting with the character indicated by the pointer expression.

If the arithmetic expression indicates a string of characters that is less than or equal to 48 bits long, this string is right-justified in one word with leading zeros, if necessary, and this word is returned as the function result. If the arithmetic expression indicates a string of characters more than 48 bits long but less than or equal to 96 bits long, this string is right-justified with leading zeros, if necessary, in a two-word operand, and then only the first word is returned as the function result. If the arithmetic expression indicates a string of characters more than 96 bits long, a run-time error occurs.

<remainingchars function>

```
-- REMAININGCHARS -- ( --<pointer expression>-- ) --|
```

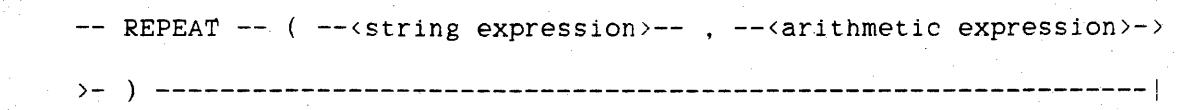
The REMAININGCHARS function returns, as an integer value, the number of characters between the character position referenced by the pointer expression and the end of the array row, including the character at which the pointer expression is pointing. The returned value is in terms of the character size of the pointer expression. If it is a word pointer, the value returned is in terms of 8-bit characters. If the pointer expression points to a paged (segmented) array, the REMAININGCHARS function gives the number of characters left in the entire array row.

For a function that returns the number of characters between the character position referenced by a pointer expression and the beginning of the array row, see the OFFSET function in this section.

<removefile statement>

The REMOVEFILE statement returns a Boolean value. For more information, refer to "REMOVEFILE Statement."

<repeat function>



The REPEAT function returns a string according to the following rules:

1. If <arithmetic expression> = 0, the result of the REPEAT function is the null string.

2. If <arithmetic expression> > 0, the result of the REPEAT function is the same as if <arithmetic expression> occurrences of the value of the string expression were all concatenated together.

3. If <arithmetic expression> < 0, the result of the REPEAT function is either a compile-time or a run-time error.

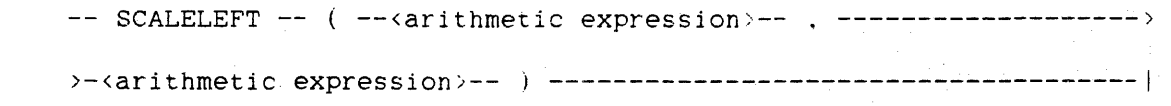
The value of the arithmetic expression is rounded to an integer, if necessary, before it is used as the repeat count.

Examples

REPEAT("ABC",3) = "ABCABCABC"

REPEAT("WON'T WORK",0) = the null string

<scaleleft function>

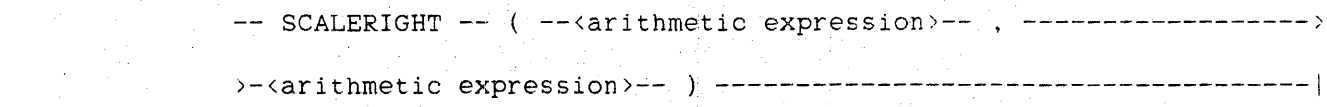


The SCALELEFT function returns, as an integer value,

first <arithmetic expression>*(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. The SCALELEFT function is undefined when the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<scaleright function>

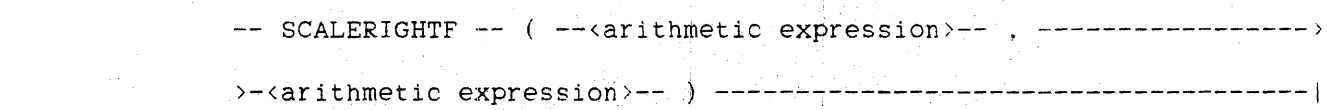


The SCALERIGHT function returns, as an integer value, the rounded result of

first <arithmetic expression>/(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<scalerightf function>



The SCALERIGHTF function returns, as a real value, a left-justified, packed decimal (4-bit decimal) number representing

first <arithmetic expression> MOD (10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

The number of significant digits returned by the function is equal to the integerized value of the second arithmetic expression. The external sign flip-flop (EXTF) is set to reflect the sign of the first arithmetic

expression for use with the editing phrases specified in a PICTURE declaration.

Examples

SCALERIGHTF(1234,4) = 4"123400000000"

SCALERIGHTF(12345678,12) = 4"000012345678"

<scalerightt function>

```
-- SCALERIGHTT -- ( --<arithmetic expression>-- , ----------------->

>-<arithmetic expression>-- ) ------------------------------------|
```

The SCALERIGHTT function returns, as an integer value, the truncated result of

first <arithmetic expression>/(10 ** second <arithmetic expression>)

where the second arithmetic expression, rounded to an integer, has a value in the range 0 to 12. A run-time error occurs if the integerized value of the second arithmetic expression is less than 0 or greater than 12.

<secondword function>

```
-- SECONDWORD -- ( --<arithmetic expression>-- ) --|
```

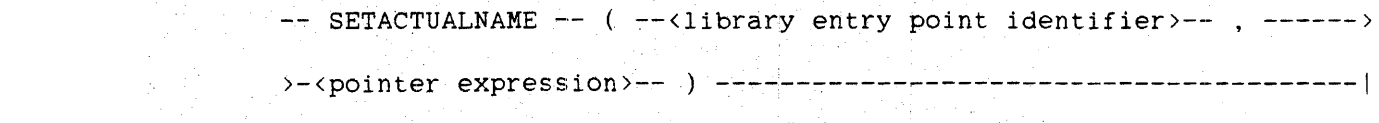
The SECONDWORD function returns, as a real value, the second word of the double-precision arithmetic expression. The arithmetic expression is first extended to double precision, if necessary.

For a function that returns the first word of a double-precision arithmetic expression, see the FIRSTWORD function in this section.

<seek statement>

The SEEK statement returns a Boolean value. For more information, refer to "SEEK Statement."

<setactualname function>



<library entry point identifier>

A <procedure identifier> declared with a <library entry point specification>.

See also
<library entry point specification> 169
<procedure identifier>. 165

The SETACTUALNAME function determines whether or not the ACTUALNAME of the library entry point specified by the library entry point identifier can be changed to the name pointed to by the pointer expression. The function then makes the change, if it is possible. Results of a successful change are returned upon completing the ACTUALNAME change; otherwise, results indicating the reason for failure are returned.

The ACTUALNAME of an entry point of a linked library cannot be modified. Therefore, a linked library must be delinked before calling the SETACTUALNAME function to change the ACTUALNAME of any of its entry points. The function can be called to modify an entry point of a library that has not yet been linked.

Starting with the first character pointed to by the pointer expression, characters are included as the new entry point name until either a period is encountered, the maximum allowable number of characters is included, or the end of the array row is encountered. The last case results in an error condition.

The SETACTUALNAME function returns the following integer values:

Value	Meaning
1	A successful change was made to the ACTUALNAME of the entry point.
0	The new ACTUALNAME is the same as the current ACTUALNAME of the entry point.
-1	The library is linked. The library must be delinked before the SETACTUALNAME function is called.
-2	The library entry point identifier is not currently in an accessible library template.
-3	A parameter error occurred in the SETACTUALNAME function.

Example

```
REPLACE NEWEPNAME BY "ENTRYPOINT2.";
SETACTUALNAME(EP2,NEWEPNAME);
```

Changes the ACTUALNAME of the library entry point EP2 to "ENTRYPOINT2", if possible.

<sign function>

The SIGN function returns an integer 1 if the value of the arithmetic expression is greater than zero, an integer 0 if the value of the arithmetic expression is equal to zero, and an integer -1 if the value of the arithmetic expression is less than zero.

<sin function>

```
-- SIN -- ( --<arithmetic expression>-- ) --|
```

The SIN function returns, as a real value, the sine of an angle of <arithmetic expression> radians.

<single function>

```
-- SINGLE -- ( --<arithmetic expression>-- ) --|
```

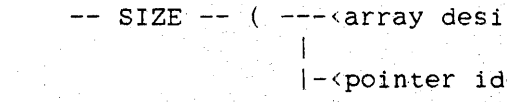
The SINGLE function returns, as a real value, the value of the arithmetic expression normalized and truncated to single precision.

<sinh function>

```
-- SINH -- ( --<arithmetic expression>-- ) --|
```

The SINH function returns, as a real value, the hyperbolic sine of the specified arithmetic expression.

<size function>



See also
<array designator>. 43
<pointer identifier>. 160

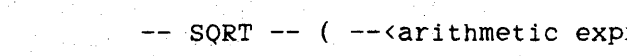
The "SIZE(<array designator>)" form of the SIZE function returns, as an integer value, the size of one dimension of the specified array in elements. If the array designator is an array name, the SIZE function returns the size of the first dimension of the specified array. If the array designator contains a subarray selector, the SIZE function returns the size of the dimension that corresponds to the first asterisk (*) subscript.

The "SIZE(<pointer identifier>)" form of the SIZE function returns, as an integer value, the character size of the specified pointer. If the character size of the pointer is four, six, or eight bits, the value returned is 4, 6, or 8, respectively. If the pointer is word-oriented, the value returned is 0 or 2, depending on whether the pointer is single precision or double precision, respectively. If the pointer is uninitialized, the SIZE function returns zero. See the POINTER function in this section for a discussion of character size.

<space statement>

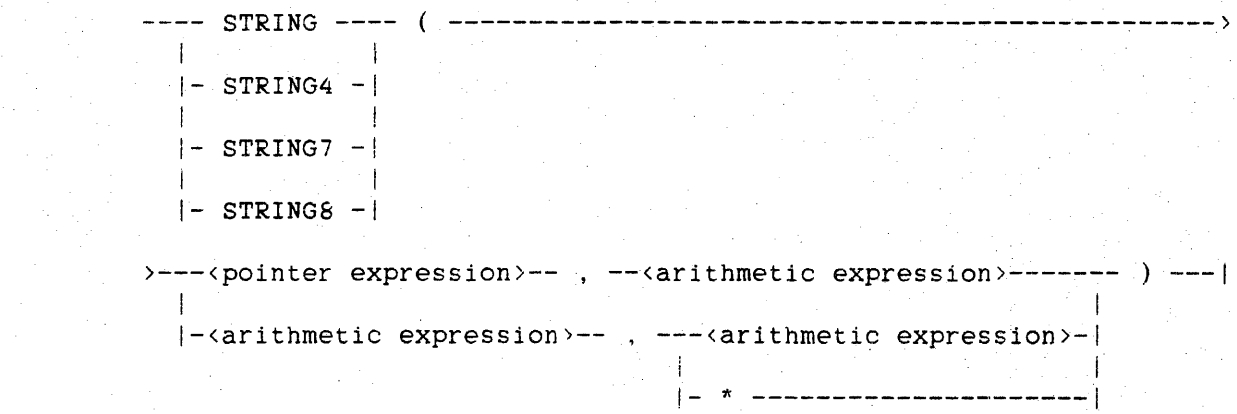
The SPACE statement returns a Boolean value. For more information, refer to "SPACE Statement."

<sqrt function>



The SQRT function returns, as a real value, the square root of the arithmetic expression. The value of the arithmetic expression must be greater than or equal to zero.

<string function>



The STRING function returns a string value. The function STRING4 returns a hexadecimal string, the function STRING7 returns an ASCII string, and the function STRING8 returns an EBCDIC string. The function STRING returns a string of the default character type. (For more

information on the default character type, see "Default Character Type" in the appendix "Data Representation.")

See also
Default Character Type. 817

If the first parameter to the STRING function is a pointer expression, the STRING function converts the string of characters pointed to by the pointer expression into a string. The number of characters converted is given by the value of the arithmetic expression, rounded to an integer, if necessary. If this rounded value is less than zero, a compile-time or run-time error occurs. If the rounded value is zero, the null string is returned.

If the first parameter to the STRING function is an arithmetic expression, the STRING function returns a string consisting of a decimal representation of the value of that arithmetic expression. If the second parameter is also an arithmetic expression, then the value of this expression, rounded to an integer, specifies the length of the resulting string. If this value is less than zero, a compile-time or run-time error occurs. If this value is equal to zero, the null string is returned. If the second parameter is an asterisk (*), the resulting string is exactly long enough to represent the value of the arithmetic expression with no blanks. If the value of the first arithmetic expression is zero and the second parameter is an asterisk, the resulting string is one character long.

When the STRING function is to return an ASCII or EBCDIC string and the first parameter is an arithmetic expression, its value is converted into the most efficient form, depending on the length specified by the second parameter. If both parameters are arithmetic expressions and the rounded value of the second parameter is greater than the minimum number of characters needed to represent the first parameter, leading blanks are inserted in the resulting string. If both parameters are arithmetic expressions and the rounded value of the second parameter is less than the number of characters needed to represent the first parameter, then a string of all asterisks is returned.

For the function STRING4, when the first parameter is an arithmetic expression, only the integer portion of the value of this expression is converted. If both parameters to STRING4 are arithmetic expressions and the rounded value N of the second parameter is less than the number of characters needed to represent the first parameter, then the rightmost N digits of the converted value of the first parameter are returned. If both parameters to STRING4 are arithmetic expressions and the rounded value of the second parameter is greater than the number of characters needed to represent the converted value of the first parameter, then leading zeros are inserted into the resulting string.

Examples

```
STRING(P,20)

STRING(POINTER(A),N-3)

STRING(256,*) = 8"256"

STRING(-335.25,8) = 8" -335.25"

STRING(4.78@-2,5) = 8".0478"

STRING(555000,1) = 8"*"

STRING(456.789,7) = 8"456.789"

STRING4(123.456,8) = 4"00000123"

STRING4(12345678,4) = 4"5678"
```

<tail function>

```
    -- TAIL -- ( --<string expression>-- , --<string character set>---->

    >- ) ------------------------------------------------------------|
```

See also
    <string character set>. 556

The TAIL function returns a string whose value consists of the rightmost characters of the string expression beginning with the first character that is not a member of the string character set. If all characters in the string expression are members of the string character set, the null string is returned. If the first character of the string expression is not a member of the string character set. the entire string expression is returned.

The string character set must be of the same character type as the string expression. For an explanation of the string character set, see the HEAD function in this section.

The TAIL function and the HEAD function are complementary functions. This means that for any string expression S and any string character set C, the following relation is always TRUE:

```
S = HEAD(S,C) CAT TAIL(S,C)
```

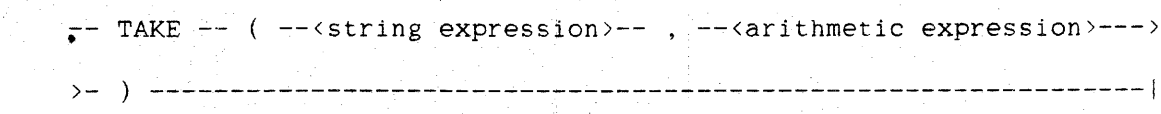
See the HEAD function in this section.

Examples

In the examples below, S is a string of length 9 that contains 8"ABC/1-2+3".

```
TAIL(S,NOT "-") = 8"-2+3"

TAIL(DROP(S,7),"+-") = 8"3"
```

<take function>



The TAKE function returns a string whose value is equal to the first <arithmetic expression> characters taken from the value of the string expression. The value of the arithmetic expression is rounded to an integer, if necessary. An error occurs if the rounded value of the arithmetic expression is greater than the number of characters in the string expression or less than zero. If the rounded value of the arithmetic expression is zero, the result is the null string. If the rounded value of the arithmetic expression is equal to the length of the string expression, the result is the same as the value of the string expression.

The TAKE function and the DROP function are complementary functions. This means that for any string expression S and any arithmetic expression A in the range 0 <= A <= LENGTH(S), the following relation is always true:

```
S = TAKE(S,A) CAT DROP(S,A)
```

See the DROP function in this section.

Examples

In the examples below, string S has a length of 6 and contains 8"ABCDEF".

TAKE(S,2) = 8"AB"

TAKE(S,4) = 8"ABCD"

TAKE(DROP(S,2),2) = 8"CD"

<tan function>

```
-- TAN -- ( --<arithmetic expression>-- ) --|
```

The TAN function returns, as a real value, the tangent of an angle of <arithmetic expression> radians.

<tanh function>

```
-- TANH -- ( --<arithmetic expression>-- ) --|
```

The TANH function returns, as a real value, the hyperbolic tangent of the specified arithmetic expression.

<time function>

```
-- TIME -- ( --<arithmetic expression>-- ) --|
```

The TIME function makes various system time values available. The value of the arithmetic expression is rounded to an integer, if necessary, before being used. The results returned for different values of the integerized arithmetic expression are given in the following table. If the integerized value of the arithmetic expression is not one of the values listed in the table, the TIME function returns the value zero.

Parameter	Result Returned

0 TIME(0) returns the current date in BCL characters in the format 6"YYDDD", where YY is the year and DDD is the day of the year.

1 TIME(1) returns the time of day, in sixtieths of a second, as an integer value.

2 TIME(2) returns the elapsed processor time of the program, in sixtieths of a second, as an integer value.

3 TIME(3) returns the elapsed I/O time of the program, in sixtieths of a second, as an integer value.

4 TIME(4) returns the value of a 6-bit clock that increments 60 times per second.

5 TIME(5) returns the month, day, and year as six BCL characters, right-justified, in the format 6"00MMDDYY".

6 TIME(6) returns a unique number for the time and date (a timestamp) in the following form:

```
0 & (JULIANDATE-70000) [47:16] & (TIME(11) DIV 16) [31:32]
```

7 TIME(7) returns the current date and time in the following form:

```
[47:12]  Year (1900-1999)
[35:06]  Month (1-12)
[29:06]  Day (1-31)
[23:06]  Hour (0-23)
[17:06]  Minute (0-59)
[11:06]  Second (0-59)
[05:06]  Day of the week
         (0 = Sunday, 1 = Monday, ..., 6 = Saturday)
```

9 TIME(9) returns the current time in EBCDIC characters in the format 8"HHMMSS", where HH is the hour, MM is the minute, and SS is the second.

10 TIME(10) returns the same value as TIME(0), except that the time is expressed in EBCDIC characters in the format 8"YYDDD".

11 TIME(11) returns the same value as TIME(1), except that the time is expressed in multiples of 2.4 microseconds.

12 TIME(12) returns the same value as TIME(2), except that the time is expressed in multiples of 2.4 microseconds.

Parameter	Result Returned

13 TIME(13) returns the same value as TIME(3), except that the time is expressed in multiples of 2.4 microseconds.

14 TIME(14) returns the time elapsed since the last Halt/Load in multiples of 2.4 microseconds.

15 TIME(15) returns the current date in EBCDIC characters in the format 8"MMDDYY".

16 TIME(16) returns the same value as TIME(6).

23 TIME(23) returns the system identification information in the following format:

```
[31:08]  ASD System flag.
           0 - Non-ASD system
           1 - Non-ASD system
           2 - ASD system
[23:16]  System serial number.
[07:08]  Type of machine DIV 100
           (for example, 68 means B 6800).
           If the system type is not of the form
           B XX00, this field contains zero.
```

24 TIME(24) returns the system type as a real value containing six EBCDIC characters. The name is left-justified with blank fill. For example, on a B 5920 or B 5930 system, TIME(24) returns 8"B5900 ".

<translate function>

```
-- TRANSLATE -- ( --<string expression>-- , --<translate table>---->

>- ) ------------------------------------------------------------|
```

See also
<translate table> 382

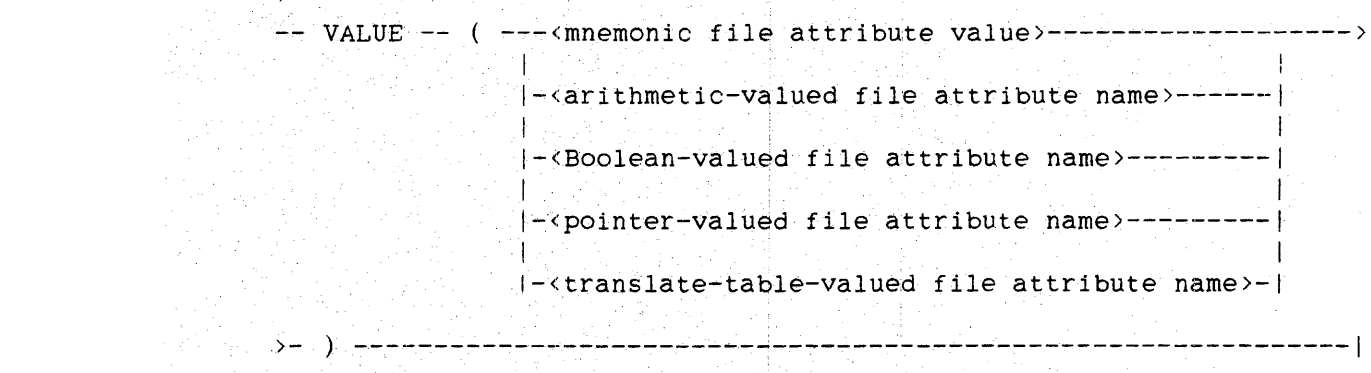
The result of the TRANSLATE function is a string of the same length as the string expression with each character of the string expression translated according to the translate table.

The translate table used in the TRANSLATE function must not be composed of items of different character types. The translate table can be a translate table identifier declared in the program or one of the intrinsic translate tables. The use of a subscripted variable as a translate table is not allowed in the TRANSLATE function.

Examples

```
TRANSLATE(S,HEXTOEBCDIC)

TRANSLATE(TAKE(S,10),MYTT)
```

<value function>



See also

<arithmetic-valued file attribute name> 86
<Boolean-valued file attribute name>. 86
<mnemonic file attribute value> 86
<pointer-valued file attribute name>. 86
<translate-table-valued file attribute name>. 86

If a mnemonic file attribute value is specified, the VALUE function returns the integer value that corresponds to that mnemonic file attribute value. If a file attribute name is specified, the VALUE function returns the attribute number that corresponds to that attribute name. For more information on file attribute names and mnemonic file attribute values, refer to the "I/O Subsystem Reference Manual."

Examples

```
F.KIND := VALUE(DISK)

F.INTMODE := VALUE(EBCDIC)
```

<wait statement>

Depending on the form used, the WAIT statement returns no value, a Boolean value, or an integer value. For more information, refer to "WAIT Statement."

<waitandreset statement>

The WAITANDRESET statement returns an integer value. For more information, refer to "WAITANDRESET Statement."

<write statement>

The WRITE statement returns a Boolean value. For more information, refer to "WRITE Statement."

7 COMPILING PROGRAMS

This chapter presents information outside of the ALGOL language. This information is necessary to compile and run an ALGOL program. The chapter describes the files used by the ALGOL compiler when it compiles a program, the format of a source record, and compiler control options.

7.1 FILES USED BY THE COMPILER

When the ALGOL compiler compiles a program, it requires, at minimum, one input file that contains the source code to be compiled. If the compile is successful, the compiler produces, at minimum, one output file that contains executable object code.

Through the use of compiler control options, the compiler can be directed to use additional input files and to produce additional output files. Among the optional input files that can be used are files containing source code that is to be merged or inserted into the required source code file, and files containing information needed to perform separate compilation and binding. Among the optional output files that can be produced are a printer listing of the program, a cross-reference file, an updated source file, an error message file, and a file containing information needed for future separate compilation.

Table 7-1 lists the logical input and output files used by the ALGOL compiler. Each file is listed with values for the INTNAME, KIND, INTMODE, MAXRECSIZE, BLOCKSIZE, and FILETYPE attributes. Some or all of the attributes for these files can be changed, using compiler file equation, when the compiler is initiated. (For more information, refer to the "I/O Subsystem Reference Manual" and the "Work Flow Language (WFL) Reference Manual" for detailed information about file attributes and compiler file equation, respectively.)

Table 7-1. Compiler Files

Compiler Input Files

Description	INTNAME	Initiation	KIND	INTMODE	MAXRECSIZE	BLOCKSIZE	FILETYPE
PRIMARY INPUT FILE	CARD	WFL	READER	EBCDIC	Taken from physical file	Taken from physical file	8
		CANDE	DISK				
OPTIONAL SECONDARY INPUT FILE	TAPE	WFL and CANDE	DISK	EBCDIC	Taken from physical file	Taken from physical file	8
OPTIONAL SOURCE FILES INPUT BY $ INCLUDE COMPILER CONTROL OPTIONS	(INCLUDEd files)	WFL and CANDE	DISK	EBCDIC	Taken from physical file	Taken from physical file	8
OPTIONAL OBJECT FILE INPUT	HOST	WFL and CANDE	DISK	SINGLE	30 words	270 words	8
OPTIONAL INFO FILE	INFO	WFL and CANDE	DISK	SINGLE	Taken from physical file	Taken from physical file	0

Compiler Output Files

OPTIONAL PROGRAM EXECUTION	OBJECT CODE FILE	CODE	WFL and CANDE	DISK	SINGLE	30 words	270 words	—
	OPTIONAL UPDATED SYMBOLIC FILE	NEWTAPE	WFL and CANDE	DISK	EBCDIC	15 words	420 words	—
	OPTIONAL LINE PRINTER LISTING	LINE	WFL and CANDE	PRINTER	EBCDIC	22 words	22 words	—
	OPTIONAL ERROR MESSAGE FILE	ERRORFILE	WFL	PRINTER	EBCDIC	12 words	12 words	
			CANDE	REMOTE				
	OPTIONAL CROSS REFERENCE FILE	XREFFILE	WFL and CANDE	DISK	EBCDIC	510 words	510 words	0
	OPTIONAL INFO FILE	INFO	WFL and CANDE	DISK	SINGLE	256 words	2560 words	0

INPUT FILES

The input files used by the compiler consist of the following:

- CARD, which is the required source code file
- TAPE, which is a source code file that can be merged with CARD
- INCLUDE files, which are source code files that can be inserted into CARD or TAPE
- HOST, which contains information used for separate compilation and binding
- INFO, which contains information used for separate compilation

The EXTMODE attribute (the character type of the physical file) of these input files can be EBCDIC, ASCII, or BCL. The MAXRECSIZE attribute of these input files must be large enough to accommodate at least 72 characters. Because the values of the MAXRECSIZE and BLOCKSIZE attributes for these files are taken from the physical file (when FILETYPE = 8), these two attributes do not require explicit assignment.

CARD File

The CARD file supplies the primary source input to the compiler and must be present for each compilation. If the compiler is initiated from the Work Flow Language (WFL) and compiler file equation is not applied to the CARD file, the file is assumed to be a card reader file. If the compiler is initiated through CANDE and compiler file equation is not applied to the CARD file, the file is assumed to be a disk file.

TAPE File

The TAPE file supplies secondary source input to the compiler. Its presence is optional. If compiler file equation is not applied to the TAPE file, the file is assumed to be a disk file regardless of whether the compiler is initiated from WFL or CANDE.

When this file is present and the MERGE option is TRUE, records from the TAPE file are merged with those of the CARD file on the basis of sequence numbers. If a record from the CARD file and a record from the TAPE file have the same sequence number, the record from the CARD file is compiled and the TAPE record is ignored. For more information on the MERGE option, see "<merge option>" in this chapter.

INCLUDE Files

INCLUDE files provide source input to the compiler in addition to that supplied by the CARD and TAPE files. An INCLUDE file is used only if an INCLUDE compiler control option appears in the source input being compiled. For more information on the INCLUDE option, see "<include option>" in this chapter.

HOST File

The HOST file provides the compiler with information that allows it to separately compile and bind to a host program only procedures that are being changed. This process is called a "sepcomp" and the HOST file is used if the SEPCOMP compiler control option is TRUE. The HOST file is a special object code file, created by a previous compile when the MAKEHOST compiler control option was TRUE. The HOST file contains information about the outer block environment of the program and the environments of selected procedures. For more information on the SEPCOMP and MAKEHOST options, see "<sepcomp option>" and "<makehost option>," respectively, in this chapter.

INFO File

The INFO file contains the contents of variables and tables used in the compiler, saved from a previous compile. This file is used for separate compilation of procedures. It is created by the DUMPINFO compiler control option and is used as an input file by the LOADINFO compiler control option. For more information on the DUMPINFO and LOADINFO options, see "<dumpinfo option>" and "<loadinfo option>," respectively, in this chapter.

See also

OUTPUT FILES

The output files produced by the compiler consist of the following:

- CODE, which is the object code file
- NEWTAPE, which is the updated source code file
- LINE, which is the printer listing of the program
- ERRORFILE, which is the error message file
- XREFFILE, which contains cross-reference information
- INFO, which contains information used for separate compilation

CODE File

The CODE file is produced unconditionally and contains the executable object code produced by the compiler. This file is either stored permanently, executed and then discarded, or discarded, depending on the specifications in the WFL or CANDE "COMPILE" statement that initiated the compiler. and on whether or not syntax errors occurred during the compilation. For more information on the WFL and CANDE "COMPILE" statements, refer to the "Work Flow Language (WFL) Reference Manual" and the "CANDE Reference Manual."

NEWTAPE File

The NEWTAPE file is produced only if the NEW compiler control option is TRUE. The NEWTAPE file is an updated source file that consists of the source input from the CARD file, the TAPE file, and (if the INCLNEW compiler control option is TRUE) the included files that was actually compiled. If compiler file equation is not applied to the NEWTAPE file, it is a disk file. For more information on the NEW and INCLNEW options, see "<new option>" and "<inclnew option>," respectively, in this chapter.

LINE File

The LINE file is produced if either of the compiler control options LIST or TIME is TRUE. If compiler file equation is not applied to the LINE file, it is a printer file.

The contents and format of the LINE file depend on the values of the following compiler control options:

CODE	PAGE
FORMAT	SEGS
LISTDELETED	SINGLE
LISTINCL	STACK
LISTOMITTED	TIME
LISTP	

The LINE file always contains at least the following information:

1. The title of the source file CARD used as input to the compiler
2. Code segmentation information
3. Error messages and error count, if syntax errors occur
4. Summary information about the compile, such as the number of lines read and the size of the object code file

For more information about the compiler control options that affect the LINE file, see "Option Descriptions" in this chapter.

ERRORFILE File

The ERRORFILE file is produced only if the ERRLIST compiler control option is TRUE. If compiler file equation is not applied to the ERRORFILE file, it is a printer file if the compile was initiated through WFL and a remote file if the compile was initiated through CANDE. If no syntax errors occur during compilation, no ERRORFILE file is produced, regardless of the value of the ERRLIST option. For more information on the ERRLIST option, see "<errlist option>" in this chapter.

For every syntax error that occurs during compilation, two records are written to the ERRORFILE file. The first record is a copy of the source record that contains the error. The second record contains the sequence number of that source record (if the source record was sequenced) and the error message.

XREFFILE File

When either of the compiler control options XREF or XREFFILES is TRUE, the compiler saves raw cross-reference information in the XREFFILE file. The contents of the file are affected by the compiler control options XDECS and XREFS. Before this information can be printed or read by SYSTEM/INTERACTIVEXREF or the Editor, it must be analyzed by SYSTEM/XREFANALYZER. The XREFFILE file is given the file name "XREF/<code file name>", where <code file name> is the file name of the object code file being produced. For more information about the cross-reference options, see "Option Descriptions" in this chapter.

INFO File

The INFO file contains the contents of variables and tables used in the compiler. It is intended for use in separate compilations of procedures. This file is created by the DUMPINFO compiler control option, and it is used as an input file by the LOADINFO compiler control option.

See also

<errlist option> 612
<inclnew option> 614
<new option> 629
Option Descriptions 603

7.2 SOURCE RECORD FORMAT

The records of a source code file read by the ALGOL compiler can be any size greater than or equal to 72 characters.

Assume that the character positions (called columns) of a source record are numbered from 1 to n, where n is the size of the record in characters. The compiler divides each source record as follows:

1. The data in columns 1 through 72 is assumed to be ALGOL source language as defined in this manual. Any characters that appear beyond column 72 are not compiled as source language constructs.

2. Characters in columns 73 through 80 of a record are treated as a sequence number, which is optional.

3. Any information beyond column 80 is ignored.

The column in which an ALGOL construct begins is not significant, unless the source record is a compiler control record or the construct continues beyond column 72. (For more information on compiler control records, refer to "Compiler Control Records" in this chapter.) The data in columns 1 through 72 is treated as a continuous stream from record to record. In other words, no delimiters are implied at the end of a record, and string literals, identifiers, and all other valid ALGOL constructs can be continued from column 72 of one record to column 1 of the succeeding record.

7.3 COMPILER CONTROL OPTIONS

Compiler control options provide the programmer with the means to control many aspects of the compilation of an ALGOL program. These options can instruct the compiler to use optional input files or to produce optional output files. In addition to other things, the options can do all of the following:

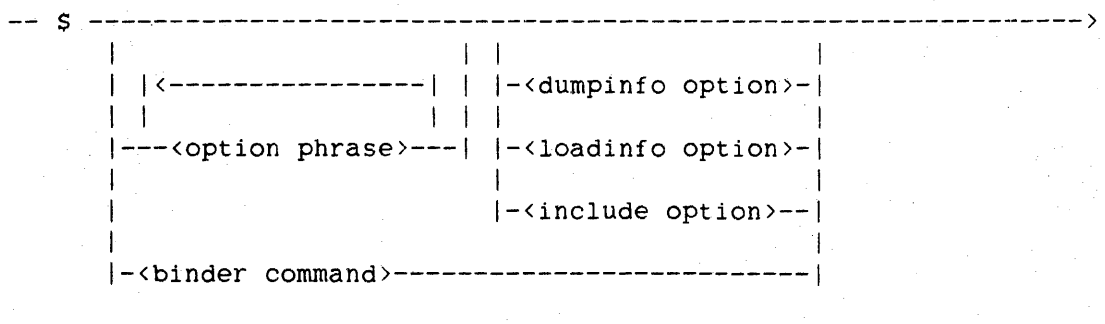
- Affect the contents of the printer listing
- Designate the target computer for which code is to be produced
- Set the default character type
- Cause the compiler to perform separate compilation
- Invoke the Binder
- Invoke debugging features
- Exclude source records
- Set a limit on the number of syntax errors the program can get
- Resequence source records
- Check that sequence numbers are in order
- Control code file segmentation
- Flag BCL constructs

COMPILER CONTROL RECORDS

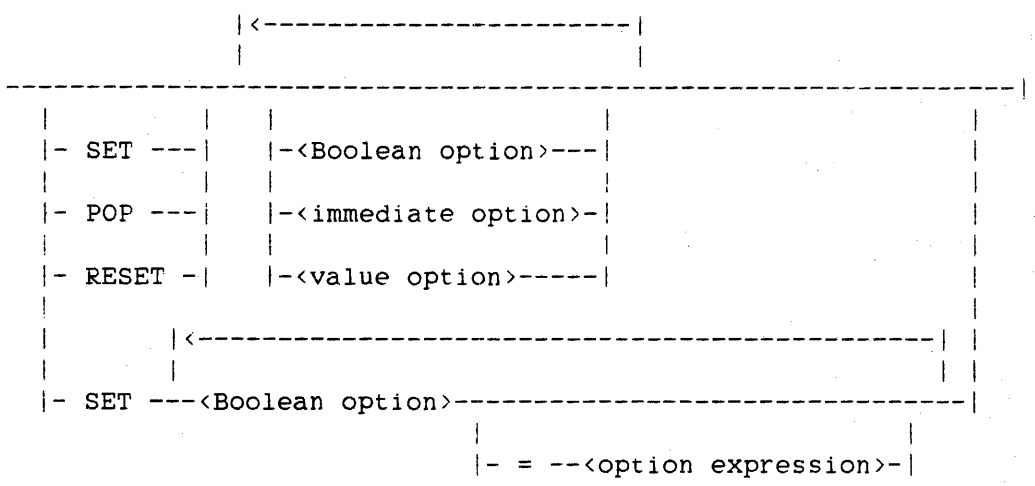
Compiler control options are included in an ALGOL program by using special source records called compiler control records.

Syntax

<compiler control record>



<option phrase>

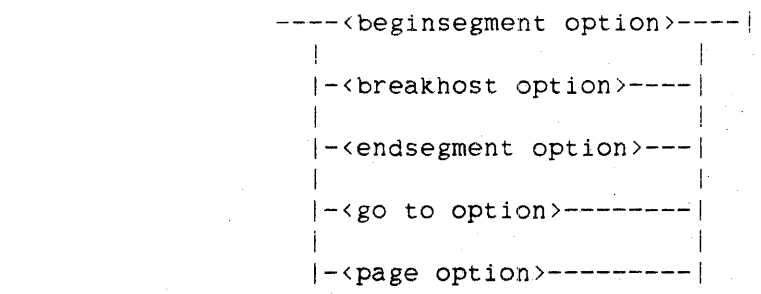


<Boolean option>

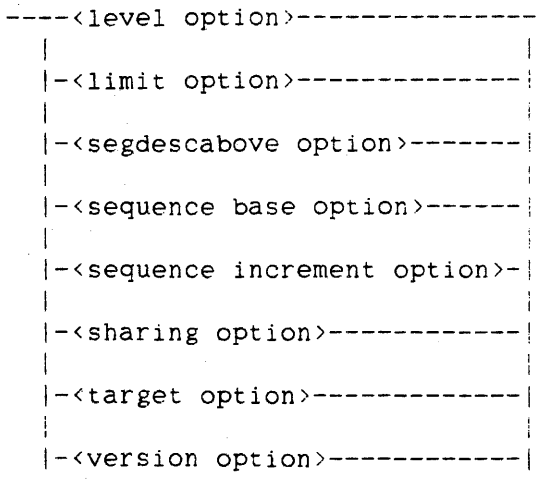
A <Boolean option> can be any of the following:

<$ option>
<ASCII option>
<autobind option>
<BCL option>
<breakpoint option>
<B7700 option>
<check option>
<code option>
<errlist option>
<format option>
<inclnew option>
<inclseq option>
<installation option>
<intrinsics option>
<library option>
<lineinfo option>
<list option>
<listdeleted option>
<listincl option>
<listomitted option>
<listp option>
<makehost option>
<MCP option>
<merge option>
<new option>
<newseqerr option>
<noBCL option>
<nobindinfo option>
<nostackarrays option>
<noxreflist option>
<oldresize option>
<omit option>
<optimize option>
<segs option>
<sepcomp option>
<seq option>
<seqerr option>
<single option>
<stack option>
<statistics option>
<TADS option>
<time option>
<user option>
<void option>
<voidt option>
<warnsupr option>
<writeafter option>
<xdecs option>
<xref option>
<xreffiles option>
<xrefs option>

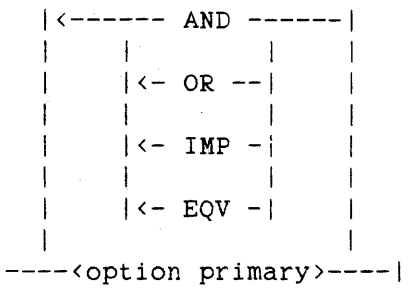
<immediate option>



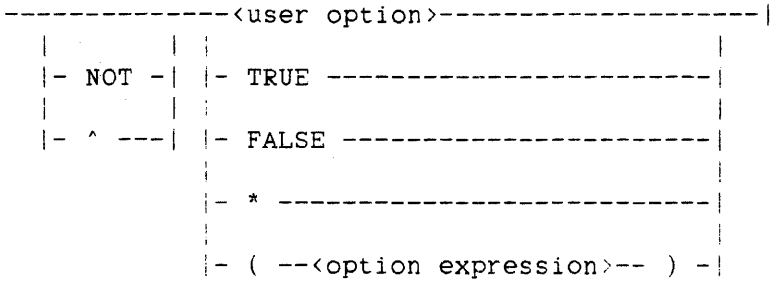
<value option>



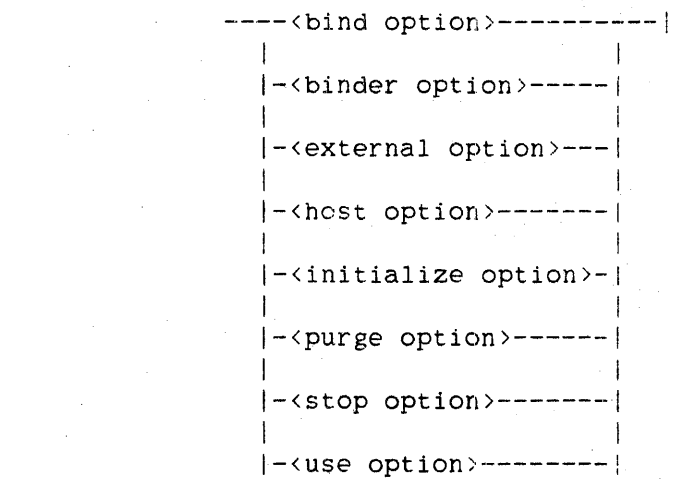
<option expression>



<option primary>



<binder command>



Semantics

Compiler control records are submitted to the ALGOL compiler as part of the source input and are distinguished from other constructs by the dollar sign ($) that must begin every compiler control record.

In a compiler control record, the column in which the initial $ occurs is significant. Compiler control records with the initial $ in column 1 affect only the current compilation and are not saved in the NEWTAPE file, if any. Compiler control records with the initial $ in column 2 are considered permanent compiler control records and are saved in the NEWTAPE file, if any. Compiler control records with the initial $ in columns 3 through 72, inclusive, are considered both permanent and conditional, in that they are saved in the NEWTAPE file, if any, and they are ignored when the OMIT option is TRUE.

A compiler control record can contain the following:

1. Boolean options
2. Value options
3. Immediate options
4. Special options
5. Binder commands

A Boolean option is one that is either enabled (TRUE) or disabled (FALSE). When enabled, it causes the compiler to apply an associated function to all subsequent processing until the option is disabled.

A value option causes the compiler to store a value associated with a given function.

An immediate option causes the compiler to perform immediately a function that is not applied to subsequent processing.

The special options (the DUMPINFO option, the LOADINFO option, and the INCLUDE option), like the immediate options, cause the compiler to perform immediately a function that is not applied to subsequent processing. However, they are not grouped with the immediate options because of special syntactic requirements.

Binder commands are passed directly to the Binder program. For more information on Binder statements and control options, refer to the "Binder Reference Manual."

The keywords SET, RESET, and POP affect the value of Boolean options. Each Boolean option has an associated "stack" in which the current value and up to 46 previous values of the option are saved. The management of this stack of values is described below:

1. If the first keyword to the left of a Boolean option in a compiler control record is SET or RESET, the current value of the option is pushed onto the stack and the option is assigned a value of TRUE or FALSE, respectively. In other words, the option is assigned a new value, and the previous value is saved in the stack.

2. If the first keyword to the left of a Boolean option in a compiler control record is POP, the current value of the option is discarded and the previous value is removed from the top of the stack and assigned to the option.

3. If a Boolean option is not preceded in a compiler control record by any keyword, then the following actions occur:

    a. All resettable standard Boolean options are assigned a value of FALSE and all previous values in their stacks are discarded.

    b. The Boolean options appearing in the compiler control record with no preceding keyword are assigned a value of TRUE.

The following is a list of the resettable standard Boolean options that are affected when a Boolean option appears in a compiler control record without a preceding keyword:

ASCII
AUTOBIND (if SEPCOMP is FALSE)
BCL
CHECK
CODE
FORMAT
INCLNEW
INCLSEQ
INSTALLATION
INTRINSICS
LIST
LISTDELETED
LISTINCL
LISTP
MAKEHOST (if the first syntactic item has not been seen)
NEW
NEWSEQERR
NOXREFLIST
OMIT
OPTIMIZE
SEGS
SEQ
SEQERR
STACK
STATISTICS
TIME
VOID
VOIDT
WARNSUPR
WRITEAFTER
XREF
XREFFILES
XREFS
$

A compiler control record that consists of only an initial $ (a $ followed by all blanks) has no effect unless it is in the CARD file, the MERGE option is TRUE, and a record is present in the secondary input file TAPE that has the same sequence number as this compiler control record. When these three conditions are met, then that TAPE record is ignored.

In an ‹option expression›, the Boolean operators have the same precedence as they do in Boolean expressions; that is, NOT and ^ are equivalent and have the highest precedence, followed by AND, OR, IMP, and EQV, in that order. The ‹option primary› "*" represents the current value of the Boolean option whose value is being assigned. For example, the compiler control record

```
$ SET OMIT = * OR DEBUGCODE
```

causes records from both the CARD and TAPE files to be ignored if either the OMIT option is TRUE or the user option DEBUGCODE is TRUE.

Examples

```
$ SET LIST MERGE NEW RESET SINGLE
```

This compiler control record assigns a value of TRUE to the LIST option, MERGE option, and NEW option, and assigns a value of FALSE to the SINGLE option. The previous value of each of these options is saved in the corresponding stack.

```
$ PAGE
```

This compiler control record invokes the immediate option PAGE, causing the compiler to skip the printer listing to the top of the next page (if the LIST option is TRUE). Because the PAGE option is not a Boolean option, the action described above for Boolean options not preceded by a keyword does not occur.

```
$ SET LISTP 135000 POP LIST
```

This compiler control record assigns a value of TRUE to the LISTP option, assigns the value 135000 to the sequence base option, and returns the LIST option to its most recent previous value.

```
$ LISTP POP MYOPTION
```

This compiler control record assigns FALSE to all resettable standard Boolean options and discards all previous values in their stacks, assigns the value TRUE to the LISTP option, and returns the USER option MYOPTION to its most recent previous value. The stack for MYOPTION is unaffected by the purge of the stacks for the standard resettable Boolean options, because it is not a standard option.

OPTION DESCRIPTIONS

The rest of this chapter describes the individual compiler control options.

<ASCII option>

```
-- ASCII --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the ASCII option sets the default character type to ASCII. For more information, refer to "Default Character Type" in the appendix "Data Representation."

If the BCL option is TRUE and the ASCII option is assigned the value TRUE, a syntax error is given.

See also
Default Character Type. 817

<autobind option>

```
-- AUTOBIND --|
```

(Type: Boolean, Default value: FALSE)

If TRUE, the AUTOBIND option causes the processes of compilation and program binding to be combined into one job. During compilation, the compiler produces a set of instructions to be passed to the Binder program. In most cases, these Binder instructions are sufficient. If additional Binder instructions are required, the ALGOL <binder command> syntax can be used.

The AUTOBIND option can be assigned a value at any time during compilation. However, for the following reasons, it should be assigned a value only once, at the beginning of compilation:

1. The value of the AUTOBIND option is significant only at the end of compilation. For example, if four procedures are being compiled, the first three with the AUTOBIND option FALSE and the last with the AUTOBIND option TRUE, the Binder attempts to bind all four procedures to the specified host.

2. When the AUTOBIND option is FALSE, compile-and-go on a separate procedure is not executed. If the AUTOBIND option is TRUE throughout compilation, execution of the resulting program takes place after binding.

In ALGOL, an outer block or a separate procedure compiled at lexical (lex) level two can serve as a "host" for binding. Separate ALGOL procedures compiled at lex level three (the default level) or higher can be bound into a host. Any number of separate procedures, but only one host, can be compiled in one job. The host must be the last program unit compiled. If an appropriate host file is compiled with the AUTOBIND option equal to TRUE, it is assumed to be the host for binding. (This assumption cannot be overridden by file equation or by use of the HOST option.) If no eligible host is being compiled, a host must be specified, either by file equation of the compiler file HOST or by use of the HOST option.

The object code file of any procedure compiled at lex level three or higher with the AUTOBIND option equal to TRUE is marked as nonexecutable. To be executed, the procedure must be bound into a host file by the Binder program or invoked by a PROCESS or CALL statement.

Object code files of any procedures compiled at lex level three or higher are purged after being bound into a host by the AUTOBIND option. To be retained, such a code file must be referenced specifically in either a BIND option or an EXTERNAL option.

When the batch facility is used, the AUTOBIND option cannot be assigned a value. If the TADS option is TRUE when the AUTOBIND option is assigned the value TRUE, then the AUTOBIND option is left equal to FALSE and a warning message is given.

<BCL option>

```
-- BCL --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the BCL option sets the default character type to BCL. For more information, refer to "Default Character Type" in the appendix "Data Representation."

If the ASCII option is TRUE when the BCL option is assigned the value TRUE, a syntax error is given.

See also
Default Character Type. 817

NOTE

> The BCL data type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems. The appearance of a BCL construct that may cause the creation of a BCL descriptor, such as the BCL option, will cause the program to get a compile-time warning message.

<beginsegment option>

```
-- BEGINSEGMENT --|
```

(Type: immediate)

The BEGINSEGMENT option and the ENDSEGMENT option allow the programmer to control code file segmentation. Procedures encountered between a BEGINSEGMENT option and an ENDSEGMENT option are placed in the same code segment, which is called a "user segment" because it is user-controlled instead of compiler-controlled.

The BEGINSEGMENT option must appear before the declaration of the first procedure to be included in the user segment. The ENDSEGMENT option must appear after the last source record of the last procedure to be included in the user segment.

A procedure cannot be split across user segments. The first procedure in the user segment must be one for which the compiler would normally generate a segment; that is, it must have local declarations. External procedures cannot be declared in a user segment.

Declarations of global items other than procedures within a user segment can result in those items not being initialized. This could cause the program to get a fault at run time. Declarations of global items other than procedures should be placed outside user segments.

User segments can be nested; that is, a BEGINSEGMENT option can appear in a user segment. In this case, an ENDSEGMENT option applies to the user segment currently being compiled.

If a BEGINSEGMENT option appears before the beginning of a separately compiled procedure, an ENDSEGMENT option is assumed at the end of the procedure, even if none appears. The driver procedure created for procedures compiled at lexical level three is always in a different code segment.

The segment information in the printer listing is modified for user segments. User segments are numbered consecutively in a program, beginning with 1; that is, the first user segment is USERSEGMENT1, the second user segment is USERSEGMENT2, and so forth. The code segment number of each user segment is printed at its beginning; the length of each user segment is printed at its end. Procedures or blocks whose segmentation is overridden by user segmentation are printed out as being "in" that user segment.

Forward procedure declarations are not affected by user segmentation.

If more than one BEGINSEGMENT option appears before a procedure, the warning message "EXTRA BEGINSEGMENT IGNORED" is printed. If an ENDSEGMENT option appears and there has been no BEGINSEGMENT option, the warning message "EXTRA ENDSEGMENT IGNORED" is printed.

The BEGINSEGMENT option and ENDSEGMENT option allow the programmer to reduce presence-bit overhead by grouping frequently called procedures and infrequently called procedures in separate segments.

<bind option>

```
-- BIND --<text>--|
```

See also
<text>. 61

(Type: Binder command)

During autobinding, the BIND option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder BIND statement and are described in the "Binder Reference Manual."

When the batch facility is used, the BIND option cannot be used.

<binder option>

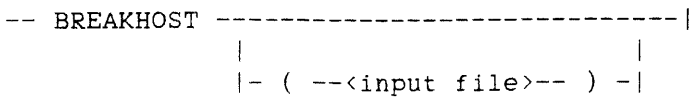
```
-- BINDER --<text>--|
```

See also
<text>. 61

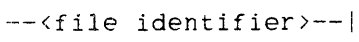
(Type: Binder command)

During autobinding, the <text> in the BINDER option is passed directly to the Binder program for analysis. The format and function of <text> are the same as those of the Binder control options and are described in the "Binder Reference Manual."

<breakhost option>



<input file>



See also
<file identifier> 85

(Type: immediate)

To create the necessary environment for interactive debugging with the breakpoint intrinsic, the BREAKHOST option must appear in the outer block of any program that uses the BREAKPOINT option or the BREAKPOINT statement. This option must appear after the first "BEGIN" of a program but before the first statement of the outer block.

A part of the environment created by the BREAKHOST option is a remote file. If a program to be debugged has a remote input file, the name of that file must be specified as the <input file> to allow the breakpoint intrinsic to use that remote file, because only one remote input file can be open for each station.

Another part of the environment created when the BREAKHOST option is TRUE is a 32,768-element real array. If a program that uses the BREAKHOST option is run with its task attribute OPTION equal to LONG, the run-time error

DIMENSION SIZE ERROR 1=32768

is given if the overlay row size of the system cannot accommodate an array of that size.

The BREAKHOST option should not be used in a library program.

For more information about the breakpoint feature, see "BREAKPOINT Statement" and the description of the BREAKPOINT option is this section.

NOTE

The BREAKHOST option, the BREAKPOINT option, and the BREAKPOINT statement are being deimplemented on the Mark 3.7 release. For a debugging feature, refer to the TADS option in this section.

<breakpoint option>

(Type: Boolean, Default value: FALSE)

While the BREAKPOINT option is TRUE, the code emitted for each ALGOL statement is followed by a call on the breakpoint intrinsic. The execution of the program stops (breaks) after each statement in this range (and at any explicit call of the breakpoint intrinsic) to allow debugging through the use of breakpoint commands.

For more information about the breakpoint feature, see "BREAKPOINT Statement" and the description of the BREAKHOST option in this section.

NOTE

The BREAKHOST option, the BREAKPOINT option, and the BREAKPOINT statement are being deimplemented on the Mark 3.7 release. For a debugging feature, refer to the TADS option.

<B7700 option>

(Type: Boolean, Default value: TRUE)

When TRUE, the B7700 option causes optimized code to be generated for the B 7000 Series systems. The B7700 option cannot be assigned a value if the TARGET option has been used or if the computer on which the program is run is an A Series system.

NOTE

The B7700 option will be deimplemented on the Mark 3.7 release. For information on code optimization for a particular computer, refer to the TARGET option in this section.

<check option>

-- CHECK --|

(Type: Boolean, Default value: FALSE)

When TRUE, the CHECK option causes an error to be given if the sequence number on a record of the TAPE or NEWTAPE file is not strictly greater than the sequence number of the preceding record. If a sequence error occurs in the TAPE file, the word "SEQERR" followed by the sequence number of the previous source record is printed at the right side of the source record on the printer listing. If a sequence error occurs in the NEWTAPE file, the message "NEWTAPE SEQ ERROR" followed by the sequence number of the previous source record is printed on the listing, and the message "NEWTAPE SEQ ERR" is displayed on the Operator Display Terminal (ODT). In the ERRORFILE, the sequence number of the record that caused the sequence error and the sequence number of the previous source record appear on the line following the source record.

If the NEW option is FALSE and resequencing is occurring, the old sequence number is the sequence number that is checked.

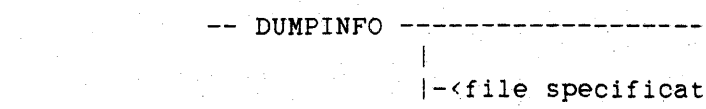
<code option>

-- CODE --|

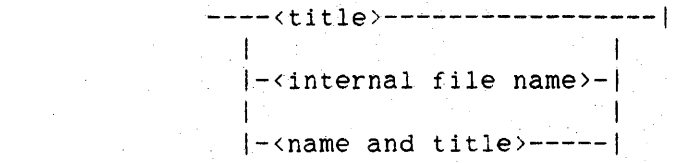
(Type: Boolean, Default value: FALSE)

If both the LIST option and the CODE option are TRUE, the printer listing includes the compiler-generated object code. If the LIST option is TRUE but the CODE option is FALSE, the printer listing does not include the object code. The value of the CODE option is ignored if the LIST option is FALSE.

<dumpinfo option>



<file specification>

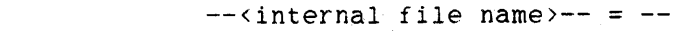


<title>

A quoted string containing a file title.

<internal file name>

<name and title>



(Type: special)

The DUMPINFO option is described with the LOADINFO option in this section.

<endsegment option>

(Type: immediate)

The ENDSEGMENT option is described with the BEGINSEGMENT option in this section.

<errlist option>

```
-- ERRLIST --|
```

(Type: Boolean, Default value: TRUE for CANDE-originated compiles, FALSE otherwise)

When TRUE, the ERRLIST option causes syntax error information to be written to the ERRORFILE file. When a syntax error is detected in the source input, the source record that contains the error, an error message, and the syntactical item where the error occurred are written on two lines in the ERRORFILE file. This option is provided primarily for use when the compiler is invoked at a terminal by CANDE, but it can be used regardless of the manner in which the compiler is invoked. When the compiler is invoked from CANDE, the default value of the ERRLIST option is TRUE, and the ERRORFILE file is automatically equated to the remote device from which the compiler was invoked.

<external option>

```
-- EXTERNAL --<text>--|
```

See also
<text>. 61

(Type: Binder command)

During autobinding, the EXTERNAL option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder EXTERNAL statement and are described in the "Binder Reference Manual."

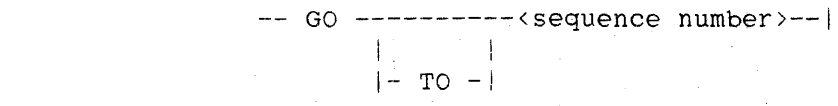
When the batch facility is used, the EXTERNAL option cannot be used.

‹format option›

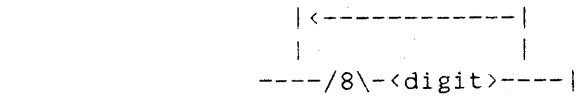
(Type: Boolean, Default value: FALSE)

If both the LIST option and the FORMAT option are TRUE, then to aid readability of the printer listing, several blank lines are inserted after each procedure. If the LIST option is TRUE but the FORMAT option is FALSE, no blank lines are inserted after procedures. If the LIST option is FALSE, the value of the FORMAT option is ignored.

‹go to option›



‹sequence number›



(Type: immediate)

The GO TO option is used to reposition the secondary source input file TAPE. This option is intended for use with disk files and does not work on tape files.

The ‹sequence number› construct specifies a sequence number appearing on a record in the TAPE file. The GO TO option causes the TAPE file to be repositioned so that the next record from this file used by the compiler is the first record with a sequence number greater than or equal to the specified sequence number. The TAPE file must be properly sequenced in ascending order; that is, the sequence number on each record in the file must be strictly greater than the sequence number on the preceding record. The specified sequence number can be greater than or less than the sequence number of the record on which the option appears.

This option cannot appear within a DEFINE declaration or in included source input, and cannot be used when the batch facility is used.

<host option>

```
-- HOST --<text>--|
```

See also
<text>. 61

(Type: Binder command)

During autobinding, the HOST option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder HOST statement and are described in the "Binder Reference Manual."

When the batch facility if used, the HOST option cannot be used.

<inclnew option>

```
-- INCLNEW --|
```

(Type: Boolean, Default value: FALSE)

If both the NEW option and the INCLNEW option are TRUE, included source input is written to the NEWTAPE file. If the NEW option is TRUE but the INCLNEW option is FALSE, included source input is not written to the NEWTAPE file. If the NEW option is FALSE, the value of the INCLNEW option is ignored.

When the batch facility is used, the INCLNEW option cannot be assigned a value.

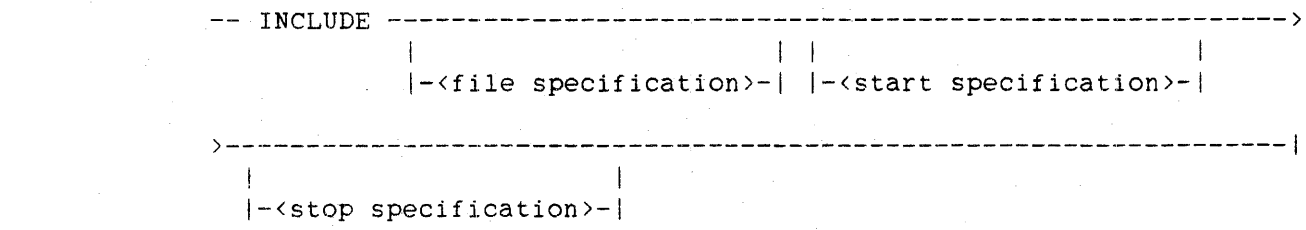
<inclseq option>

(Type: Boolean, Default value: FALSE)

If both the SEQ option and the INCLSEQ option are TRUE, included source input is resequenced. If the SEQ option is TRUE but the INCLSEQ option is FALSE, included source input is not resequenced. If the SEQ option is FALSE, the value of the INCLSEQ option is ignored.

When the batch facility is used, the INCLSEQ option cannot be assigned a value.

<include option>



<start specification>

<stop specification>

See also
<file specification>. 611
<sequence number> 613

(Type: special)

The INCLUDE option causes the compiler to accept source language input from files other than the CARD and TAPE files. The included records are compiled in place of the record on which the INCLUDE option appears. The included records can themselves contain INCLUDE options; in this way, included source input can be nested up to five levels deep.

The <file specification> construct specifies the file from which source input is to be included. If the <title> form is used, the quoted string specifies the TITLE attribute of the file. The <internal file name> form provides an internal file name that can be associated with an actual file through file equation. The <name and title> form provides both an internal file name available for file equation and a title to be used if the internal file name is not file-equated.

If the <file specification> construct is not used, the same file as that specified in the previous INCLUDE option at the same level of nesting is used. Therefore, the first INCLUDE option at any of the five possible levels of nesting must contain the <file specification> construct.

The <start specification> construct specifies the record of the included file at which inclusion is to start. If the <sequence number> form is specified, inclusion begins with the first record with a sequence number greater than or equal to the specified sequence number. If the asterisk (*) form of <start specification> is used, inclusion begins at the point where it left off the last time inclusion took place from this file at the same level of nesting. If the <start specification> construct is not used, inclusion begins with the first record of the file.

The <stop specification> construct specifies the record after which inclusion is to stop. If the <stop specification> construct is not used, inclusion ends after the last record of the file.

Source files suitable for use by the INCLUDE option can be produced by the compiler by using the NEW option.

Files declared globally in Work Flow Language (WFL) jobs should not be used as INCLUDE files.

The INCLUDE option must be the last option appearing on a compiler control record, and it cannot be used when the programmer is using the batch facility.

Examples

```
$ INCLUDE FILE8 00001000 - 09000000
```

This example instructs the compiler to accept as input all records from the file indicated by the internal name FILE8, with a sequence number greater than or equal to 00001000 and less than or equal to 09000000.

```
$ INCLUDE *
```

This example instructs the compiler to accept as input a portion of the file accessed by the last INCLUDE option at this level of nesting. The records to be included are all records that follow the last record included by that preceding INCLUDE option. If, for example, the preceding INCLUDE option was the one in the first example above, the file that is accessed is FILE8, and the records that are included are all the records with a sequence number greater than 09000000.

```
$ INCLUDE "SOURCE/XYZ." - 900
```

This example instructs the compiler to accept as input a portion of the file with the title "SOURCE/XYZ". The included records are all records of the file with a sequence number less than or equal to 00000900.

```
$ INCLUDE INCL = "SYMBOL/ALGOL/INCLUDE1."
```

This example instructs the compiler to accept as input all records of either the file to which the internal name "INCL" was file-equated or, if the INCL file was not file-equated, the file "SYMBOL/ALGOL/INCLUDE1".

<initialize option>

See also

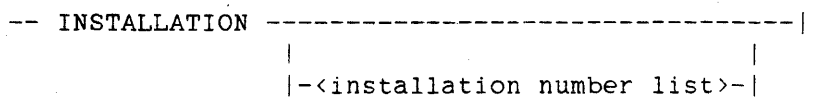
<text> 61

(Type: Binder command)

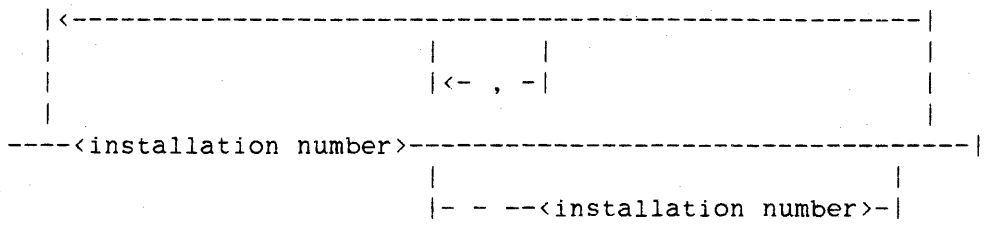
During autobinding, the INITIALIZE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder INITIALIZE statement and are described in the "Binder Reference Manual."

When the batch facility is used, the INITIALIZE option cannot be used.

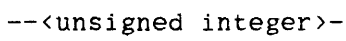
<installation option>



<installation number list>



<installation number>



(Type: Boolean, Default value: FALSE)

When TRUE, the INSTALLATION option causes the compiler to recognize one or more groups of installation intrinsics so that they can be referenced in an ALGOL program. This option must be assigned a value before the first syntactic item in a program. Assigning a value to this option at any other time has no effect.

An installation number must be an unsigned integer between 1 and 2047, inclusive. Each installation number in an installation number list must be strictly greater than the preceding installation number in that list. Installation numbers larger than 2047 are treated as if they were equal to 2047.

An INSTALLATION option with no installation number list is equivalent to one with an installation number list of 100 through 2047.

When the batch facility is used, the INSTALLATION option cannot be assigned a value.

<intrinsics option>

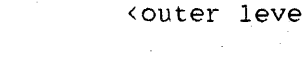
(Type: Boolean, Default value: FALSE)

When TRUE, the INTRINSICS option causes separately compiled procedures to be compiled at lexical (lex) level two and allows a <global part> construct to appear before the procedures. These procedures can then be used as installation intrinsics. A <global part> is not normally allowed when compiling separate procedures at lex level two.

The title of the object code file generated for a procedure when the INTRINSICS option is TRUE is the same as if the procedure were compiled at lex level three. Thus, the separate procedures being compiled can be bound into the intrinsics. When the Binder program is used to bind procedures into the intrinsics, the INTRINSICS option must be assigned the value TRUE before the first source statement.

When the batch facility is used, the INTRINSICS option cannot be assigned a value.

<level option>



(Type: value, Default value: 2 for programs, 3 for separately compiled procedures)

The LEVEL option allows the programmer to override the lexical (lex) levels assigned by the compiler. This feature is needed when compiling separate procedures for binding to a host program. The <outer level> construct specifies the lex level at which compilation is to begin.

The LEVEL option must appear before the first syntactic item in a program, and it cannot be used when the batch facility is used.

<library option>

```
-- LIBRARY --|
```

(Type: Boolean, Default value: TRUE for CANDE-originated compiles and when the SEPCOMP option is TRUE, FALSE otherwise)

When compiling multiple separate procedures (such as intrinsics), assigning TRUE to the LIBRARY option improves the efficiency of the binding. When TRUE, this option causes all object code from this compilation to be put in one file, which is marked as a multiprocedure code file. If the LIBRARY option is FALSE, each separate procedure produces its own object code file.

The LIBRARY option must appear before the first syntactic item in a program, and it cannot be assigned a value when the programmer is using the batch facility.

The LIBRARY option is unrelated to the library facility described in the chapter "Interface to the Library Facility."

<limit option>

```
-- LIMIT --<error limit>--|
```

<error limit>

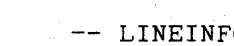
```
--<unsigned integer>--|
```

(Type: value, Default value: 10 for CANDE-originated compiles, 150 otherwise)

The LIMIT option allows the programmer to specify the number of compile-time errors that can occur before the compilation is terminated because of excessive errors.

A limit of 0 indicates that the program is not to be terminated for excessive errors. If, when the LIMIT option is assigned a value, the number of syntax errors already equals or exceeds that value, then the program is immediately terminated.

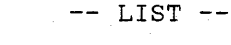
<lineinfo option>



(Type: Boolean, Default value: TRUE for CANDE-originated compiles, FALSE otherwise)

When TRUE, the LINEINFO option causes the compiler to associate sequence number information with the object code. This information is then displayed in the event of a run-time error. A larger code file is generated if the LINEINFO option is TRUE than if it is FALSE.

<list option>



(Type: Boolean, Default value: FALSE for CANDE-originated compiles, TRUE otherwise)

When TRUE, the LIST option causes source input from the CARD and TAPE files and other information to be printed on the compiler LINE file.

When a value is assigned to the LIST option, the same value is assigned to the SEGS option.

<listdeleted option>

```
-- LISTDELETED --|
```

(Type: Boolean, Default value: FALSE)

When both the LIST option and the LISTDELETED option are TRUE, the printer listing includes records from the secondary input file TAPE that are replaced, voided, or deleted during the compilation. The word "REPLACED" appears to the right of the source records replaced by a record from the primary input file, CARD; the word "VOIDT" appears if the record is voided from the TAPE file by the VOIDT option; and the word "DELETED" appears if the record is deleted by a compiler control record that consists of a dollar sign ($) followed by all blanks.

If the LIST option is TRUE but the LISTDELETED option is FALSE, records from the TAPE file that are replaced, voided, or deleted are not written to the printer listing. If the LIST option is FALSE, the value of the LISTDELETED option is ignored.

<listincl option>

```
-- LISTINCL --|
```

(Type: Boolean, Default value: FALSE)

When both the LIST option and the LISTINCL option are TRUE, source records included by using the INCLUDE option are written to the printer listing. If the LIST option is TRUE but the LISTINCL option is FALSE, the included records are not written to the printer listing. If the LIST option is FALSE, the value of the LISTINCL option is ignored.

When the batch facility is used, the LISTINCL option cannot be used.

<listomitted option>

(Type: Boolean, Default value: TRUE)

When both the LIST option and the LISTOMITTED option are TRUE, source records omitted by the OMIT option are written to the printer listing. In the listing, the word "OMIT" appears next to the sequence number of each omitted record. If the LIST option is TRUE but the LISTOMITTED option is FALSE, omitted records are not written to the printer listing. If the LIST option is FALSE, the value of the LISTOMITTED option is ignored.

<listp option>

(Type: Boolean. Default value: FALSE)

When TRUE, the LISTP option causes records from the primary source input file, CARD, to be written to the printer listing. Because these records are also written when the LIST option is TRUE, the LISTP option is effective only when the LIST option is FALSE.

<loadinfo option>

See also
<file specification>. 611

(Type: special)

The DUMPINFO option and the LOADINFO option make it possible for the contents of certain simple variables and arrays of the compiler to be saved in a disk file and subsequently reloaded for separate compilation.

The <file specification> construct specifies the file to be created by the DUMPINFO option or loaded by the LOADINFO option. If the <title> form is used, the quoted string specifies the TITLE attribute of the file. The <internal file name> form provides an internal file name that can be associated with an actual file by file equation. The <name and title> form provides both an internal file name available for file equation and a title to be used if the internal file name is not file-equated. If the <file specification> construct is not used, the default INFO file or the file that is file-equated to the INFO file is used.

These options are used in conjunction with separate compilation of procedures. Typically, all global declarations are compiled, and then the DUMPINFO option is used to dump information about the global declarations from the compiler to the file INFO. When the separate procedures are to be compiled, the file INFO, containing information about all of the global declarations, is read in by the LOADINFO option before the procedures are compiled. For example, consider the following programs:

Program 1

```
BEGIN
   <global declarations>
$ DUMPINFO
END.
```

Program 2

```
%%% LOAD THE GLOBALS
[
$ LOADINFO
   <additional global declarations>
]
<separate PROCEDURE declarations>
```

Each time a loadinfo operation is done, the old information in the affected variables and tables of the compiler is discarded. Thus, compiling different portions of the same program, even if they are in different environments, can be done in the same compilation.

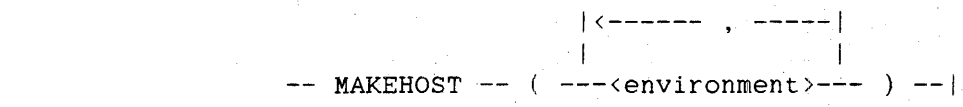
The loadinfo operation changes all items in the INFO file to globals and all procedures already compiled to forward PROCEDURE declarations. Thus, an INFO file created by a DUMPINFO operation that is done immediately before a PROCEDURE declaration in a normal compilation is suitable for loading global declarations when that procedure is to be compiled separately.

When two or more items with the same identifier are declared at different lexical (lex) levels, a separate compilation can access only the last declaration seen before the loadinfo operation occurred.

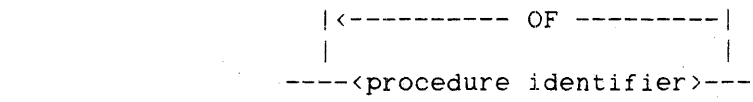
If the release level of the compiler that performs the dumpinfo operation to create an INFO file and the release level of the compiler that performs the loadinfo operation on that file are not the same, a syntax error is given, and the compilation is discontinued. If a loadinfo operation is attempted and the file specified as the INFO file is not in fact an INFO file, or the INFO file was created by a compiler for a different language, a syntax error is given and the compilation is discontinued.

The DUMPINFO option and LOADINFO option must be the last options appearing on a compiler control record, and they cannot be used when using the batch facility.

<makehost option>



<environment>



See also
<procedure identifier>. 165

(Type: Boolean, Default value: FALSE)

Given only the source code and object code of a host program to be changed and the patches to change it, the sepcomp facility of the compiler can separately compile and bind to the host program only the procedures that are being changed. This method, which is particularly useful for large programs, requires that information not normally collected and saved during the compilation of the host program be saved. When TRUE, the MAKEHOST option causes this information to be saved when compiling a program or a procedure at lexical (lex) level two.

If the MAKEHOST option is TRUE, information is saved in the object code file of the program about the symbolic file used or created by the compilation, the sequence ranges of all procedures declared in the outer block of the program, and the global declarations. The saving of the outer block environment enables lex-level-three procedures to be compiled separately within this environment.

The information saved about the items declared in an environment describes all of the items in that environment. There is no information about the relative order of the declarations and therefore which items are visible from which procedures. Thus, a sepcomp of a patch is not necessarily equivalent to a full compile of that patch. For example, given the following host program:

```
$ SET MAKEHOST
BEGIN
   REAL X;
   PROCEDURE P;
      BEGIN
      REAL R;
      R := X + 5;
      END P;
   REAL ARRAY A[0:4];
   P;
END.
```

If a patch replaces the statement

```
R := X + 5;
```

by the statement

```
R := X + A[2];
```

a full compile with the patch fails with a syntax error on the assignment to R, because array A has not yet been declared. A sepcomp of the patch, however, is successful, because the environment information saved for the outer block describes A as well as X and P.

Additional environments can be saved, if desired, so that procedures at lex levels greater than three can be replaced. The list of environments can extend across several source records. Environments must be fully qualified through the outermost level of PROCEDURE declaration, except that for a program that is a procedure, the name of that procedure must not appear. A procedure that is specified as an environment must contain a local declaration. If a specified environment does not contain a local declaration or if it is never found during the course of compilation, the compiler gives a syntax error containing the name of the environment. Environments can appear in any order, without regard to the actual block structure of the host program.

Source records that are inserted into a program using the INCLUDE option are not included in the environment information saved by the MAKEHOST option. This information is not saved because sequence numbers on included records can duplicate sequence numbers occurring in the rest of the program.

The information necessary to make a program into a host program includes the information saved for the Binder program when the NOBINDINFO option is FALSE; therefore, an error is given if both the NOBINDINFO option and the MAKEHOST option are TRUE.

When making a host program, the NEW option should be TRUE if any changes are made to the host program. The default source file associated with the host program is the NEWTAPE file, if one has been created; otherwise, it is the TAPE file.

The MAKEHOST option must appear before the first syntactic item in a program, and it cannot be assigned a value when the programmer is using the batch facility.

Examples

```
$ SET MAKEHOST
BEGIN
   ARRAY A[0:9];
   PROCEDURE P1;
      BEGIN
      BOOLEAN B;
      PROCEDURE INNER;
         BEGIN
         REAL R;
         IF B THEN
            R := * + A[2];
         ...
         END INNER;
      ...
      END P1;
   PROCEDURE P2;
      BEGIN
      ...
      END P2;
   ...
END.
```

When the above program is compiled, information about the global environment is saved in the object code file for the program. If a patch is made to the body of procedure INNER, then during the sepcomp process, all of procedure P1 is recompiled and bound to the

host program. If, however, the MAKEHOST option was

```
$ SET MAKEHOST (P1)
```

then the local environment of P1 is also saved in the object code file of the program. A sepcomp of a patch to the body of procedure INNER would cause only INNER to be recompiled and bound to the host program.

```
$ SET MAKEHOST
$ SET MAKEHOST (PASSONE, PASSTWO, WRAPUP OF PASSTWO)
```

In this example, the second compiler control option overrides the first, saving the environment of procedures PASSONE, PASSTWO, and WRAPUP OF PASSTWO in addition to the global environment.

<MCP option>

```
-- MCP --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the MCP option causes all value arrays, translate tables, truth sets, and constant pools to be allocated at lexical (lex) level two.

The MCP option cannot be assigned a value after the appearance of the first syntactical item in a program or when the programmer is using the batch facility.

<merge option>

```
-- MERGE --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the MERGE option causes the primary source input file, CARD, to be merged with a secondary source input file, TAPE, to form the input to the compiler. If matching sequence numbers occur, the record from CARD overrides the record from TAPE. If the MERGE option is FALSE, only primary source input is used, and the TAPE file is ignored.

The total input to the compiler when the MERGE option is TRUE consists of all records from the CARD file, all records from the TAPE file that do not have sequence numbers identical to those on records in the CARD file, and all records inserted by INCLUDE options. Records in the CARD file also override INCLUDE options in the TAPE file if matching sequence numbers are encountered.

When the batch facility is used, the MERGE option cannot be assigned a value.

<new option>

(Type: Boolean, Default value: FALSE)

When the NEW option is TRUE, the source input to the compiler from the CARD file is written to the updated source output file NEWTAPE. If the MERGE option is also TRUE, then the merged source input from the CARD and TAPE files is written to NEWTAPE. The format of the file written to NEWTAPE is such that it can later be used as input to the compiler.

Records included in the source input by the INCLUDE option are written to the NEWTAPE file only if the INCLNEW option is also TRUE. Compiler control records are written to the NEWTAPE file only if the initial dollar sign ($) does not appear in column 1.

The NEWTAPE file is created whether or not syntax errors occur in the source input.

If the MAKEHOST option is TRUE and the first syntactic item has been compiled, any attempt to assign a value to the NEW option results in a syntax error.

When the batch facility is used, the NEW option cannot be assigned a value.

<newseqerr option>

```
-- NEWSEQERR --|
```

(Type: Boolean. Default value: FALSE)

When TRUE, the NEWSEQERR option causes an error to be given if the sequence number on a record of the NEWTAPE file is not strictly greater than the sequence number of the preceding record. If sequence errors occur and the NEWSEQERR option is TRUE, the NEWTAPE file is not locked, the message "NEWTAPE NOT LOCKED" is displayed on the Operator Display Terminal (ODT), and the message "NEWTAPE NOT LOCKED <number of errors> NEWTAPE SEQUENCE ERRORS" is printed on the printer listing. The NEWSEQERR option is effective even if the CHECK option is FALSE.

When the batch facility is used, the NEWSEQERR option cannot be assigned a value.

<noBCL option>

```
-- NOBCL --|
```

(Type: Boolean. Default value: FALSE)

When TRUE, the NOBCL option causes a syntax error to be given whenever a BCL construct that may lead to the creation of a BCL descriptor is encountered. This option is intended to aid in the elimination of BCL constructs from a program so that the program can run on non-BCL systems.

<nobindinfo option>

-- NOBINDINFO --|

(Type: Boolean, Default value: FALSE)

When TRUE, the NOBINDINFO option prevents information needed by the Binder program from being written to the object code file. The resulting object code file can be executed, but it cannot be used as an input file to the Binder program. Object code files that do not contain binding information (bindinfo) are smaller than object code files that contain bindinfo.

Object code files that contain the timing code provided by the STATISTICS option cannot be bound. Thus, the NOBINDINFO option is assigned the value TRUE if the STATISTICS option is assigned the value TRUE. If the STATISTICS option is TRUE when the NOBINDINFO option is assigned the value FALSE, a syntax error is given.

If the MAKEHOST option is TRUE when the NOBINDINFO option is assigned the value TRUE, a syntax error is given.

When the batch facility is used, the NOBINDINFO option cannot be assigned a value.

<nostackarrays option>

-- NOSTACKARRAYS --|

(Type: Boolean, Default value: FALSE)

When TRUE, the NOSTACKARRAYS option prevents arrays from being allocated within the stack.

When the NOSTACKARRAYS option is FALSE, the data from certain arrays is allocated within the stack. Such arrays are referred to as "in-stack arrays," and can be accessed slightly faster than an array whose data area is allocated in memory.

<noxreflist option>

```
-- NOXREFLIST --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the NOXREFLIST option prevents the SYSTEM/XREFANALYZER program from being initiated by the compiler when cross-reference information is being saved (that is, when either the XREF option or the XREFFILES option is TRUE). Instead, the file "XREF/<code file name>", where <code file name> is the name of the object code file generated by the compiler, remains on disk. SYSTEM/XREFANALYZER can be run later using the file "XREF/<code file name>" as input. The NOXREFLIST option has no effect if both the XREF option and the XREFFILES option are FALSE.

For more information on cross-referencing, refer to the description of the XREF option in this section.

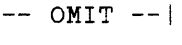
<oldresize option>

```
-- OLDRESIZE --|
```

(Type: Boolean, Default value: FALSE)

The OLDRESIZE option, which affected the semantics of the RESIZE statement, no longer has any effect. The pre-Mark 3.3 semantics are no longer available for newly compiled programs. Any compiler control record that assigns the value TRUE to the OLDRESIZE option causes a syntax error to be given. Any other appearance of the OLDRESIZE option in a compiler control record causes a warning message to be given.

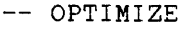
<omit option>



(Type: Boolean, Default value: FALSE)

When TRUE, the OMIT option causes records from the CARD file (and, if the MERGE option is TRUE, from the TAPE file) to be ignored (not compiled). If both the LIST option and the LISTOMITTED option are TRUE, then in the printer listing, the word "OMIT" appears next to the sequence number of each omitted record. When the OMIT option is TRUE, compiler control records with the initial dollar sign ($) in either column 1 or column 2 are recognized, but compiler control records with the $ in columns 3 through 72, inclusive, are ignored.

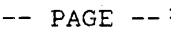
<optimize option>



(Type: Boolean, Default value: FALSE)

When TRUE, the OPTIMIZE option causes additional analysis of Boolean expressions to be performed, and object code is generated to permit early termination of the expression evaluation. Any portion of the Boolean expression that could cause side effects is always evaluated.

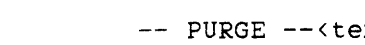
<page option>



(Type: immediate)

When the LIST option is TRUE and the PAGE option appears, the printer listing is spaced to the top of the next page.

<purge option>



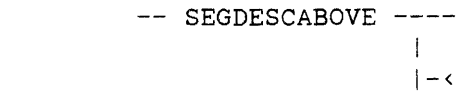
See also
<text>. 61

(Type: Binder command)

During autobinding, the PURGE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder PURGE statement and are described in the "Binder Reference Manual."

When the batch facility is used, the PURGE option cannot be used.

<segdescabove option>



(Type: value, Default value: none)

The SEGDESCABOVE option is used when compiling large programs that may have difficulty in addressing the segment dictionary.

When a host program is compiled, this option causes all code segment descriptors to be allocated starting at the word in the D1 stack specified by the unsigned integer. The unsigned integer must be in the range 4 to 4095, inclusive. If the option is used after the first syntactic item has been compiled, the given value is added to the current size of the D1 stack. The Binder program preserves the segdescabove specification. Care should be taken when using this option, because unused D1 stack locations below the code segment descriptors occupy "save" memory when the program is running.

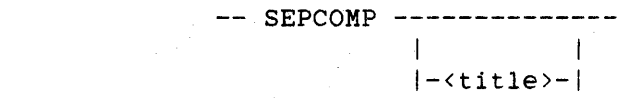
This option is intended to be used when compiling host files and is ignored when separate procedures are compiled. When the batch facility is used, the SEGDESCABOVE option cannot be used.

<segs option>

(Type: Boolean, Default value: FALSE for CANDE-originated compiles, TRUE otherwise)

If both the LIST option and the SEGS option are TRUE, the printer listing will contain beginning and ending segment messages. Assigning a value to the LIST option assigns the same value to the SEGS option. However, to suppress the segment messages, the SEGS option can be assigned the value FALSE even though the LIST option is TRUE. When the value of the LIST option is FALSE, the value of the SEGS option is ignored.

<sepcomp option>



See also
<title> 611

(Type: Boolean, Default value: FALSE)

When TRUE, the SEPCOMP option invokes the automatic separate compilation and binding facility, called the "sepcomp facility."

The title of the host program can be specified either by using the <title> syntax of the SEPCOMP option or by file-equating the HOST file of the compiler. The <title> specification takes precedence over file equation. The title of the default source file is stored in the host program, but this title can be overridden by file equation of the compiler file TAPE.

Compiler control records with blank sequence numbers are accepted following the compiler control record that assigns TRUE to the SEPCOMP option and before the first "patch record." A patch record is a source record with a nonblank sequence number: at least one patch record is required in a sepcomp. Sequence number errors among patch records are not allowed. The SEPCOMP option examines the patch records, decides

which procedures of the host program must be recompiled, and generates Binder input for binding these procedures to the host program. The SEPCOMP option always tries to compile procedures at the highest possible lexical (lex) level. Therefore, the number of extra environments specified when making a host program affects the choices available to the SEPCOMP option.

When TRUE, the SEPCOMP option assigns TRUE to both the AUTOBIND option and the LIBRARY option, causing all procedures to be compiled into one multiprocedure code file (a temporary file used by the Binder program). Explicitly assigning FALSE to the AUTOBIND option prevents the Binder from being called and causes the object code file to be locked on disk if the LIBRARY option is TRUE. Explicitly assigning FALSE to the LIBRARY option causes each procedure compiled to be put in a separate, permanent object code file. Binding still occurs, but at a somewhat slower rate.

If procedures are put in separate code files, the titles of the code files are determined in the standard way, with the procedure name replacing the last identifier from the title on the compile statement that invoked the compiler. Procedures compiled at lex level four and higher have the name of their environment in the code file name also. In the following example, when two lex-level-four procedures are compiled having the same name but different environments, two code files are produced (titled "A/PASSONE/Q" and "A/PASSTWO/Q") in addition to the new host file titled "A/HOST" (assuming that PASSONE and PASSTWO were specified as extra environments when A/HOST was made).

```
$ SEPCOMP "A/HOST"
$ RESET LIBRARY
   % PATCH CARD TO Q OF PASSONE      <sequence number>
   % PATCH CARD TO Q OF PASSTWO      <sequence number>
```

The special information associated with a host program is always copied by the Binder to the object code file of the new program so it can be used as a host. This information is not updated by either the Binder or the compiler during the sepcomp process. Following a sepcomp, this information can inaccurately reflect the actual structure and content of the host program with which it is associated.

Because the arrangement of data in a bound code file differs from that of an unbound code file, binding to a bound host is faster than binding to an unbound host. For this reason, assigning TRUE to the AUTOBIND option when compiling a host program can be advantageous, because it causes the Binder to be invoked.

If the release level of the compiler that creates the host code file and the release level of the compiler that is attempting a sepcomp to that file are not the same, a syntax error is given and the compilation is discontinued. If the code file specified as the host program was not compiled with the MAKEHOST option equal to TRUE, or if the host program was compiled by a compiler for a different language, a syntax error is given and the compilation is discontinued.

The SEPCOMP option automatically assigns values to several other compiler control options in order to simplify operation. The MERGE option is unavailable for use while the SEPCOMP option is TRUE. Assigning TRUE to the MERGE option before assigning TRUE to the SEPCOMP option is not allowed because it destroys the default file equation of the source file to be used for the patches.

When the batch facility is used, the SEPCOMP option cannot be assigned a value. If the TADS option is TRUE when the SEPCOMP option is assigned the value TRUE, the value of the SEPCOMP option is left equal to FALSE and a warning message is given. The SEPCOMP option cannot be assigned a value after the first syntactic item of a program has been compiled. Multiple SEPCOMP option settings are not allowed because, when first assigned TRUE, the SEPCOMP option initiates preprocessing of the source input from the CARD file.

For more information on the sepcomp facility, refer to the description of the MAKEHOST option in this section.

<seq option>

(Type: Boolean, Default value: FALSE)

When TRUE, the SEQ option causes the printer listing and the updated source file, NEWTAPE, to contain new sequence numbers. These new sequence numbers are determined by the current values of the sequence base option and the sequence increment option.

This option is effective only when the LIST option or NEW option is TRUE. The sequence numbers that appear on the records in these files when the SEQ option is FALSE are identical to the sequence numbers on the corresponding records in the input files.

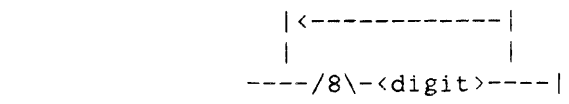
The sequence base option and the sequence increment option are described in this section.

<seqerr option>

(Type: Boolean, Default value: FALSE)

When TRUE, the SEQERR option causes an error to be given if the sequence number on a record of the TAPE file is not strictly greater than the sequence number of the preceding record. If sequence errors occur and the SEQERR option is TRUE, the object code file is not locked, the message "CODE FILE NOT LOCKED" is displayed on the Operator Display Terminal (ODT), and the message "CODE FILE NOT LOCKED <number of errors> TAPE SEQUENCE ERRORS" is printed on the printer listing. The SEQERR option is effective even when the CHECK option is FALSE.

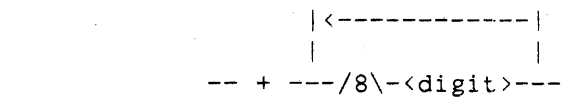
<sequence base option>



(Type: value, Default value: 1000)

The sequence base option specifies the sequence number that is to be assigned to the next record when the SEQ option is TRUE. After each record is resequenced, the value of the sequence base option is increased by the value of the sequence increment option.

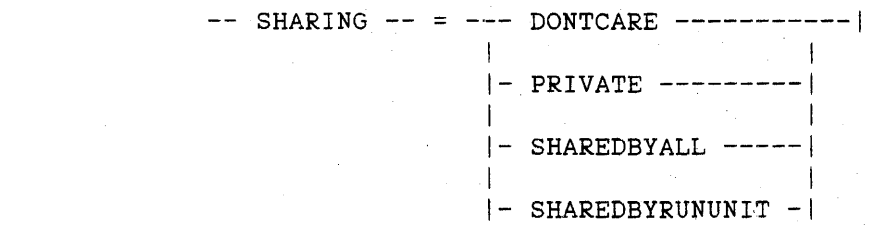
‹sequence increment option›



(Type: value, Default value: 1000)

The value of the sequence increment option is used to increment the value of the sequence base option when records are resequenced because the SEQ option is TRUE.

‹sharing option›



(Type: value, Default value: DONTCARE)

The SHARING option is used in a library program to specify how other programs are to share the library.

DONTCARE
: If the SHARING option has the value DONTCARE, the Master Control Program (MCP) determines the sharing, and it is unknown to all users invoking the library. DONTCARE is the default value of the SHARING option.

PRIVATE
: If the SHARING option has the value PRIVATE, a separate instance of the library is started for each invocation of the library. Any changes made to global items in the library by a block that invoked the library apply only to that user of the library.

SHAREDBYALL

If the SHARING option has the value SHAREDBYALL, all invocations of the library share the same instance of the library. Any changes made to global items in the library by a block that has invoked the library apply to all users of that library.

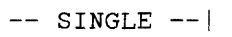
SHAREDBYRUNUNIT

A run unit consists of a program and all libraries that are initiated either directly or indirectly by that program. A "program," in this context, does not include either a library that is not frozen or any tasks that are initiated by the program (that is, a process family is not a run unit). If the SHARING option has the value SHAREDBYRUNUNIT, all invocations of a library within a run unit share the same instance of the library.

Note that a library is its own run unit until it freezes. For example, program P initiates library A and, before library A freezes, it in turn initiates library B. Now library B is in library A's run unit, not in program P's run unit. Had library A initiated library B after freezing, both library A and library B would be in program P's run unit.

The SHARING option must appear before the first syntactic item in the program, and it cannot appear when the batch facility is used.

<single option>



(Type: Boolean, Default value: TRUE if the compiler was compiled with the DOUBLESPACE compiler-generation option equal to FALSE, FALSE otherwise)

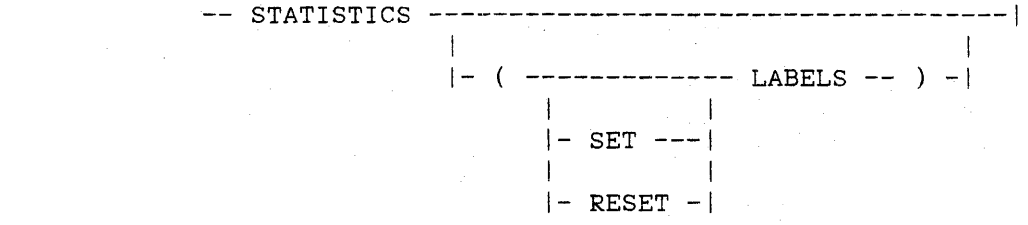
When TRUE, the SINGLE option causes the printer listing to be single-spaced. When FALSE, the SINGLE option causes the printer listing to be double-spaced.

‹stack option›

(Type: Boolean, Default value: FALSE)

If both the LIST option and the STACK option are TRUE, the printer listing includes the relative stack addresses, in the form of address couples, for all program variables. If the LIST option is TRUE but the STACK option is FALSE, the printer listing does not contain these address couples. The value of the STACK option is ignored if the LIST option is FALSE.

‹statistics option›



(Type: Boolean, Default value: FALSE)

When TRUE, the STATISTICS option causes timing statistics to be gathered. The option is examined at the beginning of each procedure or block and, if it is TRUE at that time, timing statistics are gathered for that procedure or block. Although the value of the option can be altered at any time, only its value at the beginning of procedures and blocks is significant in determining whether timings are made.

The Binder program cannot bind object code files that contain the timing code necessary for statistics; therefore, if the NOBINDINFO option is FALSE when the STATISTICS option is assigned the value TRUE, the NOBINDINFO option is assigned the value TRUE and a warning message is given. If the STATISTICS option is TRUE when the NOBINDINFO option is assigned the value TRUE, a syntax error is given.

If statistics are taken for a procedure or block, the frequency of execution of that procedure or block is measured, along with the length of time spent in that procedure or block. When the program is completed for any reason (including both normal and abnormal termination), the statistics of the executing task are printed out to the TASKFILE.

On the statistics output listing, an asterisk (*) indicates that doubt exists about the timing for the specific procedure whose name precedes the asterisk. In addition, timings are invalid for any procedure or block that is resumed by a "bad go to."

Only the first six characters of any identifier are printed on the listing.

For any procedure or block that has statistics gathered, the timings can be broken down to the label level within that procedure or block by using the LABELS syntax. The word "LABELS" can be preceded by "SET" or "RESET"; if both are omitted, SET is assumed. For example,

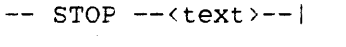
```
$ SET STATISTICS (LABELS)
```

begins timing of label breakpoints, and

```
$ SET STATISTICS (RESET LABELS)
```

ends timing of label breakpoints. The words "SET" or "RESET" inside the parentheses affect only the LABELS specification.

<stop option>



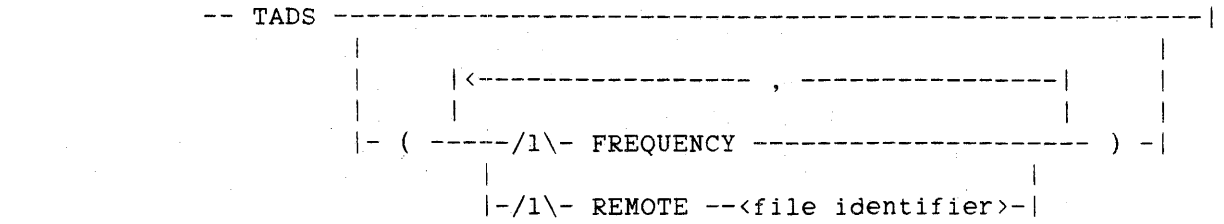
See also
<text> 61

(Type: Binder command)

During autobinding, the STOP option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder STOP statement and are described in the "Binder Reference Manual."

When the batch facility is used, the STOP option cannot be used.

<TADS option>



See also
<file identifier> 85

(Type: Boolean, Default value: FALSE)

When the TADS option is TRUE, special debugging code and tables are generated as part of the object code file. The tables are generated to support the symbolic debugging environment of the ALGOL Test and Debug System (TADS). For more information on ALGOL TADS, refer to the "ALGOL Test and Debug System (TADS) User's Guide."

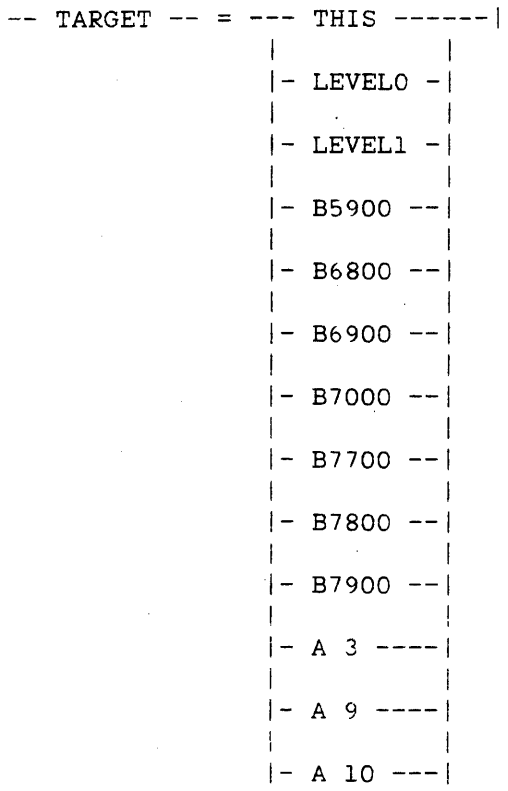
The FREQUENCY option causes additional code and tables to be generated for coverage and frequency analysis. This option must be specified if either the TADS "FREQUENCY" command or the TADS "COVERAGE" command is to be used to determine the number of times individual statements have been executed or which statements have not been executed.

The REMOTE option allows TADS to share a remote file with the program being tested. Sharing a file might be necessary, because only one remote input file can be open for each station. If specified, the <file identifier> must correspond to a file identifier declared in a FILE declaration occurring later in the outer block of the program. The file must have the attributes KIND equal to REMOTE, UNITS equal to CHARACTERS, and FILEUSE equal to IO. The MAXRECSIZE attribute must not be less than 72. The file must not be declared to be a direct file.

If the AUTOBIND option, the MAKEHOST option, or the SEPCOMP option is TRUE when the TADS option is assigned the value TRUE, then the TADS option is assigned the value FALSE and a warning message is given. Programs compiled with the TADS option equal to TRUE cannot assign TRUE to the MAKEHOST option, the AUTOBIND option, or the SEPCOMP option, and they cannot be used as input files to the Binder program.

The TADS option must appear before the first syntactic item in a program.

<target option>



(Type: Value, Default value: installation-defined)

The TARGET option specifies the computer family for which the generated code is to be optimized. The keyword "THIS" indicates that code is to be optimized for the computer on which it is compiled. The keyword "LEVEL0" indicates that code is to be optimized for the B 5000/B 6000/B 7000 Series of computers. The keyword "LEVEL1" indicates that code is to be optimized for the A Series of computers. The generated object code can be run on any computer that is code-compatible with the specified family. Table 7-2 indicates code-compatible families.

Table 7-2. Code-Compatible Families

Target	Compatible Families									
	B 5900	B 6800	B 6900	B 7000	B 7700	B 7800	B 7900	A 3	A 9	A 10
LEVEL0	x	x	x	x	x	x	x	x	x	x
B5900	x	x	x	x	x	x	x	x	x	x
B6800	x	x	x	x	x	x	x	x	x	x
B6900	x	x	x	x	x	x	x	x	x	x
B7000	x	x	x	x	x	x	x	x	x	x
B7700	x	x	x	x	x	x	x	x	x	x
B7800	x	x	x	x	x	x	x	x	x	x
B7900	x	x	x	x	x	x	x	x	x	x
LEVEL1								x	x	x
A 3								x	x	x
A 9								x	x	x
A 10								x	x	x

For information on how an installation defines the default target computer family, refer to the "COMPILERTARGET" command in the "Operator Display Terminal (ODT) Reference Manual."

This option must appear before the first syntactical item in the program. It cannot be used in the same program as the B7700 option.

<time option>

(Type: Boolean, Default value: FALSE)

When TRUE, the TIME option causes trailer information, such as the number of errors, the number of code segments, and the compilation time, to be printed on the printer listing. Because this trailer information is also printed when the LIST option is TRUE, the value of the TIME option is effective only when the LIST option is FALSE.

<use option>

```
-- USE --<text>--|
```

See also
<text> 61

(Type: Binder command)

During autobinding, the USE option is passed directly to the Binder program for analysis. The format and function of this option are the same as those of the Binder USE statement and are described in the "Binder Reference Manual."

When the batch facility is used, the USE option cannot be used.

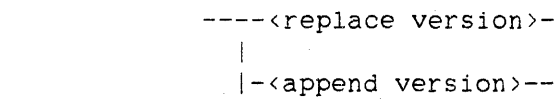
<user option>

```
--<identifier>--|
```

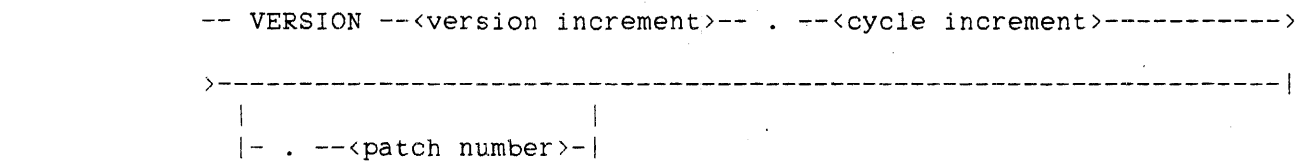
(Type: Boolean, Default value: FALSE)

If an identifier on a compiler control record is not recognized as one of the predefined options, it is considered to be a user option. A user option can be manipulated exactly like any other Boolean option; that is, it can be assigned values by using the SET, RESET, and POP keywords. In addition, it can be used in option expressions to assign values to standard Boolean options or to other user options.

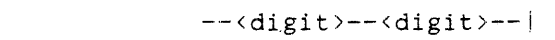
<version option>



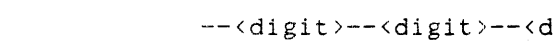
<replace version>



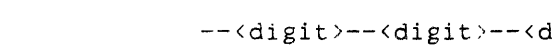
<version increment>



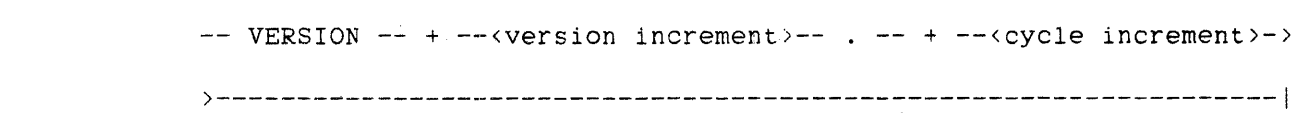
<cycle increment>



<patch number>



<append version>



(Type: value, Default value: 00.000.000)

The VERSION option allows the user to specify an initial version number for a program, replace an existing version number, or append to an existing version number.

If the NEW option is TRUE, a VERSION option appears in the TAPE file, and the CARD file contains a <replace version> or <append version>, then the VERSION option record in the NEWTAPE file is updated with the version, cycle, and patch number in the last VERSION option record in

the CARD file. The sequence number of the VERSION option in the CARD file must be less than the sequence number of the VERSION option in the TAPE file.

The functions COMPILETIME(20), COMPILETIME(21), and COMPILETIME(22) allow a programmer to access the current version, cycle, and patch numbers, respectively. For more information, see "<compiletime function>" in the chapter "Expressions."

When the batch facility is used, the VERSION option cannot be used.

See also
<compiletime function>. 538

Examples

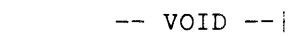
```
$ VERSION 25.010.010
```

Sets the current version to 25.010.010.

```
$ VERSION +01.+001.010
```

Increments the current version by 01 in the version increment, by 001 in the cycle increment, and assigns 010 to the patch number. For example, if the existing version is 25.010.005 and this VERSION option is compiled, the resulting version is 26.011.010.

<void option>



(Type: Boolean, Default value: FALSE)

When TRUE, the VOID option causes all source input other than compiler control records from both the TAPE and CARD files to be ignored by the compiler until the VOID option is assigned the value FALSE. The ignored source input is neither listed nor included in the updated source file, regardless of the values of the LIST option and the NEW option. Once the VOID option is assigned TRUE, it can be assigned FALSE only by a compiler control record in the CARD file.

<voidt option>

```
-- VOIDT --|
```

(Type: Boolean, Default value: FALSE)

If the MERGE option is TRUE and the VOIDT option is TRUE, all source input from the TAPE file is ignored by the compiler until the VOIDT option is assigned the value FALSE. Therefore, while the VOIDT option is TRUE, only primary source input from the file CARD is compiled. The ignored input is neither listed nor included in the updated source file, regardless of the values of the LIST option and the NEW option. Once the VOIDT option is assigned TRUE, it can be assigned FALSE only by a compiler control record in the CARD file. If the MERGE option is FALSE, the value of the VOIDT option is ignored.

<warnsupr option>

```
-- WARNSUPR --|
```

(Type: Boolean, Default value: FALSE)

When TRUE, the WARNSUPR option prevents warning messages from being given.

<writeafter option>

```
-- WRITEAFTER --|
```

(Type: Boolean, Default value: FALSE)

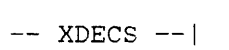
The WRITEAFTER option provides the ability to designate whether carriage control is performed before or after a write operation. The option can be assigned values repeatedly in order to select before-write or after-write carriage control for individual files and I/O statements.

Normally in ALGOL, carriage control is performed following a write operation.

If the WRITEAFTER option is TRUE when a FILE declaration is compiled, then for all I/O statements that explicitly reference the file, carriage control is done before a write operation. For any I/O statement compiled when the WRITEAFTER option is TRUE, carriage control is done before the write operation.

The WRITEAFTER option does not apply to direct files or direct I/O statements.

<xdecs option>

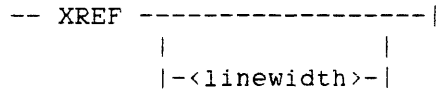


(Type: Boolean, Default value: TRUE if either the XREF option or the XREFFILES option is TRUE, FALSE otherwise)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only identifiers declared while the XDECS option is TRUE are included in the cross-reference information. This option is assigned TRUE when the XREF option or the XREFFILES option is assigned TRUE and can be assigned a value as many times as desired. If the XDECS option is assigned the value TRUE and both the XREF option and the XREFFILES option are FALSE, the XDECS option is assigned FALSE and a syntax error is given.

For more information on cross-referencing, refer to the description of the XREF option in this section.

<xref option>



<linewidth>

An <integer> between 72 and 160, inclusive.

(Type: Boolean, Default value: FALSE)

Depending on the values of several related compiler control options, the compiler optionally generates cross-reference information containing an alphabetized list of identifiers that appear in the program and, for each identifier, the type of the item named by that identifier, the sequence number of the source input record on which the identifier is declared, the sequence numbers of the input records on which the identifier is referenced, and other relevant information. The following factors can be controlled through the use of compiler control options:

1. Whether or not the compiler saves cross-reference information as it is processing the source input

2. Which identifiers and which references to these identifiers are cross-referenced

3. Whether or not the SYSTEM/XREFANALYZER program is initiated automatically by the compiler

4. If SYSTEM/XREFANALYZER is automatically initiated, whether it is to produce printed output, disk files suitable for input to SYSTEM/INTERACTIVEXREF and the Editor, or both

The compiler saves cross-reference information if either the XREF option or the XREFFILES option or both options are TRUE. This information is discarded if any syntax errors occur during compilation. If used, these options should be assigned TRUE before any source input has been processed. Once assigned TRUE, these options cannot be assigned FALSE.

The identifiers and their references to be cross-referenced can be selected by the XDECS option and the XREFS option, respectively. Neither of these options can be used if cross-reference information is not being saved by the compiler (that is, if both the XREF option and the XREFFILES option are FALSE).

When the compiler is saving cross-reference information, this information is written to a disk file in raw form. Before this information can be printed or read by SYSTEM/INTERACTIVEXREF and the Editor, it must be analyzed by SYSTEM/XREFANALYZER. If the NOXREFLIST option is FALSE, the compiler automatically initiates SYSTEM/XREFANALYZER to process the raw cross-reference file. If the NOXREFLIST option is TRUE, SYSTEM/XREFANALYZER is not initiated, and the compiler's raw file is left on disk with the title "XREF/<code file name>", where <code file name> is the name of the object code file produced by the compiler; this file can be analyzed at a later time by running SYSTEM/XREFANALYZER directly.

If the compiler initiates SYSTEM/XREFANALYZER (that is, if the NOXREFLIST option is FALSE), the program produces either a printed listing or a pair of disk files suitable for the Editor and

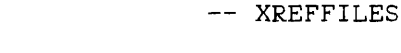
SYSTEM/INTERACTIVEXREF. If the XREF option is TRUE, a listing is produced. If the XREFFILES option is TRUE, the pair of disk files is produced; these files are titled "XREFFILES/<code file name>/XREFS" and "XREFFILES/<code file name>/XDECS". If both the XREF option and the XREFFILES option are TRUE, both a listing and the disk files are produced.

The line width, in characters, of the listing produced by SYSTEM/XREFANALYZER can be specified by the <linewidth> construct. If not specified, the line width is 132.

User options are included in the cross-reference information.

SYSTEM/XREFANALYZER is described under "XREFANALYZER" in the "System Software Utilities Reference Manual," and SYSTEM/INTERACTIVEXREF is described under "INTERACTIVEXREF" in the "System Software Utilities Reference Manual." The Editor is described in the "Editor User's Guide."

<xreffiles option>

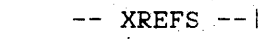


(Type: Boolean, Default value: FALSE)

When TRUE, the XREFFILES option causes cross-reference information to be saved by the compiler and causes the SYSTEM/XREFANALYZER program, if it is initiated by the compiler, to produce files that can be used by the Editor and SYSTEM/INTERACTIVEXREF. These files have the titles "XREFFILES/<code file name>/XDECS" and "XREFFILES/<code file name>/XREFS", where <code file name> is the name of the object code file that the compiler is generating.

For more information on cross-referencing, refer to the description of the XREF option in this section.

<xrefs option>

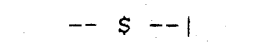


(Type: Boolean, Default value: TRUE if either the XREF option or the XREFFILES option is TRUE, FALSE otherwise)

When the compiler is saving cross-reference information because the XREF option or the XREFFILES option is TRUE, only identifier references that are encountered while the XREFS option is TRUE are included in the cross-reference information. This option is assigned TRUE when the XREF option or the XREFFILES option is assigned TRUE, and it can be assigned a value as many times as desired. If the XREFS option is assigned the value TRUE when both the XREF option and the XREFFILES option are FALSE, then the XREFS option is assigned FALSE and a syntax error is given.

For more information on cross-referencing, refer to the description of the XREF option in this section.

<$ option>



(Type: Boolean, Default value: FALSE)

If both the LIST option and the $ option are TRUE, the printer listing includes all compiler control records. If the LIST option is TRUE but the $ option is FALSE, only compiler control records with an initial dollar sign ($) in columns 2 through 72, inclusive, appear in the printer listing. If the LIST option is FALSE, the value of the $ option is ignored.

8 INTERFACE TO THE LIBRARY FACILITY

The library facility is a feature that can be used to structure processes. A library is a program containing one or more procedures that can be called by other programs, which are referred to as "calling programs." These procedures are "entry points" into the library. Unlike a regular program, which is always entered at the beginning, a library can be entered at any entry point.

Libraries provide all the benefits of procedures plus the added advantages that they can be reused, and they can be shared by a number of programs. Consolidating logically related functions into a library can make programming easier and program structure more visible. A call on a procedure in a library is equivalent to a call on a procedure in the calling program.

Libraries offer the following improvements over binding:

1. Interlanguage communication is significantly improved.

2. Standard packages of functions (such as plotting and statistics) need not be copied into any calling programs.

Libraries offer the following improvements over installation intrinsics:

1. A library can have its own global files, databases, transaction bases, and so on.

2. Libraries can contain initialization and termination code.

3. Libraries can themselves call other libraries.

4. Individual users can create their own libraries without possessing special privileges.

5. Libraries can be written in more languages than can the installation intrinsics.

6. More than one version of a library can be in use at a time.

FUNCTIONAL DESCRIPTION OF LIBRARIES

This section describes how the library facility operates.

Library Programs

A library program is a program that specifies entry points (procedures) for use by calling programs and is distinguished by the occurrence of EXPORT declarations and FREEZE statements. A procedure in a library is specified to be an entry point by appearing in an EXPORT declaration. A library program becomes a library after execution of a FREEZE statement.

Calling Programs

A calling program is a program that calls entry points provided by a library and is distinguished by the appearance of LIBRARY declarations and PROCEDURE declarations that contain the ‹library entry point specification› construct.

A library can itself function as a calling program and call other libraries. The only restriction is that a chain of library linkages must never be circular. That is, a library cannot reference itself, either directly or indirectly, through a chain of library references.

Library Directories and Templates

The information used by the Master Control Program (MCP) to match entry points in a library with entry points declared in a calling program is contained in a pair of data structures called the "directory" and the "template," which are built by the compilers.

When a program exports entry points and contains a FREEZE statement, the object code file for that program contains a library directory. One directory exists for each block that contains an EXPORT declaration. After a library "freezes" (executes a FREEZE statement), only one directory is in effect until the library program finishes executing.

A library directory contains a description of all the entry points in the library. This description includes the following information:

1. The name of the entry point

2. The entry point's type

3. A description of the entry point's parameters

4. Information on how the entry point is provided (see "Linkage Provisions" later in this chapter)

When a program declares a library and entry points in that library, the object code file for the program contains a library template that describes the library and the declared entry points. One template exists for each library declared in the calling program. A template contains the following information:

1. A description of the attributes of the library

2. A description of all the entry points of the library that are declared by the program. Each description includes the following information:

   a. The name of the entry point

   b. The entry point's type

   c. A description of the entry point's parameters

The SETACTUALNAME function can be used to change the name of an entry point in an unlinked library template. (For more information, refer to the SETACTUALNAME function in the chapter "Expressions.")

See also
Linkage Provisions. 659
<setactualname function>. 575

Library Initiation

On the first call on an entry point of a library or at an explicit linkage request, the calling program is suspended. The description of the entry point in the library template of the calling program is compared to the description of the entry point with the same name in the library directory associated with the referenced library.

If the entry point does not exist in the library, or if the two entry point descriptions are not compatible, a run-time error is given and the calling program is terminated. If the entry point exists and the two entry point descriptions are compatible, the MCP automatically initiates the library program (if it has not already been initiated). The library program runs normally until it executes a FREEZE statement, which makes

the entry points available. All of the entry points of the library that are declared in the calling program are linked to the calling program, and the calling program resumes execution.

If a calling program declares an entry point that does not exist in the library, no error is generated when the library is initiated; however, a subsequent call on that entry point causes a "MISSING ENTRY POINT" run-time error, and the calling program is terminated.

A library can be specified to be permanent or temporary. A permanent library remains available until it is terminated either by the Operator Display Terminal (ODT) commands DS (Discontinue) or THAW, or by execution of a CANCEL statement. A temporary library remains available as long as users of the library remain. A temporary library that is no longer in use "unfreezes" and resumes running as a regular program.

The PERMANENT or TEMPORARY specifications of the FREEZE statement control the duration of a library. Any running program that executes a FREEZE statement becomes a library. When a library is initiated by explicitly running the library program instead of by calling an entry point, the FREEZE statement should specify PERMANENT. (If TEMPORARY is specified, the library immediately unfreezes because it has no users.) After a library unfreezes, it must not execute another FREEZE statement in an attempt to become a library again.

Because a library program initially runs as a regular program, the flow of execution can be such that the execution of a FREEZE statement is conditional and can occur anywhere in the program.

If a calling program causes a library program to be initiated and the library program terminates without executing a FREEZE statement (for example, because it was not actually a library program and, thus, had no FREEZE statement), the attempted linkage to the library entry points cannot be made, and the calling program is terminated.

Linkage to a library can be explicitly requested by using the LINKLIBRARY function, which is described in the chapter "Expressions."

See also
<linklibrary function>. 559

Linkage Provisions

Entry points declared in a calling program are linked to corresponding entry points provided by a library in one of three ways:

1. Directly
2. Indirectly
3. Dynamically

The library program specifies the form of linkage. Indirect and dynamic linkages allow linkage to be established to libraries other than the library specified by the calling program. The calling program can control the library invocation to which it is linked only by specifying the object code file title or the function name of the library or, for dynamic linkage, by specifying the LIBPARAMETER library attribute. Depending on the value of the LIBACCESS library attribute, the TITLE attribute or FUNCTIONNAME attribute is used to specify the object code file title of the library. (For a discussion of the library attributes, refer to "Library Attributes" in this chapter.)

Direct linkage occurs when the library program contains the procedure that is named in the EXPORT declaration of the library.

Indirect linkage occurs when the library program exports a procedure that is declared as an entry point of another library. The MCP then attempts to link the calling program to this second library, which can provide the entry point directly, indirectly, or dynamically.

Dynamic linkage allows a library program to determine at link time which library task the calling program will be linked to. The library program must provide a selection procedure that accepts the value of the LIBPARAMETER library attribute as a parameter. Based on the value of LIBPARAMETER, the selection procedure selects and initiates a library task. The selection procedure must also accept, as a second parameter, a procedure. This procedure, which is provided by the MCP to verify that the library task is valid and complete, must be called before the selection procedure is exited. The MCP calls the selection procedure at link time. (Refer to "Library Examples" in this chapter for a more detailed explanation and examples of libraries that provide dynamic linkage.)

The only restrictions on the complexity of indirect and dynamic linkages are as follows:

1. Eventually, some library must provide the entry point directly.

2. The chain of referenced libraries must never become circular.

See also
Library Attributes. 665
Library Examples. 671

Discontinuing Linkage

A program can delink from a library program by using either the CANCEL statement or the DELINKLIBRARY function.

The CANCEL statement causes the library program to unfreeze and resume running as a regular program. The CANCEL statement can be used only on PRIVATE and SHAREDBYRUNUNIT libraries.

The DELINKLIBRARY function affects only the linkage between the program executing the DELINKLIBRARY function and the specified library. Any other programs linked to the specified library are not affected. (For more information, see the DELINKLIBRARY function in the chapter "Expressions.")

See also
<delinklibrary function>. 543

Error Handling

Any fault caused (and ignored) by a procedure in a library that is invoked by a calling program is treated as a fault in the calling program. If ignored by the calling program, this fault causes the calling program to be terminated but has no effect on the status of the library.

If a library program faults (or is otherwise terminated) before executing a FREEZE statement, then all calling programs that are waiting to link to that library program are also terminated.

If a library is terminated while calling programs are linked to it. those calling programs are also terminated.

The first call on an entry point in a library causes library linkage to be made. In this phase, an attempt is made to locate and establish links to all entry points declared by the calling program. If an entry point declared in the calling program does not exist in the library, its linkage cannot be established, and any subsequent calls to that entry point result in a "MISSING ENTRY POINT" error. This error continues to occur whenever a calling program links to that instance of the library and calls that entry point. Thus, it is advisable to remove that instance of the library (by either a THAW or DS (Discontinue) ODT command) and initiate a correct version of the library. (For more information, refer to the "Operator Display Terminal (ODT) Reference Manual" for a description of these commands.)

CREATING LIBRARIES

A library program is created by using the EXPORT declaration to declare procedures to be exported as entry points, and by using the FREEZE statement. The duration of a library program following initiation is controlled by the TEMPORARY or PERMANENT specification of the FREEZE statement. The allowed sharing of a library program is controlled by the SHARING compiler control option, described below.

Library Sharing Specifications

Users of a library can be restricted through the normal file access features provided by the system. The allowed simultaneous usage of a library can be specified by the creator of the library at compile time through the SHARING compiler control option. The library sharing can be PRIVATE, SHAREDBYALL, SHAREDBYRUNUNIT, or DONTCARE.

PRIVATE
: A separate instance of the library is started for each invocation of the library. Any changes made to global items in the library by the program unit (block, procedure, or external task) invoking the library apply only to that particular calling program.

SHAREDBYALL
: All invocations of the library share the same instance of the library. Any changes made to global items in the library by a program unit that has invoked the library apply to all users of that library.

SHAREDBYRUNUNIT
: A run unit consists of a program and all libraries that are called, either directly or indirectly, by that program. A "program," in this context, excludes both a library that is not frozen and any tasks that are initiated by the program (that is, a process family is not a run unit). All invocations of a library within a run unit share the same instance of the library.

  Note that a library is its own run unit until it freezes. For example, program P initiates library A and, before library A freezes, it in turn initiates library B. Now library B is in library A's run unit, not in program P's run unit. Had library A initiated library B after freezing, both library A and library B would be in program P's run unit.

DONTCARE
The MCP determines the sharing. This determination is unknown to all users invoking the library.

The default value of the SHARING compiler control option is DONTCARE.

REFERENCING LIBRARIES

To use a library, the calling program does the following:

- Declares the library in a LIBRARY declaration, specifying the attributes of the library
- Declares the entry points of the library in PROCEDURE declarations with <library entry point specification> parts

When an entry point is invoked or at an explicit linkage request, the MCP automatically creates the library linkage. If the library program has not already been initiated, the MCP initiates it; then, when the library is frozen, the MCP links the library to the calling program. The MCP attempts to make linkages to all entry points referenced in a library at the time that the library is first invoked.

The LINKLIBRARY function can be used to determine whether or not the calling program is currently linked to or is capable of being linked to a particular library program. If the calling program is not currently linked but is capable of being linked, the linkage is performed. During the linkage process, an attempt is made to link to every entry point exported from the library whose name matches an entry point declared in the calling program. Only those names that match are checked for correct function type, number of parameters, and parameter types. Therefore, the LINKLIBRARY function does not check that every entry point declared in the calling program is also exported from the library. (For more information, see the LINKLIBRARY function in the "Expressions" chapter.)

See also
<linklibrary function>. 559

The CANCEL statement and the DELINKLIBRARY function can be used to terminate the linkage between a calling program and a library. The CANCEL statement causes the library to unfreeze and resume running as a regular program regardless of whether it is temporary or permanent. Only PRIVATE libraries or SHAREDBYRUNUNIT libraries can be canceled. The DELINKLIBRARY function has no effect on any other users of the library. (For more information on the DELINKLIBRARY function, refer to the description of the function in the chapter "Expressions.")

See also
<delinklibrary function>. 543

The SETACTUALNAME function determines whether or not the name of a particular library entry point can be changed in the template to a particular character string and, if possible, makes the change. The name of an entry point of a linked library cannot be modified. Therefore, a linked library must be delinked before the SETACTUALNAME function can be called to change the name of any of its entry points. (For more information, see the SETACTUALNAME function in the chapter "Expressions.")

See also
‹setactualname function›. 575

Library Attributes

Libraries, like files, have attributes that can be assigned values and tested programmatically.

The calling program can change library attributes dynamically; however, since the MCP ignores any changes made to library attributes of linked libraries, these changes must not be made while the program is linked to the library. Any library attribute changes must be made before the calling program has linked to the library or after the library has been delinked from the program.

This section describes the library attributes. The first line of each description tells whether the attribute can be read or written or both; its type; and its default value, if any.

FUNCTIONNAME

(Read/Write, EBCDIC string-valued)

FUNCTIONNAME specifies the system function name used to find the target object code file for the library. (For more information, refer to the LIBACCESS attribute.)

INTNAME

(Read/Write, EBCDIC string-valued)

INTNAME specifies the internal identifier for the library.

LIBACCESS

(Read/Write, mnemonic-valued, Default value: BYTITLE)

LIBACCESS specifies the way in which a library object code file is accessed when a library is called. LIBACCESS has one of the mnemonic values BYTITLE or BYFUNCTION. If the value is BYTITLE, the TITLE attribute of the library is used to locate the object code file. If the value is BYFUNCTION, the value of the FUNCTIONNAME attribute of the library is used to access the MCP library function table, and the object code file associated with that FUNCTIONNAME is used.

LIBPARAMETER

(Read/Write, EBCDIC string-valued)

LIBPARAMETER is used to transmit information from the calling program to the selection procedures of libraries that provide entry points dynamically.

TITLE

(Read/Write, EBCDIC string-valued)

TITLE specifies the object code file title of the library. (For more information, refer to the LIBACCESS attribute.)

Example

The following program shows examples of how the LIBACCESS, TITLE, and FUNCTIONNAME library attributes can be used.

```
BEGIN
   % LIBRARY1, DECLARED BELOW, IS NOT A SYSTEM LIBRARY.  ITS LIBACCESS
   % ATTRIBUTE, SET TO BYTITLE, INDICATES THAT THE TITLE ATTRIBUTE
   % IS USED TO LOCATE THE LIBRARY'S CODE FILE.

   LIBRARY LIBRARY1(TITLE="OBJECT/LIBRARY1.",LIBACCESS=BYTITLE);
   REAL PROCEDURE PROC1;
      LIBRARY LIBRARY1;
   REAL PROCEDURE PROC2;
      LIBRARY LIBRARY1;
   REAL PROCEDURE PROC3;
      LIBRARY LIBRARY1;

   % THE LIBRARY DECLARED BELOW IS A SYSTEM LIBRARY.  ITS LIBACCESS
   % ATTRIBUTE, SET TO BYFUNCTION, INDICATES THAT THE FUNCTIONNAME
   % ATTRIBUTE OF THE LIBRARY IS LOOKED UP IN THE MCP LIBRARY FUNCTION
   % TABLE, AND THE CODE FILE ASSOCIATED WITH THE FUNCTIONNAME,
   % SYSTEMLIB, IS USED.

   LIBRARY LIBRARY2(FUNCTIONNAME="SYSTEMLIB.",LIBACCESS=BYFUNCTION);
   PROCEDURE SYSTEMLIBPROC;
      LIBRARY LIBRARY2;
         .
         .
   <executable statements>
         .
         .

END.
```

Entry Point Type Matching

Library entry points can be either typed or untyped, and they can have parameters. Type matching is performed on entry points during library linkage. If the description of an entry point in the template does not match the description of the entry point in the directory, the linkage is not made, and the calling program is terminated. Matching is based on several factors: the procedure type, the number of parameters, the parameter types, and the ways in which the parameters are passed. Parameters are passed as call-by-value, call-by-reference, or call-by-name.

An ALGOL library entry point can be any of the following:

- ASCII string procedure
- Boolean procedure
- Complex procedure
- Double procedure
- EBCDIC string procedure
- Hexadecimal string procedure
- Integer procedure
- Real procedure
- Untyped procedure

The parameters of an ALGOL library entry point can be any of the following:

- Boolean variable, array, or direct array
- Double variable, array, or direct array
- Real variable, array, or direct array
- Integer variable, array, or direct array
- Complex variable or array
- EBCDIC string variable or array
- ASCII string variable or array

- Hexadecimal string variable or array
- EBCDIC character array or direct array
- ASCII character array or direct array
- Hexadecimal character array or direct array
- Event variable or array
- Task variable or array
- File or direct file
- Pointer
- A fully-specified procedure (declared using "FORMAL") with the above restrictions on its possible parameters and type

Parameter Passing

If a library program declares a parameter to be call-by-name, the calling program can declare the parameter to be call-by-name, call-by-reference, or call-by-value. If a library program declares a parameter to be call-by-reference, the calling program can declare the parameter to be call-by-name, call-by-reference, or call-by-value. If a library program declares a parameter to be call-by-value, the calling program can declare the parameter only to be call-by-value.

Table 8-1 illustrates the parameter passing rules.

Table 8-1. Parameter Passing Rules

	Calling Program		
Library Program	Name	Reference	Value
Name	X	X	X
Reference	X	X	X
Value			X

In ALGOL programs, parameters are declared to be either call-by-value or call-by-name. In ALGOL library programs, parameters to entry points that are declared to be call-by-value are described in the directory as call-by-value; parameters declared to be call-by-name are described in the directory as call-by-reference, except for formal procedures and Boolean, complex, double, integer, and real variables, which are described in the directory as call-by-name. In ALGOL calling programs, parameters to entry points that are declared to be call-by-value are described in the template as call-by-value; parameters declared to be call-by-name are described in the template as call-by-reference, except for Boolean, complex, double, integer, and real variables, which are described in the template as call-by-name.

An array parameter in ALGOL that is declared with any of its lower bounds as an asterisk (*) lower bound is described in the template or directory as N+1 parameters, where N is the number of dimensions of the formal array. The first parameter is the array itself, followed by the N lower bounds described as call-by-value integer variables.

LIBRARY EXAMPLES

This section gives examples of libraries and calling programs that call these libraries.

Library: OBJECT/FILEMANAGER/LIB

The following library program illustrates dynamic linkage. This library provides a set of file management routines. The users of this library would assign the title of the file to be used to the LIBPARAMETER attribute. LIBPARAMETER is then used at link time to determine to which library task the user is to be linked.

This library represents features of dynamic linkage but does not necessarily represent efficient programming.

```
$ SHARING = PRIVATE
BEGIN
   TASK ARRAY LIBTASKS[0:10];       % PROVIDES UP TO 11 DIFFERENT LIBRARY
                                    % TASKS
   STRING ARRAY FILETITLES[0:10];   % LIBPARAMETER FOR EACH OF THE TASKS

   PROCEDURE FILEMANAGER(TASKINDEX);
   VALUE TASKINDEX;
   INTEGER TASKINDEX;

      BEGIN
      PROCEDURE READFILE;
         BEGIN
            .
            .
            .
         END READFILE;
      PROCEDURE WRITEFILE;
         BEGIN
            .
            .

         END WRITEFILE;

      EXPORT READFILE,WRITEFILE;
      FREEZE(TEMPORARY);
      FILETITLES[TASKINDEX] := ".";
      END FILEMANAGER;
```

```
PROCEDURE SELECTION(USERSFILE,MCPCHECK);
VALUE USERSFILE;
EBCDIC STRING USERSFILE;
PROCEDURE MCPCHECK(T); TASK T; FORMAL;

   BEGIN
   INTEGER TASKINDEX;
   BOOLEAN FOUND;
   % LOOK AT ALL THE FILETITLES, CHECKING TO SEE IF A LIBRARY TASK
   % HAS ALREADY BEEN INITIATED FOR FILE TITLE USERSFILE.

   WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
      BEGIN
      IF FILETITLES[TASKINDEX] = USERSFILE THEN
         FOUND := TRUE
      ELSE
         TASKINDEX := *+1;
      END;

   IF NOT FOUND THEN
      BEGIN
      % A LIBRARY TASK DOES NOT EXIST FOR THIS FILE TITLE.
      WHILE NOT FOUND DO   % FIND AN UNUSED TASK
         BEGIN
         TASKINDEX := 0;
         WHILE NOT FOUND AND (TASKINDEX LEQ 10) DO
            IF LIBTASKS[TASKINDEX].STATUS LEQ 0 THEN
               FOUND := TRUE
            ELSE
               TASKINDEX := *+1;
         IF NOT FOUND THEN
            % WAIT A SECOND AND MAYBE A LIBRARY TASK WILL GO TO EOT.
            WAIT((1));
         END;

      PROCESS FILEMANAGER(TASKINDEX) [LIBTASKS[TASKINDEX]];
      WHILE LIBTASKS[TASKINDEX].STATUS NEQ VALUE(FROZEN) DO
         WAIT((1));
      FILETITLES[TASKINDEX] := USERSFILE;
      END;

   MCPCHECK(LIBTASKS[TASKINDEX]);
   END SELECTION;
```

```
   PROCEDURE READFILE;
      BY CALLING SELECTION;
   PROCEDURE WRITEFILE;
      BY CALLING SELECTION;

   EXPORT READFILE,WRITEFILE;
   FREEZE(TEMPORARY);
END.
```

At library linkage time, the procedure SELECTION is invoked. SELECTION accepts two parameters, USERSFILE and MCPCHECK.

USERSFILE is passed the value of the LIBPARAMETER attribute, which was assigned a value by the calling program. The SELECTION procedure checks to see if a library task has been initiated for the file specified by USERSFILE. If it has, then the calling program is linked to that task. If no library task exists for that file, then a new library task is initiated and the calling program is linked to it.

Only one call is made on the SELECTION procedure per linkage; that is, all links to entry points in this library are resolved during linkage. Therefore, any changes made to any library attributes after linkage is made are ignored. The attributes can be changed if the library is delinked.

MCPCHECK is a procedure that is provided by the MCP and must be called before exiting the SELECTION procedure. The parameter to MCPCHECK is the task variable of the library task to which the calling program is to be linked. MCPCHECK verifies that the task is valid and complete. The actual library linkage is not performed until SELECTION has been exited.

Calling Program #1

The following calling program invokes the dynamic library OBJECT/FILEMANAGER/LIB previously described. In this example, the user has set the LIBPARAMETER attribute to "MYFILE", which is the name of the file to be accessed. At library linkage time, which occurs during the call on READFILE, the library procedure SELECTION is invoked. All links to the library's entry points are resolved during linkage. Changes to the library attributes are ignored after linkage is made. However, the program can delink from the library and change the library attributes before any re-linking.

```
BEGIN
   LIBRARY L (TITLE="OBJECT/FILEMANAGER/LIB."):
   PROCEDURE READFILE;
      LIBRARY L;
   PROCEDURE WRITEFILE;
      LIBRARY L;

   L.LIBPARAMETER := "MYFILE";

   READFILE;                          % LINKAGE IS MADE

   CANCEL(L);                         % LINKAGE IS BROKEN

   L.LIBPARAMETER := "OTHERFILE"; % LIBPARAMETER CAN BE CHANGED
                                      % BECAUSE THE LIBRARY HAS BEEN
                                      % CANCELED.

   WRITEFILE;                         % LINKAGE IS MADE AGAIN AND NEW
                                      % VALUE OF LIBPARAMETER IS USED
END.
```

Library: OBJECT/SAMPLE/LIBRARY

The following ALGOL library, compiled as OBJECT/SAMPLE/LIBRARY, provides its entry points directly.

```
BEGIN
   ARRAY MSG[0:120];

   INTEGER PROCEDURE FACT(N);
   INTEGER N;

      BEGIN
      IF N LSS 1 THEN
         FACT := 1
      ELSE
         FACT := N * FACT(N - 1);
      END;  % OF FACT

   PROCEDURE DATEANDTIME(TOARRAY,WHERE);
   ARRAY TOARRAY[*];
   INTEGER WHERE;

      BEGIN
      REAL T;
      POINTER PTR;

      T := TIME(7);
      PTR := POINTER(TOARRAY,8) + WHERE:
      CASE T.[5:6] OF
         BEGIN
         0: REPLACE PTR:PTR BY "SUNDAY, ";
         1: REPLACE PTR:PTR BY "MONDAY, ";
         2: REPLACE PTR:PTR BY "TUESDAY, ";
         3: REPLACE PTR:PTR BY "WEDNESDAY, ";
         4: REPLACE PTR:PTR BY "THURSDAY, ";
         5: REPLACE PTR:PTR BY "FRIDAY, ";
         6: REPLACE PTR:PTR BY "SATURDAY, ";
         END;
      REPLACE PTR BY T.[35:6] FOR 2 DIGITS, "-",
                     T.[29:6] FOR 2 DIGITS, "-",
                     T.[47:12] FOR 4 DIGITS, ", ",
                     T.[23:6] FOR 2 DIGITS, ":",
                     T.[17:6] FOR 2 DIGITS, ":",
                     T.[11:6] FOR 2 DIGITS;
      END; % OF DATEANDTIME
```

```
   EXPORT FACT,DATEANDTIME AS "DAYTIME";
   REPLACE POINTER(MSG,8) BY
      "  - SAMPLE LIBRARY STARTED",
      " " FOR 94;
   DATEANDTIME(MSG,60);
   DISPLAY(MSG);
   FREEZE(TEMPORARY);
   REPLACE POINTER(MSG,8)+19 BY "ENDED  ";
   DATEANDTIME(MSG,60);
   DISPLAY(MSG);
END.
```

In this library program, two procedures are exported, making them entry points that can be called by calling programs. The two procedures, FACT and DATEANDTIME, are contained within the library program, so they are provided directly.

In the EXPORT declaration, the procedure DATEANDTIME is given the name DAYTIME in an AS clause. In the directory built for this library, the name of this entry point will be DAYTIME. Calling programs must use the name DAYTIME to call this entry point.

Library: OBJECT/SAMPLE/DYNAMICLIB

The following ALGOL library, compiled as OBJECT/SAMPLE/DYNAMICLIB, illustrates dynamic and indirect library linkage. This library references the library OBJECT/SAMPLE/LIBRARY previously described.

```
BEGIN
   TASK LIB1TASK,LIB2TASK;
   LIBRARY SAMLIB(TITLE="OBJECT/SAMPLE/LIBRARY.");

   % ENTRY POINT PROVIDED INDIRECTLY
   INTEGER PROCEDURE FACT(N);
   INTEGER N;
      LIBRARY SAMLIB;

   % POSSIBLY CALLED BY THE SELECTION PROCEDURE
   PROCEDURE DYNLIB1;
      BEGIN % PRINTS DATE WITH TIME
      LIBRARY SAMLIB(TITLE="OBJECT/SAMPLE/LIBRARY.");
      % ENTRY POINT PROVIDED INDIRECTLY
      PROCEDURE DAYTIME(TOARRAY,WHERE);
      ARRAY TOARRAY[*];
      INTEGER WHERE;
         LIBRARY SAMLIB;
```

```
   EXPORT DAYTIME;
   FREEZE(TEMPORARY);
   END; % OF DYNLIB1

% POSSIBLY CALLED BY THE SELECTION PROCEDURE
PROCEDURE DYNLIB2;
   BEGIN % PRINTS DATE WITHOUT TIME.
   % ENTRY POINT PROVIDED DIRECTLY
   PROCEDURE DAYTIME(TOARRAY,WHERE);
   ARRAY TOARRAY[*];
   INTEGER WHERE;

      BEGIN
      REAL T;
      T := TIME(7);
      REPLACE POINTER(TOARRAY,8) + WHERE
         BY T.[35:6] FOR 2 DIGITS, "-",
            T.[29:6] FOR 2 DIGITS, "-",
            T.[47:12] FOR 4 DIGITS;
      END; % OF DAYTIME

   EXPORT DAYTIME;
   FREEZE(TEMPORARY);
   END; % OF DYNLIB2

% THE SELECTION PROCEDURE
PROCEDURE THESELECTIONPROC(LIBPARSTR,NAMINGPROC);
VALUE LIBPARSTR;
EBCDIC STRING LIBPARSTR;
PROCEDURE NAMINGPROC(LIBTASK); TASK LIBTASK; FORMAL;

   BEGIN
   IF LIBPARSTR EQL "WITH TIME" THEN
      BEGIN
      IF LIB1TASK.STATUS NEQ VALUE(FROZEN) THEN
         PROCESS DYNLIB1 [LIB1TASK];
      NAMINGPROC(LIB1TASK);
      DISPLAY(" *** CALLING DYNLIB1 ");
      END
   ELSE
      BEGIN
      IF LIB2TASK.STATUS NEQ VALUE(FROZEN) THEN
         PROCESS DYNLIB2 [LIB2TASK];
      NAMINGPROC(LIB2TASK);
      DISPLAY(" *** CALLING DYNLIB2 ");
      END;
   END; % OF THE SELECTION PROCEDURE
```

```
   % ENTRY POINT PROVIDED DYNAMICALLY
   PROCEDURE DAYTIME(TOARRAY,WHERE);
   ARRAY TOARRAY[*];
   INTEGER WHERE;
      BY CALLING THESELECTIONPROC;

   EXPORT FACT,     % PROVIDED INDIRECTLY
          DAYTIME; % PROVIDED DYNAMICALLY
   FREEZE(TEMPORARY);
END.
```

Calling Program #2

The following calling program invokes OBJECT/SAMPLE/DYNAMICLIB, the library described previously.

```
BEGIN
   LIBRARY MYLIB(TITLE="OBJECT/SAMPLE/DYNAMICLIB.");
   INTEGER PROCEDURE FAKTORIAL(N);
   INTEGER N;
      LIBRARY MYLIB(ACTUALNAME="FACT");

   PROCEDURE DAYTIME(A,W);
   ARRAY A[*];
   INTEGER W;
      LIBRARY MYLIB;

   REAL T;
   ARRAY DATIME[0:120];

   MYLIB.LIBPARAMETER := "WITH TIME";
   REPLACE POINTER(DATIME[0],8) BY
      " 13 FACTORIAL IS ",
      FAKTORIAL(13) FOR 12 DIGITS,
      "     -     ";
   DAYTIME(DATIME[*],40);
   DISPLAY(DATIME[0]);
END.
```

In this program, the declaration of the library entry point FAKTORIAL specifies that the ACTUALNAME of the entry point is FACT. In the template built for the library MYLIB, the name of this entry point is FACT, so for linkage to occur, the directory of the library OBJECT/SAMPLE/DYNAMICLIB must contain an entry point named FACT. However, within the program, the entry point is referred to as FAKTORIAL.

9 DMSII INTERFACE

An interface to Burroughs Data Management System II (DMSII) is provided in the BDMSALGOL (Burroughs Data Management System ALGOL) language. The BDMSALGOL language is based on Burroughs Extended ALGOL and contains extensions that enable a programmer to declare and use databases. These extensions provide the following capabilities:

- Invoking a database
- Manipulating data through data management statements
- Using database items through a mapping syntax
- Processing exceptions

Programs written in the BDMSALGOL language must be compiled with the BDMSALGOL compiler. Typically, this compiler is titled "SYSTEM/BDMSALGOL".

The extensions to ALGOL that make up the BDMSALGOL language are described in this chapter.

9.1 INVOKING A DATABASE

DATABASE DECLARATION

Like all variables, a database must be declared in a BDMSALGOL program before it is referenced. However, a DATABASE declaration is unlike other declarations in that it is actually an invocation of a database that has already been fully described and declared in a Data and Structure Definition Language (DASDL) program.

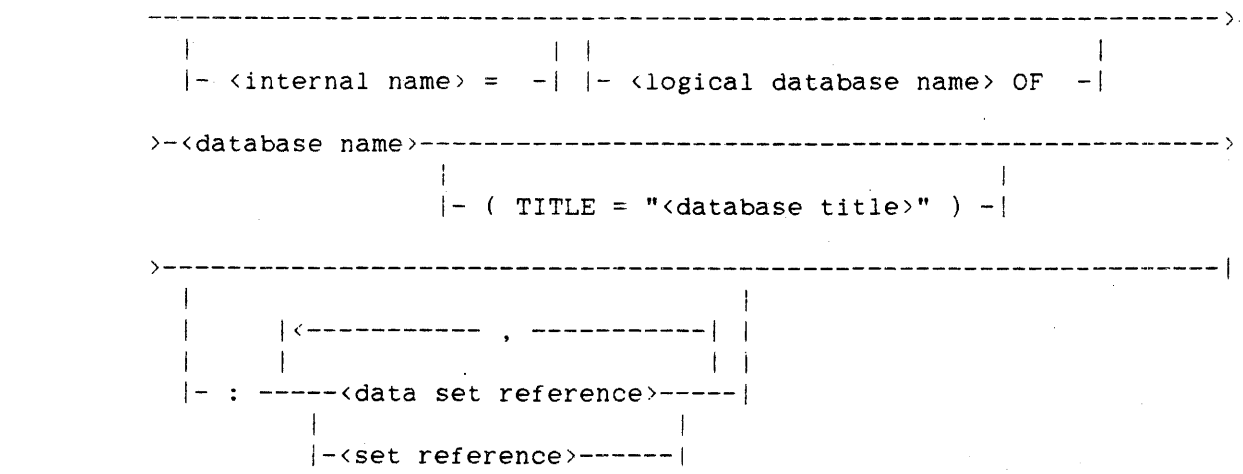
If the compiler control options LIST and LISTDB are both TRUE, all invoked structures, together with the record formats, item and key descriptions, database titles, and other pertinent information, are written on the program listing. The LISTDB option should be used, and the resulting information studied carefully, when database application programs are being developed. For more information on the LISTDB option, see "BDMSALGOL Compiler Control Options" in this chapter.

See also
<listdb option> 772

Syntax

<database declaration>

<database reference>



<internal name>

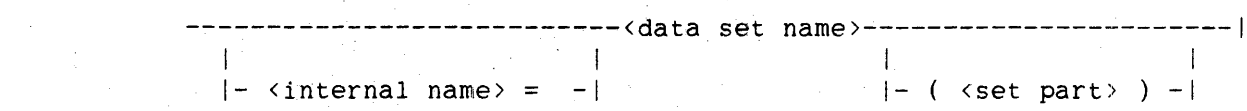
<logical database name>

<database name>

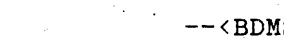
<database title>

A properly formed <file title constant> (as defined in the "Work Flow Language (WFL) Reference Manual") that has only one node; that is, that contains no slashes (/).

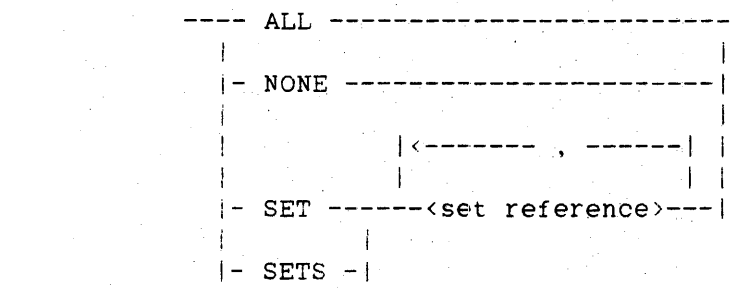
<data set reference>



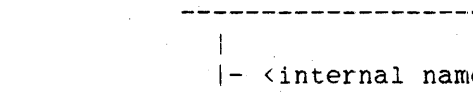
<data set name>



<set part>



<set reference>



<set name>

--<BDMS identifier>--|

See also
<BDMS identifier> 691

Semantics

A DATABASE declaration declares a database and specifies which database or which parts of a database are to be invoked. If no <data set reference> parts and no <set reference> parts are specified in a DATABASE declaration, then all data sets and all sets for each data set are implicitly invoked.

The <internal name> construct assigns an internal name by which a database, data set, set, or subset is known within the program. When an internal name is specified, all subsequent references to the structure must use this internal name.

A database, data set, set, or subset can be invoked more than once; however, the external name (the name in the description file) can be used to reference only one invocation of a structure. Internal names must be used to provide unique names for all other invocations of a structure. The default internal name of a structure is its external name.

By using the internal names in the <data set reference> or the <set reference> constructs, multiple record areas or set paths can be established. Thus, several records of a single data set can be manipulated simultaneously.

The <logical database name> construct allows the program to reference a logical database. A program can invoke structures selectively from a logical database, or it can invoke the entire logical database. Selective invocations are specified in the same manner as for physical databases; however, the choice of structures is limited to those structures included in the logical database.

The <database name> gives the external name of the database to be invoked.

The <database title> construct is an alphanumeric string. The usercode, if any, is the usercode of the control file. The single node of the title is the directory node under which the database files are stored. The family name, if any, is the family name of the control file. The default database title is the external name of the database plus the control file usercode and family name, if any, from the description file. When opening the database, the Master Control Program (MCP) builds the control file title from the database title specified in the declaration. See the "DMSII DASDL Reference Manual" for a discussion of control files and description files.

This title equation is used only at run time, and cannot be used at compile time to specify the title of the database description file. The primary use of the <database title> construct is for modeling. See the "DMSII DASDL Reference Manual" for a description of modeling.

The <data set reference> construct specifies a particular data set from the declared database. If a <data set reference> is used, only the specified structures are invoked. A data set reference must be used to invoke a disjoint data set.

The <data set name> construct gives the external name of the data set to be invoked.

The <set part> construct invokes specific sets from the data set declared in the <data set reference> that contains it. If the <set part> construct is omitted, all sets are implicitly invoked. If the <set part> construct is used, all sets (ALL), no sets (NONE), or only the specified sets are invoked.

The <set reference> construct establishes a set that is not implicitly associated with any particular record area. To load a record area using the set name specified in a set reference, The "<data set> VIA" form of the <selection expression> must be used.

The <set name> construct gives the external name of the set to be invoked.

Pragmatics

Only disjoint structures can be explicitly invoked. Embedded data sets, sets, and subsets are always implicitly invoked if their master data sets are (implicitly or explicitly) invoked. When a data set containing an embedded set associated with a disjoint data set is invoked, or a data set containing a link to another disjoint data set is invoked, then a path is established. However, the user must invoke the disjoint data set if it is to be used.

Multiple invocations of a structure provide multiple record areas or set paths, or both, so that several records of a single data set can be manipulated simultaneously. Selecting only needed structures for UPDATE and INQUIRY provides better use of system resources; a smaller Structure Information Block (SIB) is required, and fewer files can be opened.

If remaps are declared in DASDL, they are invoked in the same manner as conventional data sets.

Examples: Simple Database

The following examples apply to the database DB described by the following DASDL description:

```
D DATA SET (
    K NUMBER (6);
    R NUMBER (5):
    );
S1 SET OF D KEY K:
S2 SET OF D KEY R;
```

```
DATABASE DB: D
```

This declaration establishes one current record area for the data set D, one path for the set S1 of data set D, and one path for the set S2 of data set D. The statements "FIND S1", "MODIFY S1", "FIND S2", and "MODIFY S2" automatically load the data into the D record area.

```
DATABASE DB: D, X=D (NONE)
```

This declaration establishes two current record areas (D and X) and two paths (S1 and S2). The sets S1 and S2 are implicitly associated with the D record area. The set part NONE prevents a set from being associated with X. Thus, the statements "FIND S1" and "FIND S2" load the D record area. The statements "FIND X VIA S1" and "FIND X VIA S2" must be executed to load the X record area using a set.

```
DATABASE DB: D, X=D
```

This declaration shows how multiple current record areas and multiple current paths can be established. The statement "FIND S1 OF D" loads the D record area without disturbing the path S1 OF X, and the statement "FIND S1 OF X" loads the X record area without disturbing the path S1 OF D. Qualification of S1 is necessary to distinguish the paths.

```
DATABASE DB: D (SET S1), X=D (SET S1), Y=D (NONE)
```

This declaration shows how more current record areas than paths can be established. Three record areas (D, X, and Y) are established, but only two paths (S1 OF D, and S1 OF X). The program must execute the statement "FIND Y VIA S1 OF D", "FIND Y VIA S1 OF X", or "FIND Y" to load the Y record area.

```
DATABASE DB: X=D (SET S1), Y=D (SET T=S1)
```

This declaration uses the <set part> syntax to explicitly associate a set with a given work area. The statement "FIND S1" loads the X record area, and the statement "FIND T" loads the Y record area. S1 and T both use the same key.

```
DATABASE DB: D, SY=S1
```

This declaration shows how a set reference can be used to establish a set that is not implicitly associated with any particular record area. The statement "FIND D VIA SY" must be executed to load a record area using the set S1.

Example: Invoking Disjoint Data Sets

The following example shows when a data set reference must be used to invoke disjoint data sets. The database DB is described by the following DASDL description:

```
F DATA SET (
   FI NUMBER (4);
   );
E DATA SET (
   EK NUMBER (8);
   );
D DATA SET (
   A NUMBER (6);
   SE SET OF E KEY EK;
   LINK REFERENCE TO F;
   );
```

If data set references are not specified to invoke E and F, as in the declaration

```
DATABASE DB: D
```

the paths are established by invoking the embedded set SE and the link item LINK. However, these paths cannot be used unless data set references for E and F are specified to establish record areas associated with these paths, as in the declaration

```
DATABASE DB: D,E,F
```

Example: Invoking a Logical Database

In this example, the database EXAMPLEDB is described by the following DASDL description:

```
D1 DATA SET (
   A REAL;
   B NUMBER (5);
   C ALPHA (10);
   );
S1A SET OF D1 KEY IS A;
S1B SET OF D1 KEY IS (A,B,C);
D. DATA SET (
   X FIELD (8);
   Y NUMBER (2);
   Z REAL;
   E DATA SET (
      V1 REAL;
      V2 ALPHA (2);
      );
   SE SET OF E KEY IS V1;
   );
S2A SET OF D2 KEY IS X;
S2B SET OF D2 KEY IS (X,Y,Z);
LDB1 DATABASE (D1(NONE), D2(SET S=S2A));
LDB2 DATABASE (D1(SET S1=S1B), D2(SET S2=S2B));
LDB3 DATABASE (D=D2);
```

The following BDMSALGOL program invokes the logical database LDB1 of EXAMPLEDB. The program can see data sets D1 and D2, but not any of the sets associated with D1, and can see only set S2A associated with D2. S2A appears as set S. The output produced by the LISTDB compiler control option is shown with the program.

```
$ SET LIST LISTDB
BEGIN
   DATABASE LDB1 OF EXAMPLEDB;
*DATABASE TITLE: EXAMPLEDB ON DISK
*01 D1: DATA SET (#2)
*       INVOKED SETS:
*       RECORD ITEMS:
*02        REAL A
*02        INTEGER B: NUMBER (5)
*02        STRING C: ALPHA (10)
*01 D2: DATA SET (#5)
*       INVOKED SETS:
*          S (#8, AUTOMATIC), KEY = X
*       RECORD ITEMS:
*02        REAL X: FIELD (8)
*02        INTEGER Y: NUMBER (2)
*02        REAL Z
*02        E: DATA SET (#6)
*             INVOKED SETS:
*                SE (#7, AUTOMATIC), KEY = V1
*             RECORD ITEMS:
*03             REAL V1
*03             STRING V2: ALPHA (2)
*DESCRIPTION TIMESTAMP: 06/09/82 @ 17:30:34
END.
```

DATABASE EQUATION

The term "database equation" refers to three separate operations:

1. Specification of database titles during compilation.

2. Work Flow Language (WFL) database equation to override compiled-in titles. (For more information, refer to the "DMSII User Language Interface Software Operation Guide" for the WFL syntax.)

3. Run-time manipulation of database titles.

To take advantage of the re-entrance capability of the Accessroutines, the user must be able to specify the title of a database at run time. Database equation allows the database title to be specified at run time, and allows access to databases that are stored under other usercodes and on families that are not visible to a task.

Database equation is operationally similar to file equation. WFL database equation overrides the specification of a database title in the DATABASE declaration, and run-time modification of a database title overrides both WFL database equation and the DATABASE declaration. However, database equation differs from file equation in that a run-time error results if a BDMSALGOL program attempts to assign a value to or examine the TITLE attribute of a database while it is open. For an explanation of the TITLE database attribute, refer to "DATABASE Declaration" in this chapter.

The following syntax shows how the database TITLE attribute can be manipulated during program execution.

Syntax

<database attribute assignment statement>

```
--<string-valued database attribute>-- := --<string expression>--|
```

<string-valued database attribute>

```
--<internal name>-- . -- TITLE --|
```

See also
<internal name> 681

Semantics

The string expression must evaluate to a string in the form of a database title.

The <string-valued database attribute> construct can be used anywhere a string expression is valid.

Database titles never end with a period, and a replace pointer-valued attribute statement is not valid for making assignments to database titles.

NOTE

BDMSALGOL programs employing database equation must be compiled with a Mark 3.2 or later BDMSALGOL compiler.

Example

```
BEGIN
   STRING S;
   DATABASE MYDB (TITLE="LIVEDB");
   OPEN UPDATE MYDB;
   ...
   CLOSE MYDB;
   MYDB.TITLE := "(UC)TESTDB ON TESTPACK";
   OPEN UPDATE MYDB;
   ...
   CLOSE MYDB;
   S := TAKE(MYDB.TITLE,5);
   ...
END.
```

In this example, the first BDMS "OPEN" statement opens the database with the title "LIVEDB", whose data and control files are stored under the user's directory. The second OPEN statement invokes the database "TESTDB", whose files are stored on TESTPACK under the usercode UC.

9.2 BDMSALGOL BASIC LANGUAGE CONSTRUCTS

The following constructs are used within the DATABASE declaration and in data management statements and functions. These sections describe the forms of names for databases, data sets, sets, items, and so on; input mapping and output mapping; and the selection expression.

9.2.1 BDMS IDENTIFIERS AND QUALIFICATION

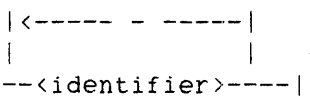
Naming conventions in DASDL for databases and their components follow COBOL rules: that is, names can contain hyphens, and some item and structure names can require qualification. Although both of these conventions contradict normal ALGOL naming rules, they must be allowed in programs that declare and use databases.

BDMS IDENTIFIERS

The identifier of a database, data set, set, item, and so on is in the form of a <BDMS identifier>.

Syntax

<BDMS identifier>.



Semantics

The <BDMS identifier> construct must be fewer than 64 characters long.

Examples

If a database is described in DASDL by the following:

```
D-S DATA SET (
   A-1 NUMBER (5);
   A-2 NUMBER (10);
   );
```

then in a BDMSALGOL program, the data set D-S and the items A-1 and A-2 can be referenced as in the following examples:

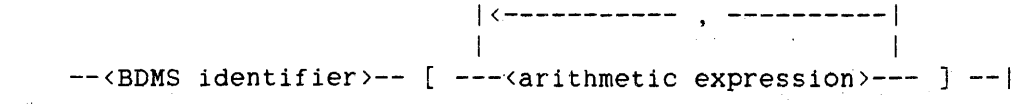
```
INTEGER I;
GET D-S (I := A-1);
PUT D-S (A-2 := I);
```

IDENTIFIERS OF OCCURRING ITEMS

If an item is declared in the DASDL description to have an OCCURS clause, then its identifier must be subscripted to denote which of its occurrences is to be used.

Syntax

<subscripted BDMS identifier>



Semantics

The leftmost arithmetic expression denotes the subscript of the outermost OCCURS clause that affects the item, the next arithmetic expression to the right denotes the subscript of the next outermost OCCURS clause, and so on.

Examples

If items A and B are described in DASDL as follows:

```
DS DATA SET (
   G GROUP (
      A ALPHA (10);
      B NUMBER (4) OCCURS 3 TIMES;
      )
   OCCURS 2 TIMES;
   );
```

there are two occurrences of A, denoted

```
A[1]        A[2]
```

and there are six occurrences of B, denoted

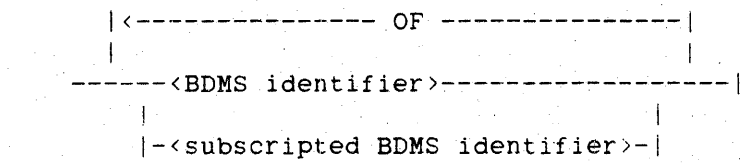
```
B[1,1]      B[2,1]
B[1,2]      B[2,2]
B[1,3]      B[2,3]
```

QUALIFICATION

Database item names need not be unique within a database. Qualification is used to distinguish between database items with the same names.

Syntax

<qualification>



Semantics

An item name can be qualified by the name of any structure that physically contains the item. Any number of qualification names desired can be used, provided that the result is unique. If improper or insufficient qualification is used, a syntax error is given.

A set name can be qualified by the name of the data set it spans.

A group name can be used to qualify an item it contains.

Qualification need not be used if the unqualified name is unique. Qualification must be used whenever there is ambiguity. A variable name can be declared with the same name as a database item in BDMSALGOL without requiring qualification of the item name.

Examples

If a database is described in DASDL as follows:

```
DS1 DATA SET (
   N NUMBER (4);
   );
DS2 DATA SET (
   N NUMBER (4);
   );
```

then the following BDMSALGOL statements indicate how qualification is used to distinguish between the two data items named N.

```
SET N OF DS1 TO NULL;
SET N OF DS2 TO NULL;
```

9.2.2 REFERENCING DATABASE ITEMS

The record area (user work area) is not directly accessible to a BDMSALGOL program. Instead, an explicit mapping between database data items and program variables must be specified whenever access to those items is desired.

Mappings specify the source and destination of data to be transferred into or out of a user work area. Mappings are of two kinds: input mappings and output mappings.

Example

If a database is described in DASDL by the following:

```
D1 DATA SET (
   A NUMBER (5);
   X NUMBER (5) OCCURS 3 TIMES;
   );
```

then the items of data set D1 can be referenced in the following ways:

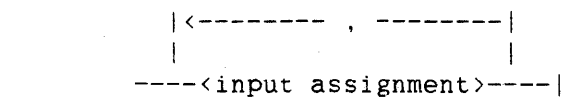
```
INTEGER B,Y1,Y2,Y3;
% THE FOLLOWING STATEMENT TRANSFERS THE VALUE OF DATABASE ITEM A
% TO THE LOCALLY DECLARED INTEGER B.
GET D1 (B := A);

% THE FOLLOWING STATEMENT TRANSFERS THE VALUE OF LOCALLY DECLARED
% INTEGER B TO THE WORK AREA FOR D1.
PUT D1 (A := B);

% THE FOLLOWING STATEMENT TRANSFERS THE VALUES OF ALL THREE
% OCCURRENCES OF X INTO Y1, Y2, AND Y3.
GET D1 (Y1 := X[1],
        Y2 := X[2],
        Y3 := X[3]);

% THE FOLLOWING STATEMENT TRANSFERS THE VALUES OF LOCALLY DECLARED
% INTEGERS Y1, Y2, AND Y3 INTO THE THREE OCCURRENCES OF DATABASE
% ITEM X.
PUT D1 (X[1] := Y1,
        X[2] := Y2,
        X[3] := Y3);
```

INPUT MAPPING

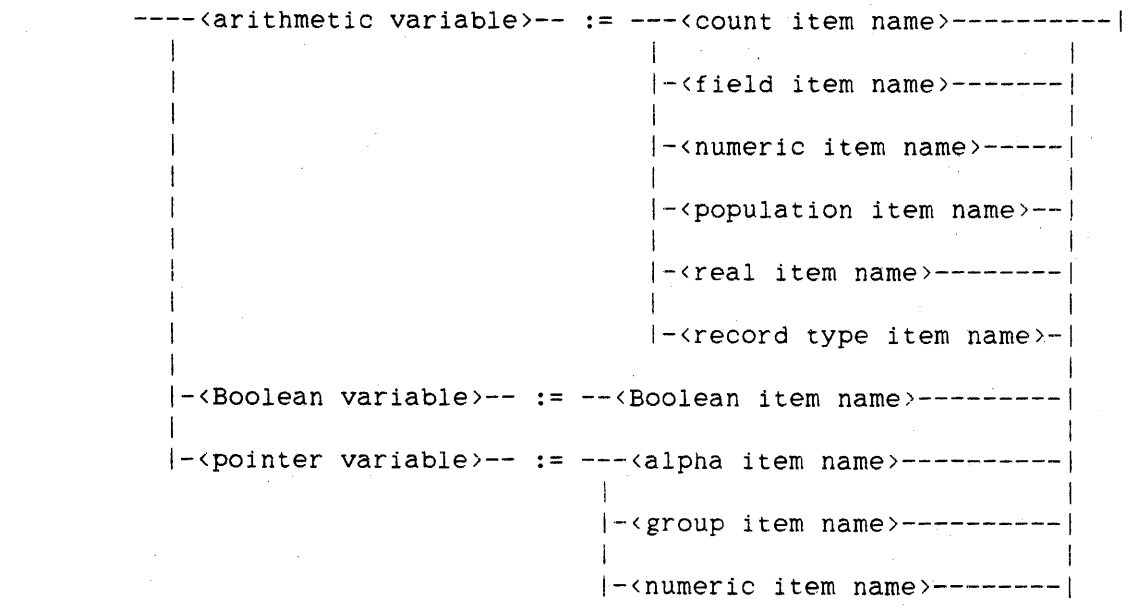
Input mappings can be used with the "retrieval" statements DELETE, FIND, GET, BDMS "LOCK," and MODIFY. Input mappings transfer the value of a DASDL-declared data item to a program variable. If the data item is an occurring item (that is, if the item is declared in DASDL with an OCCURS clause), it must be subscripted appropriately.

Syntax

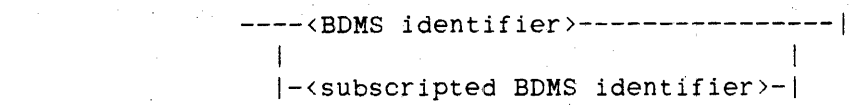
<input mapping>



<input assignment>



<alpha item name>
<Boolean item name>
<count item name>
<field item name>
<group item name>
<numeric item name>
<population item name>
<real item name>
<record type item name>



See also

Semantics

An arithmetic variable can be an integer, real, or double simple or subscripted variable. A Boolean variable can be a Boolean simple or subscripted variable. A pointer variable can be a pointer identifier or an element of a character array.

<arithmetic variable> := <field item name>

If the field item is defined to contain N bits, then N bits are stored right-justified in the arithmetic variable. All other bits are set to zero.

<arithmetic variable> := <numeric item name>
<arithmetic variable> := <real item name>

The numeric item or real item is converted into a binary value with a scale factor of zero (its true value). The value is stored in the arithmetic variable as in a normal arithmetic assignment; that is, it is integerized or extended, if necessary. An error termination results if necessary integerization is not possible, as in normal ALGOL arithmetic assignments.

<arithmetic variable> := <count item name>
<arithmetic variable> := <population item name>
<arithmetic variable> := <record type item name>

The value of the count item, population item, or record type item is placed in the arithmetic variable. Use of a count item, population item, or record type item allows read-only access to the particular field. Those items cannot be changed directly. They are accessed only through input mappings, and cannot be used in output mappings.

<Boolean variable> := <Boolean item name>

The Boolean variable is assigned the truth value (the value of bit 0) of the Boolean item. Bits 1 through 47 of the Boolean variable are set to zero.

<pointer variable> := <alpha item name>
<pointer variable> := <group item name>

If the alpha item or group item is defined to contain N EBCDIC characters, then N characters are transferred to the location pointed to by the pointer variable. A fault results if one of the following conditions is satisfied:

1. The pointer is uninitialized.
2. The pointer is not an EBCDIC (8-bit) pointer.
3. Fewer than N character positions remain in the referenced array.

A group item is treated as if it were an alpha item; all subordinate data items are transferred without change.

<pointer variable> := <numeric item name>

This assignment takes advantage of the fact that a numeric item is maintained as a hexadecimal string. If the numeric item is defined to contain N digits (including the sign digit, if specified), the N hexadecimal characters are transferred to the location pointed to by the pointer variable. A fault results if one of the following conditions is satisfied:

1. The pointer is uninitialized.

2. The pointer is not a hexadecimal (4-bit) pointer.

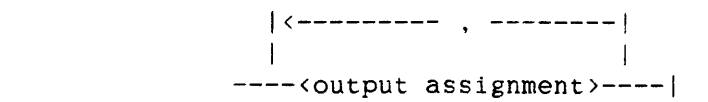
3. Fewer than N hexadecimal character positions remain in the referenced array.

OUTPUT MAPPING

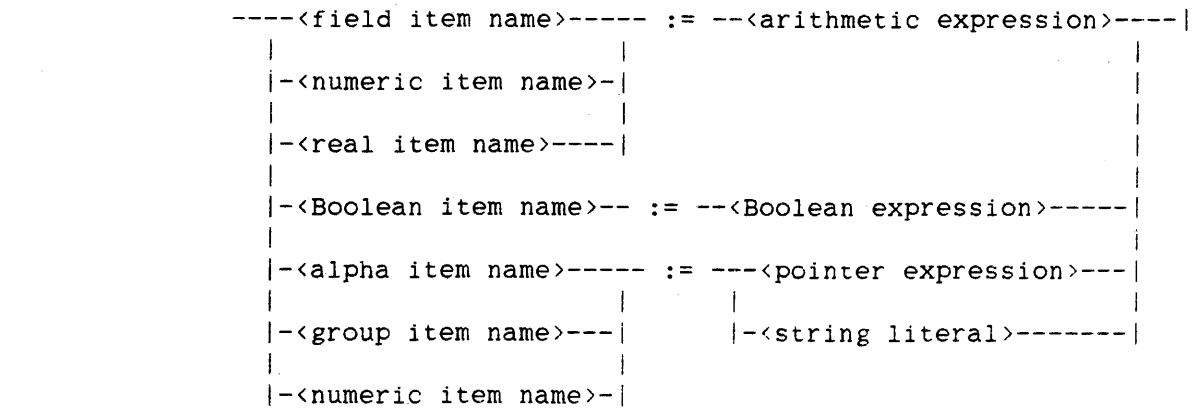
Output mappings can be used with the "storage" statements PUT and STORE. Output mappings transfer the value of a program variable or expression to a DASDL-declared data item. If the data item is an occurring item (that is, if the item is declared in DASDL with an OCCURS clause), it must be subscripted appropriately.

Syntax

<output mapping>



<output assignment>



See also

Semantics

An arithmetic expression used in an output mapping can be single precision or double precision.

<field item name> := <arithmetic expression>

If the field item is defined to contain N bits, then the N rightmost bits of the value of the arithmetic expression are assigned, unaltered, to the field item. Care should be taken if the arithmetic value is real or double precision, (that is, not integer) because the value might be normalized. in which case the N rightmost bits would not contain the value.

<numeric item name> := <arithmetic expression>
<real item name> := <arithmetic expression>

The value of the arithmetic expression is scaled appropriately and assigned to the numeric item or real item. If the numeric item or real item is unsigned, the absolute value of the arithmetic expression is used.

<Boolean item name> := <Boolean expression>

The truth value (the value of bit 0) of the Boolean expression is assigned to the Boolean item. Bits 1 through 47 of the value of the Boolean expression are ignored.

<alpha item name> := <pointer expression>
<group item name> := <pointer expression>

If the alpha item or group item is defined to contain N EBCDIC characters, then N characters are transferred from the location pointed to by the pointer expression to the alpha or group item. A fault results if any of the following conditions is satisfied:

1. The value of the pointer expression is an uninitialized pointer.

2. The value of the pointer expression is not an EBCDIC (8-bit) pointer.

3. Fewer than N character positions remain in the referenced array.

<numeric item name> := <pointer expression>

This mapping takes advantage of the fact that a numeric item is maintained as a hexadecimal string. If the numeric item is defined to contain N digits (including the sign digit, if specified), then N hexadecimal characters are transferred to the numeric item from the location pointed to by the pointer expression. The user is responsible for ensuring that the string is a valid representation of the item declared in DASDL; that is, the proper sign and numeric characters, in the proper format, must be used. A fault results if any of the following conditions is true:

1. The value of the pointer expression is an uninitialized pointer.

2. The value of the pointer expression is not a hexadecimal (4-bit) pointer.

3. Fewer than N hexadecimal character positions remain in the referenced array.

<alpha item name> := <string literal>
<group item name> := <string literal>

The string literal is transferred to the alpha item or group item. The string literal must be EBCDIC, or a syntax error results. If the string literal is shorter than the alpha item or group item, it is extended with blank fill characters on the right. If the string literal is longer than the alpha item or group item, the excess characters on the right are truncated.

<numeric item name> := <string literal>

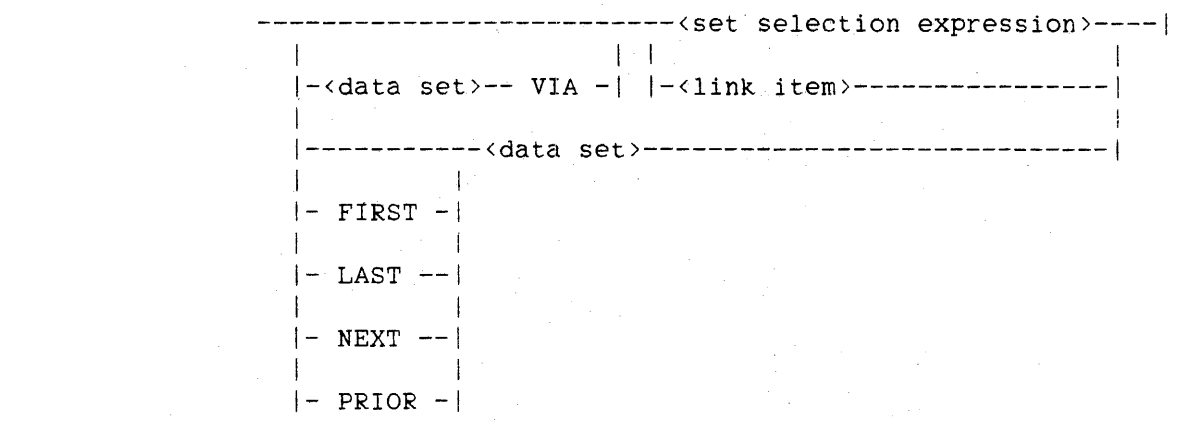
The string literal is transferred to the numeric item. The string literal must be a hexadecimal string and must contain the exact number of characters for the numeric item or a syntax error results. The user is responsible for ensuring that the string literal is a valid representation of the numeric item.

9.2.3 THE SELECTION EXPRESSION

A selection expression is used in DELETE, FIND, BDMS "LOCK," and MODIFY statements to identify a particular record in a data set.

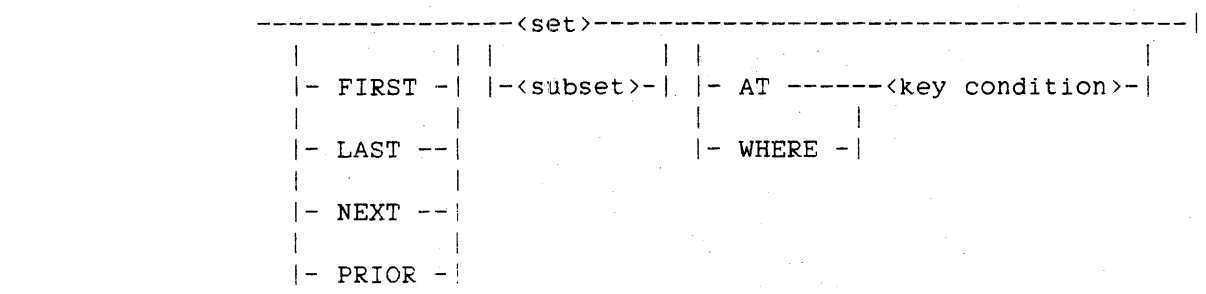
Syntax

<selection expression>

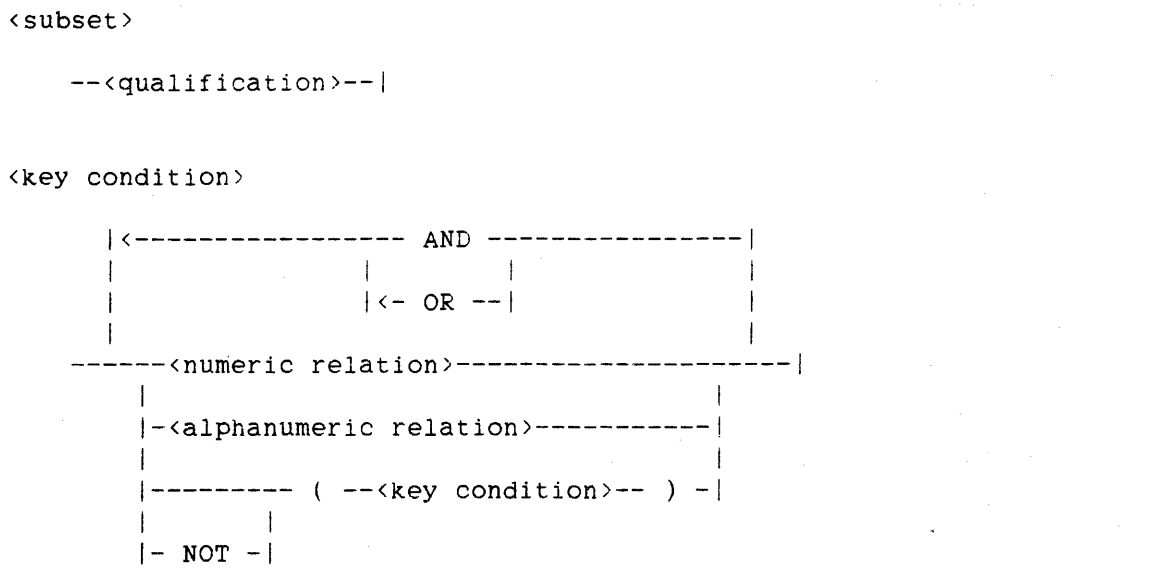


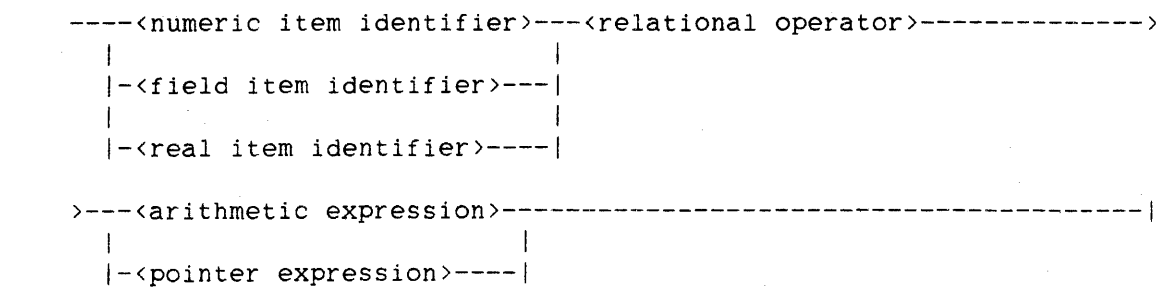
<data set>

<set selection expression>

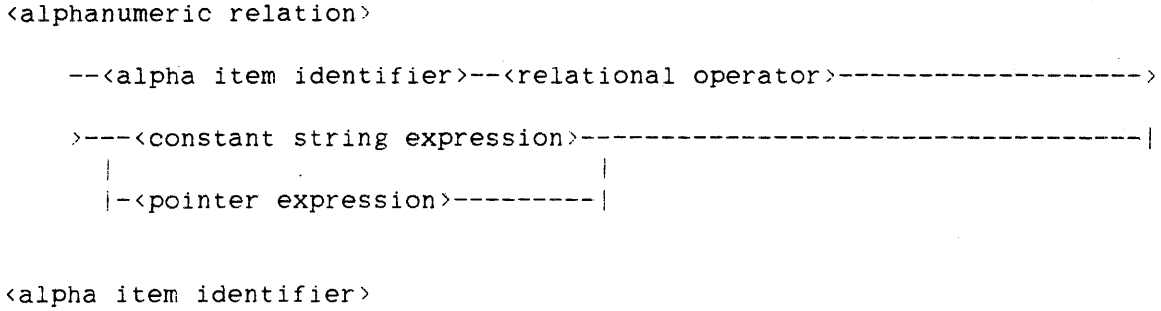


<set>


<subset>
--<qualification>--|
<key condition>
|<---------- AND ----------|
	<- OR --	
------<numeric relation>----------|
-<alphanumeric relation>----------
--------- (--<key condition>--) -
- NOT -


<numeric relation>
----<numeric item identifier>---<relational operator>---------->
| |
|-<field item identifier>---|
| |
|-<real item identifier>----|
>---<arithmetic expression>----------|
| |
|-<pointer expression>----|


<numeric item identifier>
<field item identifier>
<real item identifier>
--<BDMS identifier>--|


<alphanumeric relation>
--<alpha item identifier>--<relational operator>---------->
>---<constant string expression>----------|
| |
|-<pointer expression>---------|


<alpha item identifier>
--<BDMS identifier>--|

<link item>

--<qualification>--|

See also

<BDMS identifier> 691
<constant string expression>. 525
<qualification> 693
<relational operator> 493

Semantics

A set selection expression selects the record to which the set path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The construct "<data set> VIA" identifies the record area and current path to be affected if the desired record is found. This option is used for link items and for sets that are not implicitly associated with the data set.

The <link item> form is used to specify a link item defined in the DASDL description. The record to which the link item refers is selected. An exception is returned if the link item is NULL.

The <data set> form is used to select the record to which the data set path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The word "FIRST" selects the first record in the specified data set, set, or subset. If a key condition is also specified, the first record of the specified set or subset that satisfies the key condition is selected. FIRST is assumed by default.

The word "LAST" selects the last record in the specified data set, set, or subset. If a key condition is also specified, the last record of the specified set or subset that satisfies the key condition is selected.

The word "NEXT" selects the next record relative to either the set path (if a set or subset is specified) or the data set path (if a data set is specified). If a key condition is also specified, the next record (relative to the current path) of the specified set or subset that satisfies the key condition is selected.

The word "PRIOR" selects the prior record relative to either the set path (if a set or subset is specified) or the data set path (if a data set is specified). If a key condition is also specified, the prior record (relative to the current path) of the specified set or subset that satisfies the key condition is selected.

In a set selection expression, the <set> or <subset> construct selects the record to which the set or subset path refers. A NOTFOUND exception is returned if the record has been deleted or if the path does not refer to a valid current record.

The words "AT" or "WHERE" indicate that a key condition follows. AT and WHERE are synonyms.

A key condition specifies values used to locate specific records in a data set referenced by a particular set or subset. If the name of a data item specified in a key condition is not unique, the compiler provides implicit qualification through the set or subset of the set selection expression. Qualification of the item name by the name of the data set that contains the item, while not necessary, is allowed; however, the compiler handles this qualification as documentation only.

The expressions that appear in a key condition cannot contain any transaction item references.

A numeric relation specifies a particular numeric, field, or real item and compares it to the value of an arithmetic expression or a pointer expression. The pointer expression must evaluate to a hexadecimal pointer.

An alphanumeric relation specifies a particular alpha item and compares it to the value of a constant string expression or a pointer expression. The pointer expression must evaluate to an EBCDIC pointer. The constant string expression must be an EBCDIC string.

Examples

These examples use the database described in DASDL by the following:

```
D DATA SET (
   A ALPHA (3);
   N NUMBER (5);
   );
S SET OF D KEY IS N, DATA A;
```

```
LOCK S WHERE N NEQ 10
```

This LOCK statement acts upon the first S where the value of N is not equal to 10.

```
FIND S AT A = "ABC" AND (N = 50 OR N = 90)
```

This statement locates the first S where A is equal to the string "ABC" and either N is equal to 50 or N is equal to 90.

9.3 BDMSALGOL STATEMENTS

The following data management statements allow a BDMSALGOL program to use and manipulate the data in a database. These statements are described in this section.

<assign statement>
<begintransaction statement>
<BDMS close statement>
<create statement>
<delete statement>
<dmterminate statement>
<endtransaction statement>
<find statement>
<BDMS free statement>
<generate statement>
<get statement>
<insert statement>
<BDMS lock statement>
<midtransaction statement>
<modify statement>
<BDMS open statement>
<put statement>
<recreate statement>
<remove statement>
<BDMS set statement>
<store statement>

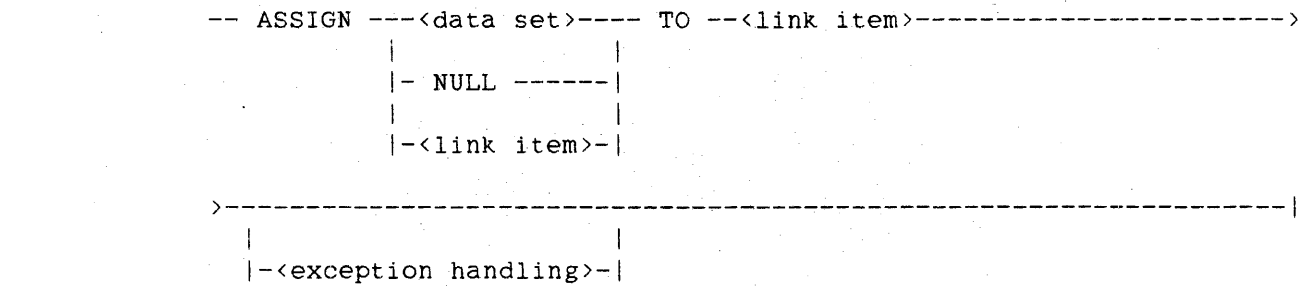
ASSIGN STATEMENT

The ASSIGN statement establishes a link from one record in a data set to another record of the same data set. It assigns either the value of the current record in a data set or the value in a link item to another link item. The value of the second link item, called the target link item, then allows the system to locate the record in the referenced data set.

The ASSIGN statement is effective immediately, therefore the record containing the target link item does not need to be stored unless data items of this record have been modified.

Syntax

<assign statement>



See also
<data set> 703
<exception handling> 768
<link item> 705

Semantics

The data set must be declared in DASDL as the object data set of the target link item. A value that points to the current record in the data set is assigned to that link item.

If the <data set> form is used, the current path of the specified data set must be valid, but the record need not be locked. If the data set path is not valid, an exception occurs.

If the word "NULL" is used, the relationship between records is severed by assigning a NULL value to the target link item. If that link item is already NULL, this option is ignored. A FIND, BDMS "LOCK," or MODIFY statement on a NULL link item results in an exception.

If the ASSIGN statement specifies two link items, the value of the first link item is assigned to the target link item. The first link item must be declared in DASDL to have the same object data set as the target link item and be the same type of link (counted link, self-correcting link, symbolic link, unprotected link, or verified link). If the link items are counted links, the count item is automatically updated, even if the record that is referenced is locked by another program.

The current path of the data set containing the first link must be valid, but the record need not be locked. If the data set path is not valid, an exception occurs.

After the ASSIGN statement has executed, the target link item points to either the current record in the specified data set or to the record pointed to by the first link item.

The current path of the data set containing the target link item must be valid, and the record must be locked; otherwise, an exception occurs.

If the target link item references a disjoint data set, then that link item can point to any record in the data set. If the target link item references an embedded data set, then only certain records in the data set can be referenced. In this case, the record being referenced must be owned by the record containing the target link item or by an ancestor of the record containing this link item. (An ancestor is the owner of the record, the owner of the owner, and so forth.)

If an exception is returned, the ASSIGN statement is not completed, and a NULL value is assigned to the target link item.

Example

If the database EXAMPLEDB is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   L IS IN E VERIFY ON N;
   ):
S SET OF D KEY A;

E DATA SET (
   N NUMBER (3);
   R REAL;
   ):
T SET OF E KEY N;
```

then the following BDMSALGOL program uses the ASSIGN statement to assign the value of the current record of data set E to link item L.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE EXAMPLEDB;
   EBCDIC ARRAY X[0:2];
   INTEGER Y;

   OPEN UPDATE EXAMPLEDB;
   WHILE NOT READ(CARD_FILE,<A3,I3>,X,Y) DO
      BEGIN
      FIND S AT A = X;
      FIND T AT N = Y;
      ASSIGN E TO L;
      END;
   CLOSE EXAMPLEDB;
END.
```

BEGINTRANSACTION STATEMENT

The BEGINTRANSACTION statement places a program in transaction state. This statement can be used only with audited databases.

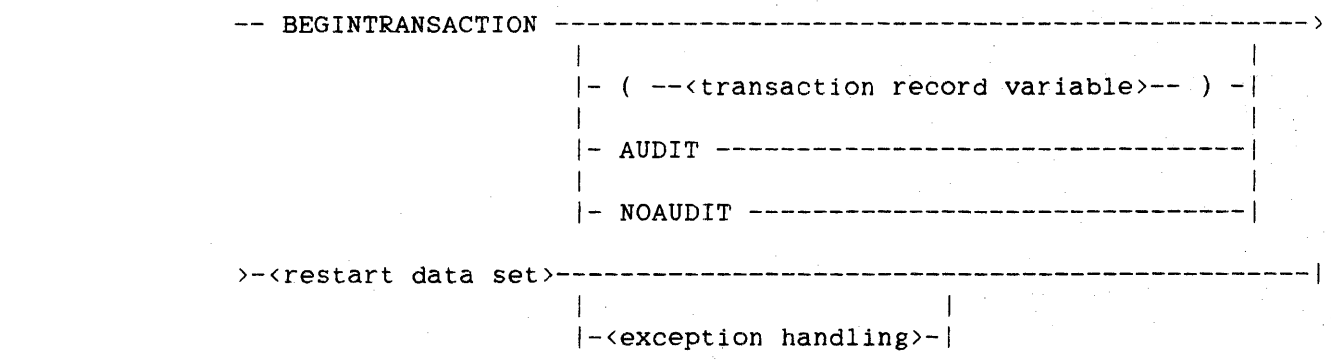
The BEGINTRANSACTION statement performs the following steps in order:

1. Captures the restart data set if AUDIT is specified
2. Places a program in transaction state

Refer to the "DMSII User Language Interface Software Operation Guide" for further details regarding audit and recovery.

Syntax

<begintransaction statement>



<transaction record variable>

The <transaction record variable> syntax is defined in the "DMSII Transaction Processing System (TPS) Programmer's Manual."

<restart data set>

See also

<exception handling>. 768
<qualification> 693

Semantics

If the ‹transaction record variable› construct is used, it is the formal input transaction record variable, and NOAUDIT is the default action.

The word "AUDIT" causes the restart area to be captured. The path of the specified restart data set is not altered when the restart record is stored. AUDIT is the default action.

The word "NOAUDIT" causes the restart area to not be captured. The ‹restart data set› construct specifies the restart data set to be updated.

An exception is returned if the BEGINTRANSACTION statement is attempted while the program is in transaction state.

If any exception is returned, the program is not placed in transaction state. If an ABORT exception is returned, all records that the program had locked are freed.

Pragmatics

Deadlock can occur during execution of a BEGINTRANSACTION statement.

Any attempt to modify an audited database when the program is not in transaction state results in a fault. The BDMSALGOL statements that modify databases are the following:

ASSIGN statement
DELETE statement
GENERATE statement
INSERT statement
REMOVE statement
STORE statement

Example

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the BEGINTRANSACTION statement can be used.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      MY_N := * + 1;
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

BDMS CLOSE STATEMENT

The BDMS "CLOSE" statement closes a database when further access is no longer required.

The CLOSE statement performs the following steps in order:

1. Closes the database
2. Frees all locked records

Syntax

<BDMS close statement>

```
-- CLOSE --<database identifier>--------------------------|
                                  |                        |
                                  |-<exception handling>-|
```

<database identifier>

```
--<BDMS identifier>--|
```

See also
<BDMS identifier> 691
<exception handling>. 768

Semantics

The database identifier specifies the database to be closed. If the database was declared to have an internal name, this internal name is the database identifier. If the database does not have an internal name but is a logical database, then the logical database name is the database identifier. If the database does not have an internal name and it is not a logical database, then the database name is the database identifier.

An exception is returned if the specified database is not open.

Pragmatics

Use of the CLOSE statement is optional; the system closes any open database at the time a program terminates.

The CLOSE statement is the only BDMSALGOL statement in which the status word has meaning when no exception is indicated. Therefore, after a CLOSE statement, the status word should be examined by the program and appropriate action taken, whether or not an exception is returned. An ABORT exception can be obtained in this manner.

Example

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how to use the CLOSE statement to close DBASE.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := N)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

CREATE STATEMENT

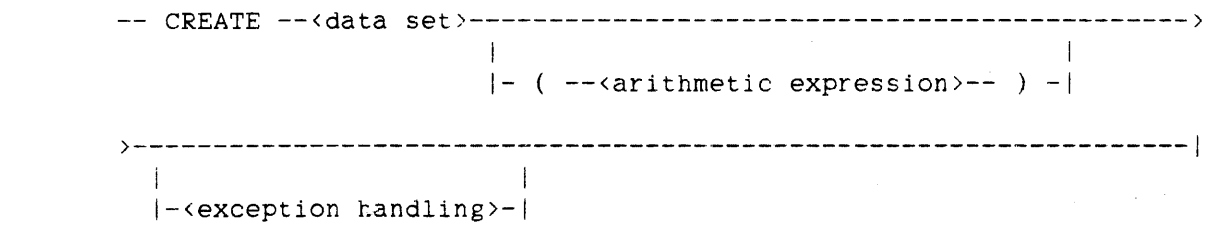
The CREATE statement initializes the user work area of a data set record.

The CREATE statement performs the following steps in order:

1. Frees the current record of the specified data set
2. Reads any specified expression to determine the format of the record to be created
3. Initializes data items to one of the following values:
   a. The DASDL-declared INITIALVALUE, if present
   b. The DASDL-declared NULL, if present
   c. The default NULL

Syntax

<create statement>



See also
<data set>. 703
<exception handling>. 768

Semantics

The <data set> construct specifies the data set to be initialized. The current path of the data set is not changed until a subsequent STORE statement has completed successfully.

The arithmetic expression specifies the type of record to be created. This arithmetic expression is required when a variable-format record is created; otherwise, it must not appear.

An exception is returned if the arithmetic expression does not represent a valid record type.

Pragmatics

Normally, the CREATE statement is eventually followed by a STORE statement, which places the newly created record into the data set. However, if a subsequent STORE operation is not desired, the CREATE statement can be nullified by a subsequent CREATE, DELETE, FIND, BDMS "FREE." BDMS "LOCK," MODIFY, or RECREATE statement.

The CREATE statement sets up only a record area. If the record contains embedded structures, the master record must be stored before entries can be created in the embedded structures. If only entries in the embedded structure are created (that is, if items in the master are not altered), the master need not be stored a second time.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program shows how a record of data set D can be created and stored.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY X[0:9];
   INTEGER Y,Z;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A10,I1,I3>,X[0],Y,Z) DO
      BEGIN
      CREATE D;
      PUT D (A := X[0]);
      IF Y = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := Z);
      STORE D;
      END;
   CLOSE DBASE;
END.
```

DELETE STATEMENT

The DELETE statement is identical to the FIND statement except that if a record is found, it is locked and then deleted.

The DELETE statement performs the following steps in order:

1. Frees the current record, unless the selection expression is the name of the data set and the current record is locked. In that case, the locked status is not altered.

2. Alters the current path to point to the record specified by the selection expression, and locks this record.

3. Transfers that record to the user work area.

4. Removes the record from all sets and automatic subsets, but not from manual subsets.

5. Removes the record from the data set.

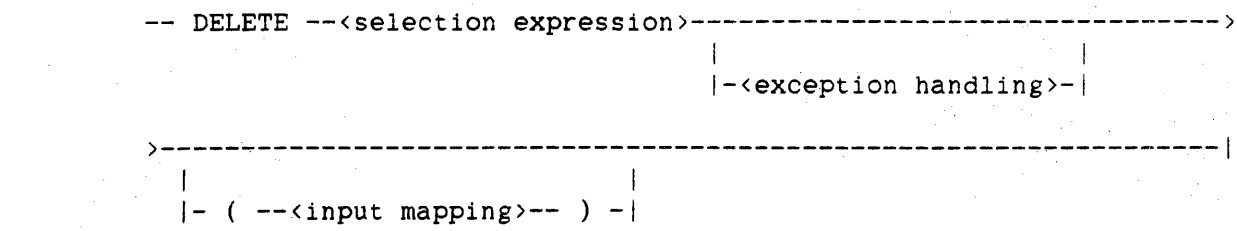
If the record is found but cannot be deleted, an exception is returned and the DELETE statement terminates, leaving the current path pointing to the record specified by the selection expression.

If a set selection expression is used and the record is not found, then an exception is returned and the set path is changed and invalid. It refers to a location between the last key less than the condition and the first key greater than the condition. A set selection expression using NEXT or PRIOR can be done from this point provided keys greater than and less than the condition exist. The current path of the data set, the current record, and the current paths of any other sets for that data set remain unchanged.

It is the responsibility of the programmer to ensure that no manual subset refers to the record being deleted.

Syntax

<delete statement>



See also

Semantics

The selection expression identifies the record to be deleted.

An exception is returned and the record is not deleted if the record has counted links pointing to it, or if the record contains a non-NULL link or a non-empty embedded structure.

Pragmatics

When the DELETE statement completes, the current paths still refer to the deleted record. Therefore, a FIND statement on the current record results in a NOTFOUND exception; however, "FIND NEXT" and "FIND PRIOR" statements are still appropriate.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates the use of the DELETE statement to delete a record of the data set D where item N is equal to the value of X.

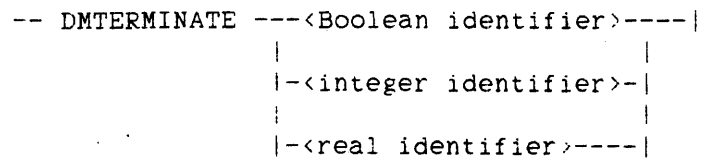
```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      DELETE S AT N = X;
   CLOSE DBASE;
END.
```

DMTERMINATE STATEMENT

The DMTERMINATE statement aborts the current action. When an exception occurs that the program does not handle, the DMTERMINATE statement can be called to produce the same results as if the exception-handling syntax had not been specified in the statement; that is, the DMTERMINATE statement causes the program to terminate with a fault.

Syntax

<dmterminate statement>



See also

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program shows an example of the use of the DMTERMINATE statement.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   BOOLEAN RSLT;
   REAL RRSLT = RSLT;
   INTEGER X;

   OPEN UPDATE DBASE;
   FIND FIRST D :RSLT;
   IF RSLT THEN
      BEGIN
      DISPLAY("D IS EMPTY DATA SET");
      DMTERMINATE(RSLT);
      END
   ELSE
      WHILE NOT READ(CARD_FILE,<I3>,X) DO
         BEGIN
         DELETE S AT N = X :RSLT;
         IF RRSLT.DMERROR = NOTFOUND THEN
            DMTERMINATE(RSLT);
         END;
   CLOSE DBASE;
END.
```

ENDTRANSACTION STATEMENT

The ENDTRANSACTION statement takes a program out of transaction state. This statement can be used only with audited databases.

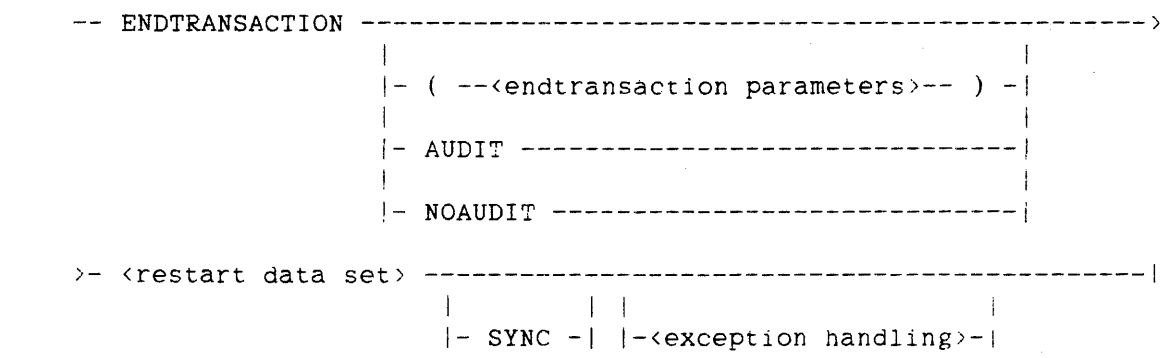
The ENDTRANSACTION statement performs the following steps in order:

1. Captures the restart area if AUDIT is specified
2. Forces a syncpoint if the SYNC option is specified
3. Implicitly frees all records of the database that the program has locked

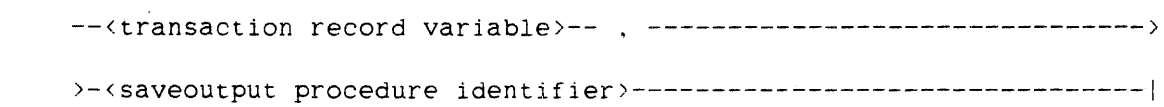
Refer to the "DMSII User Language Interface Software Operation Guide" for information regarding audit and recovery.

Syntax

<endtransaction statement>



<endtransaction parameters>



<saveoutput procedure identifier>

See also

Semantics

If the <endtransaction parameters> form is used, the transaction record variable is the formal input transaction record variable. The saveoutput procedure identifier is the name of the SAVEOUTPUT formal procedure. For more information about the SAVEOUTPUT procedure, refer to the "DMSII Transaction Processing System (TPS) Programmer's Manual."

The word "AUDIT" causes the restart area to be captured. The path of the restart data set is not altered when the restart record is stored.

The word "NOAUDIT" causes the restart area to not be captured. NOAUDIT is the default action.

The <restart data set> construct specifies the restart data set to be used.

The word "SYNC" forces a syncpoint.

An exception is returned if an ENDTRANSACTION statement is attempted and the program is not in transaction state.

Records are freed in all cases. If an exception occurs, the transaction is not applied to the database.

Example

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the ENDTRANSACTION statement can be used.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R:
      MY_N := * + 1:
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

FIND STATEMENT

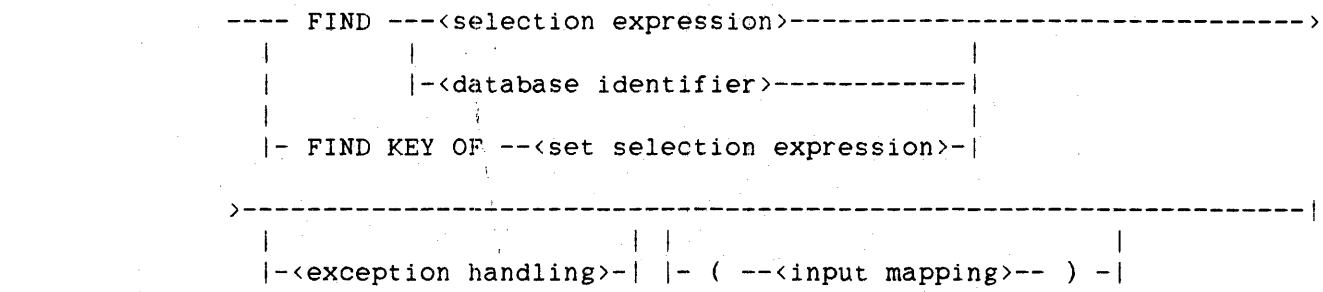
The FIND statement transfers a record to the user work area associated with a data set or global data.

The FIND statement performs the following steps in order:

1. Frees a locked record in the data set if a data set is specified in the FIND statement, or frees a locked record in the associated data set if a set is specified in the FIND statement
2. Alters the current path to point to the record specified by the selection expression or database name
3. Transfers that record to the user work area

Syntax

<find statement>



See also

Semantics

The <selection expression> form is used to specify the record to be transferred to the user work area.

The <database identifier> form is used to specify the global data record to be transferred to the user work area associated with the global data. If no global data was described in DASDL for the database, a syntax error occurs.

The form "FIND KEY OF <set selection expression>" moves the key and any associated data (as specified in DASDL) from the key entry to the user work area. A physical read is not performed on the data set; consequently, all items in the record area that do not appear in the key entry retain whatever value they had before the FIND statement. The current path of the data set is not affected.

If an exception is returned, the record is not freed.

An exception is returned if no record satisfies the selection expression.

If a set selection expression is used and the record is not found, then an exception is returned and the set path is changed and invalid. It refers to a location between the last key less than the condition and the first key greater than the condition. A set selection expression using NEXT or PRIOR can be done from this point provided keys greater than and less than the condition exist. The current path of the data set, the current record, and the current paths of any other sets for that data set remain unchanged.

To access data items, input mapping is required.

Examples

```
FIND FIRST EMP AT DEPT-NO = 1019 :RSLT;
IF RSLT THEN
   POP-EMPS[1019] := 0;


FIND EMP AT EMP-NO = SSN :RSLT;
IF RSLT THEN
   ERR_OUT(INV_EMP_NO_ERR);


FIND NEXT EMP :RSLT;
IF RSLT THEN
   GO NO_MORE_EMP;


FIND FIRST OVR-65 AT DEPT-NO = 1019 :RSLT:
IF RSLT THEN
   POP-OVR-65[1019] := 0;
```

BDMS FREE STATEMENT

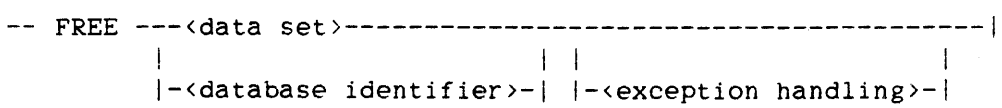
The BDMS "FREE" statement unlocks the current record.

A FREE statement can be executed after any operation. If the current record is already free, or if no current record is present, the FREE statement is ignored.

The FREE statement can be used to unlock a record that the user anticipates cannot be implicitly freed for a relatively long time. A FREE statement executed on a record allows other programs to lock the record.

Syntax

<BDMS free statement>



See also

Semantics

The <data set> form is used to specify the data set whose current record is to be unlocked. The data set path and current record area remain unchanged.

The <database identifier> form is used to specify the global data record to be unlocked. The data set path and current record area remain unchanged.

If an exception is returned, the state of the database remains unchanged.

Pragmatics

The FREE statement is optional in many situations because DELETE, FIND, BDMS "LOCK," and MODIFY statements can free a record before they execute. FIND, LOCK, and MODIFY statements that use sets or subsets can free the locked record only if a new record is successfully retrieved. Otherwise, the previously locked record remains locked. In general, an implicit FREE statement is performed, if necessary, during any operation that establishes a new data set path.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates the use of the FREE statement to unlock the current record of data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         DELETE D
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

GENERATE STATEMENT

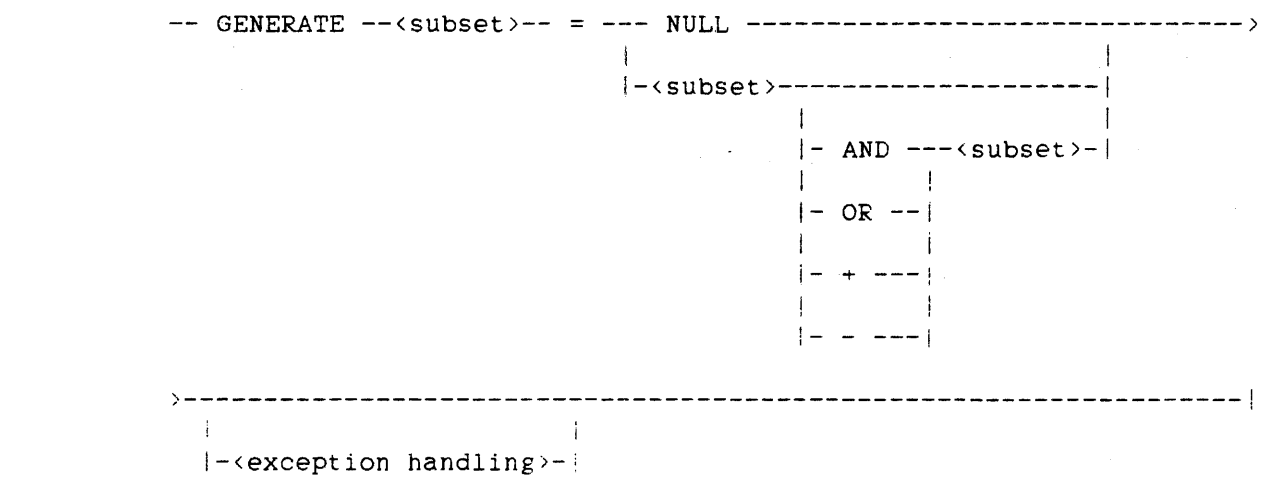
The GENERATE statement creates an entire subset in one operation. All subsets must be disjoint bit vectors.

The GENERATE statement performs the following steps in order:

1. Deletes all the records from the subset to be generated if it is not already empty
2. Assigns either a NULL value, the records in another subset, or a combination of the records in two other subsets to the subset that is generated

Syntax

<generate statement>



See also
<exception handling>. 768
<subset>. 704

Semantics

The <subset> to the left of the equal sign (=) is the name of the subset to be generated. This subset must be a manual subset, which must be a disjoint bit vector.

The word "NULL" assigns a NULL value to the generated subset.

If <subset> follows the equal sign, it is the name of the subset whose records are to be assigned to the generated subset. This subset must be of the same data set as the generated subset. and it must be a disjoint bit vector.

If to the right of the equal sign there are two <subset>s joined by the operation AND, OR, +, or -, then these two subsets are to be combined in the specified manner. The result is then assigned to the generated subset. The two subsets must be of the same data set, and must be disjoint bit vectors.

The operator "AND" specifies that the intersection of the two subsets is to be assigned to the generated subset. The intersection is defined to be all the records in the first subset that are also in the second subset.

The operator "OR" specifies that the union of the two subsets is to be assigned to the generated subset. The union is defined to be all the records that are in either the first subset or the second subset.

The operator "+" specifies that the exclusive OR of the two subsets is to be assigned to the generated subset. The exclusive OR consists of the records in either the first subset or the second subset, but not the records that appear in both subsets.

The operator "-" specifies that the subset difference of the two subsets is to be assigned to the generated subset. The subset difference is defined to be the records in the first subset that are not in the second subset.

Example

If the database DBASE is described in DASDL as the following:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN:
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D WHERE (N GEQ 21 AND NOT B) BIT VECTOR;
Y SUBSET OF D WHERE (R LSS 1000) BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how the GENERATE statement can be used to assign all the records that are in both X and Y to subset Z.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY S[0:2];
   INTEGER T,U,V;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A3,I1,I3,I4>,S,T,U,V) DO
      BEGIN
      CREATE D;
      PUT D (A := S);
      IF T = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := U);
      PUT D (R := V):
      STORE D;
      END;
   GENERATE Z = X AND Y;
   CLOSE DBASE;
END.
```

GET STATEMENT

The GET statement is used to transfer information from the user work area associated with a data set or global data record into program variables or arrays.

The GET statement does not access the database; it assumes that prior database operations have loaded the proper record or data items into the user work area.

Syntax

<get statement>

```
-- GET ---<data set>-------------- ( --<input mapping>-- ) --|
         |                        |
         |-<database identifier>-|
```

See also
<data set>. 703
<database identifier> 715
<input mapping> 696

Semantics

The <data set> construct is used to transfer information from the user work area associated with this data set into a program variable or array.

The <database identifier> is used to transfer information from the user work area associated with the global data record into a program variable or array.

Pragmatics

No exceptions are associated with the GET statement. However, if the database containing the referenced data set or global data record has not been opened at the time execution of the GET statement is attempted, the program terminates with a fault.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN:
   N NUMBER (3):
   R REAL;
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the GET statement can be used to assign current values of data items to program variables and arrays.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := R)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

INSERT STATEMENT

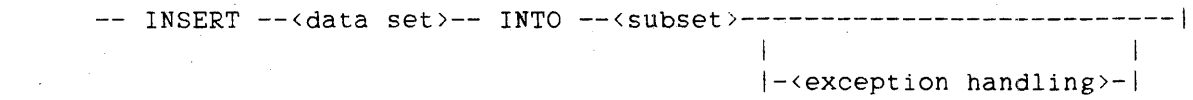
The INSERT statement places a record into a manual subset.

The INSERT statement performs the following steps in order:

1. Inserts the current record of the specified data set into the specified subset
2. Alters the set path for the specified subset to point to the inserted record

Syntax

<insert statement>



See also
<data set>. 703
<exception handling>. 768
<subset>. 704

Semantics

The <data set> construct specifies the data set whose current record is inserted into the subset specified by <subset>. The path of the specified data set must be the object data set of the specified subset.

The path of the specified data set must refer to a valid record: if not, an exception is returned.

The subset must be a manual subset, and it must be a subset of the specified data set.

An exception is returned in the following cases:

1. If duplicates are not allowed for the specified subset and the record to be inserted has a key identical to that of a record currently in that subset

2. If the specified subset is embedded in a data set that does not have a valid current record

3. If "LOCK TO MODIFY DETAILS" was specified in DASDL and the current record is not locked

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program shows how the INSERT statement can be used to place the current record of data set D into subset X.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE:
   SET D TO BEGINNING;
   FIND NEXT D :RSLT:
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN > 10 THEN
         INSERT D INTO X;
      FIND NEXT D :RSLT;
      END;
   CLOSE DBASE;
END.
```

BDMS LOCK STATEMENT

The BDMS "LOCK" statement is similar to the FIND statement, except that if a record is found, it is locked against a concurrent modification by another user.

"LOCK" and "MODIFY" are synonyms.

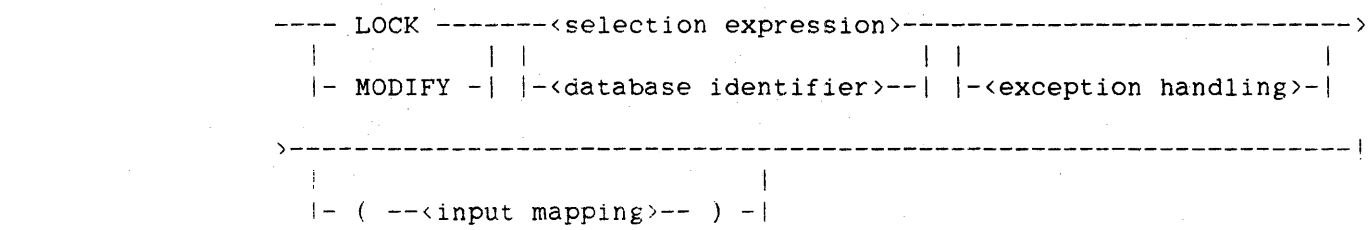
If the record to be locked has already been locked by another program, the system performs a contention analysis. In this case, the present program waits until the record is unlocked. However, if a wait would result in a deadlock, all records locked by the program with the lowest priority involved in the deadlock are unlocked, and the operation in that program terminates with a DEADLOCK exception.

The LOCK statement performs the following steps in order:

1. Frees a locked record in the data set if the LOCK statement specifies a data set, or frees a locked record in the associated data set if the LOCK statement specifies a set
2. Alters the current path to point to the record specified by the selection expression or database identifier
3. Locks the specified record and then transfers that record to the user work area

Syntax

<BDMS lock statement>



See also
<database identifier> 715
<exception handling>. 768
<input mapping> 696
<selection expression>. 703

Semantics

The selection expression is used to specify the record to be locked.

The database identifier is used to specify the global data record to be locked.

If an exception is returned, the record is not freed.

If a LOCK statement using a set selection expression returns an exception, the current path of the specified set is invalid. However, the current path of tne data set, the current record, and the current paths of any other sets for that data set remain unaltered.

To access data items, the <input mapping> construct must appear.

Pragmatics

Because no other user can lock a record once it is locked, a record must be freed when it is no longer required to be locked. A record can be freed explicitly by a BDMS "FREE" statement or implicitly by a subsequent CREATE, DELETE, FIND, BDMS "LOCK," or RECREATE statement on the same data set.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the LOCK statement to lock records of subset X.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET X TO BEGINNING;
   LOCK NEXT X :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN <= 10 THEN
         BEGIN
         REMOVE D FROM X;
         DELETE D;
         END
      ELSE
         BEGIN
         PUT D (B := TRUE);
         STORE D;
         END;
      LOCK NEXT X :RSLT;
      END;
   CLOSE DBASE;
END.
```

MIDTRANSACTION STATEMENT

The MIDTRANSACTION statement causes the compiler to generate calls on the given procedure immediately before the call on the DMS procedure in the Accessroutines.

Refer to the "DMSII Transaction Processing System (TPS) Programmer's Manual" for more information on transaction processing.

Syntax

<midtransaction statement>

```
    -- MIDTRANSACTION -- ( --<midtransaction parameters>-- ) ---------->

    >-<restart data set>-------------------------------------------------|
                         |                         |
                         |-<exception handling>-|
```

<midtransaction parameters>

```
    --<transaction record variable>-- , ------------------------------>

    >-<saveinput procedure identifier>--------------------------------|
```

<saveinput procedure identifier>

```
    --<procedure identifier>--|
```

See also

Semantics

The transaction record variable is the formal input transaction record variable.

The saveinput procedure identifier is the name of the SAVEINPUT formal procedure.

The <restart data set> construct specifies the restart data set to be used.

Example

```
MIDTRANSACTION (TRIN,SAVEINPUT) RDS :RSLT;
```

MODIFY STATEMENT

The MODIFY statement is described [redacted] section. "MODIFY" and "LOCK" are synonyms.

BDMS OPEN STATEMENT

The BDMS "OPEN" statement opens a database for subsequent access and specifies the access mode.

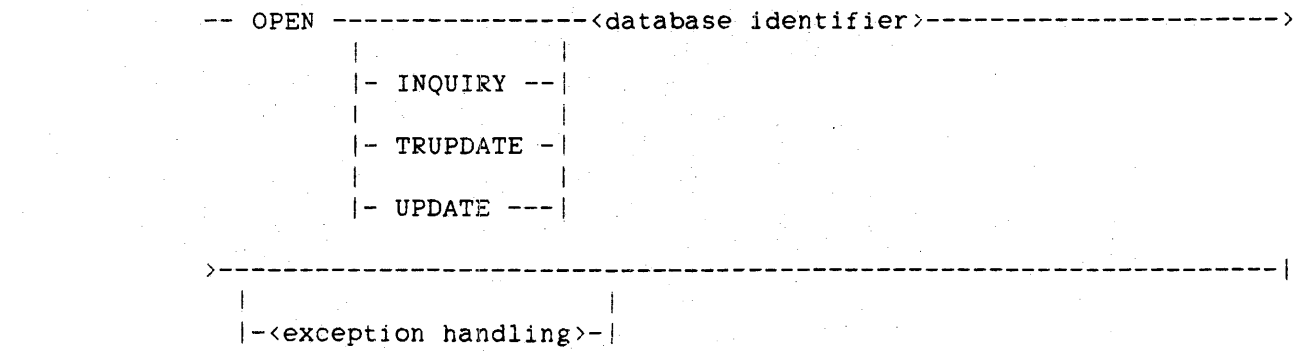
The OPEN statement performs the following steps in order:

1. Opens an existing database. Appropriate "NO FILE" messages are displayed if files required for invoked structures are not present in the system directory.

2. Performs an implicit CREATE statement on the restart data set.

Syntax

<BDMS open statement>



See also
<database identifier> 715
<exception handling>. 768

Semantics

The word "INQUIRY" enforces read-only access to the database. This option is specified when no update operations are to be performed on the database. An exception is returned if the following BDMSALGOL statements are used when the database has been opened with the INQUIRY option:

ASSIGN statement	GENERATE statement
BEGINTRANSACTION statement	INSERT statement
DELETE statement	REMOVE statement
ENDTRANSACTION statement	STORE statement

The data management system does not open any audit files if the "OPEN INQUIRY" form has been used by all programs accessing the database.

The word "UPDATE" allows the program to modify the database being opened. The UPDATE option must be specified in order to use the BDMSALGOL statements listed above under the INQUIRY option. UPDATE is the default option.

The word "TRUPDATE" must be specified in order to use the MIDTRANSACTION statement or the <transaction record variable> form of the BEGINTRANSACTION or ENDTRANSACTION statements.

The database identifier specifies the database to be opened.

If an exception is returned, the state of the database remains unchanged.

An exception is returned if the database is already open.

Pragmatics

An OPEN statement must be executed before the first access of the database; otherwise, the program terminates with a fault.

Examples

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the OPEN statement with the INQUIRY option to open database DBASE and perform read-only actions on the database.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];

   OPEN INQUIRY DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      GET D (MA[0] := A,MB := B);
      IF MB THEN
         GET D (MR := N)
      ELSE
         MR := 0;
      WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
            MN,MA[0],MB,MR);
      END;
   CLOSE DBASE;
END.
```

The following BDMSALGOL program demonstrates the use of the OPEN statement with the UPDATE option to open database DBASE and perform update actions on the database.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         DELETE D
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

PUT STATEMENT

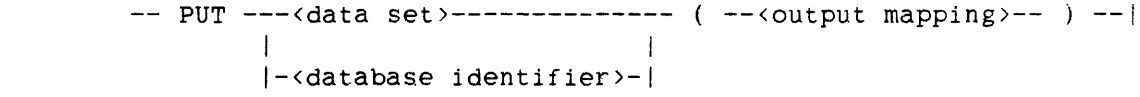
The PUT statement transfers information from program expressions into the user work area associated with a data set or global data record.

The PUT statement does not update the database; a subsequent STORE statement must be executed to place the data in the user work area into the database.

Any number of PUT statements can be used to update items before a STORE statement is executed.

Syntax

<put statement>



See also

<data set> 703
<database identifier> 715
<output mapping> 700

Semantics

The <data set> form is used to transfer information associated with this data set into the user work area.

The <database identifier> form is used to transfer information associated with the global data record into the user work area.

Pragmatics

No exceptions are associated with the PUT statement. However, if the database containing the specified data set or the specified database has not been opened, the program terminates with a fault.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
X SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates how the PUT statement can be used to assign values to data items.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY S[0:2];
   INTEGER T,U,V;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A3,I1,I3,I4>,S[0],T,U,V) DO
      BEGIN
      CREATE D;
      PUT D (A := S);
      IF T = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := U,R := V);
      STORE D;
      END;
   CLOSE DBASE;
END.
```

RECREATE STATEMENT

The RECREATE statement partially initializes the user work area. All data items remain unaltered; however, control items such as links, sets, counts, and data sets are unconditionally set to NULL.

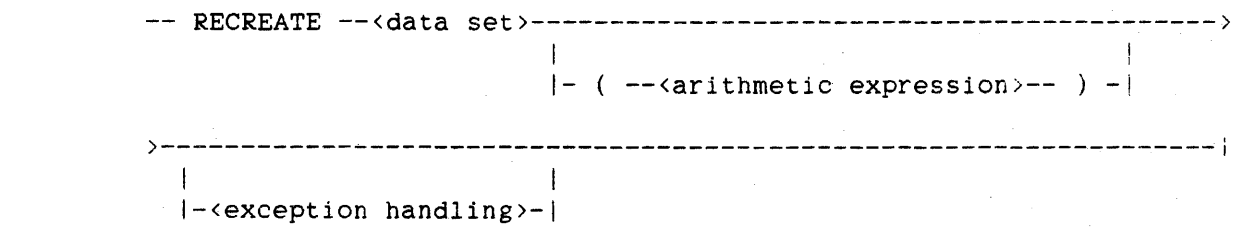
For variable-format records, the record type supplied must be the same as that supplied in the CREATE statement that created the record. If not, the subsequent STORE statement results in a DATAERROR subcategory 4.

The RECREATE statement performs the following steps in order:

1. Frees the current record of the specified data set
2. Reads any specified arithmetic expression to determine the format of the record to be created
3. Unconditionally sets links, sets, counts, and data sets to NULL

Syntax

<recreate statement>



See also
<data set>. 703
<exception handling>. 768

Semantics

The <data set> construct specifies the data set to be initialized.

The arithmetic expression specifies a value indicating the type of record to be created. This arithmetic expression is required when a variable-format record is created: otherwise, it must not appear.

An exception is returned if the arithmetic expression does not represent a valid record type.

Example

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (10);
   B BOOLEAN;
   N NUMBER (3);
   );
S SET OF D KEY N;
SS SUBSET OF D BIT VECTOR;
X SUBSET OF D BIT VECTOR;
Y SUBSET OF D BIT VECTOR;
Z SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates how the RECREATE statement can be used to partially initialize a record of data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY X[0:9];
   INTEGER Y,Z;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<A10,I1,I3>,X[0],Y,Z) DO
      BEGIN
      CREATE D;
      PUT D (A := X[0]);
      IF Y = 1 THEN
         PUT D (B := TRUE);
      PUT D (N := Z);
      STORE D;
      RECREATE D;
      PUT D (N := Z+1);
      STORE D;
      END;
   CLOSE DBASE;
END.
```

REMOVE STATEMENT

The REMOVE statement is similar to the FIND statement, except that if a record is found, it is locked and then removed from the specified subset.

The REMOVE statement performs the following steps in order:

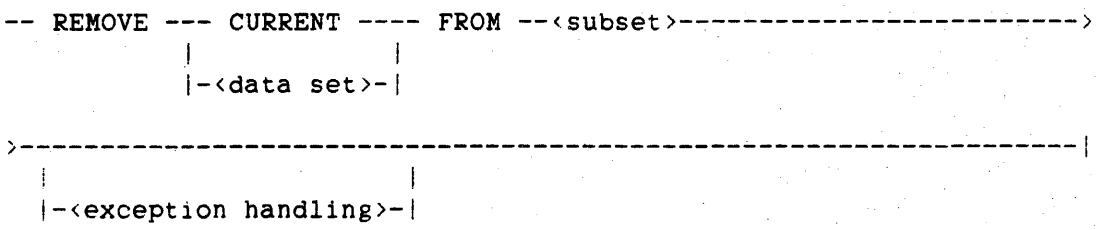
1. Frees the current record
2. Alters the current path to point to the record specified by CURRENT or the data set
3. Locks the previously found record and then removes the record from the specified subset

If an exception occurs after step 2, the current path is invalid.

If an exception occurs after step 3, the operation terminates, leaving the current path pointing to the record specified by CURRENT or by the data set.

Syntax

<remove statement>



See also
<data set>. 703
<exception handling>. 768
<subset>. 704

Semantics

The word "CURRENT" removes the current record from the specified subset. If this option is specified, the subset must have a valid current record; if it does not have a valid current record, an exception is returned.

The <data set> construct is used to find and remove from the specified subset the record referenced by the current path. An exception is returned if the record is not in the subset.

The <subset> construct specifies the subset from which a record is to be deleted. The subset must be a manual subset of the specified data set.

If the subset is embedded in a data set, the data set must have a current record defined and that record must be locked; if not, an exception is returned.

An exception is returned in the following cases:

1. If CURRENT is specified and the specified subset does not have a valid current record
2. If a data set is specified and the record is not in the subset
3. If the specified subset is embedded in a data set, and the data set does not have a current record defined and locked

Examples

When the REMOVE statement is executed, the current paths still refer to the deleted record. Therefore, a subsequent FIND statement on the current record results in a NOTFOUND exception. However, the "FIND NEXT" and "FIND PRIOR" forms of the FIND statement give valid results.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (3);
   B BOOLEAN;
   N NUMBER (3);
   R REAL;
   );
SS SUBSET OF D BIT VECTOR;
```

then the following BDMSALGOL program demonstrates the use of the REMOVE statement to lock and remove the record of data set D that is referenced by the current path from the subset SS.

```
BEGIN
   DATABASE DBASE;
   BOOLEAN RSLT;
   INTEGER MN;

   OPEN UPDATE DBASE;
   SET SS TO BEGINNING;
   FIND NEXT SS :RSLT;
   WHILE NOT RSLT DO
      BEGIN
      GET D (MN := N);
      IF MN < 10 THEN
         REMOVE D FROM SS;
      FIND NEXT SS :RSLT;
      END;
   CLOSE DBASE;
END.
```

BDMS SET STATEMENT

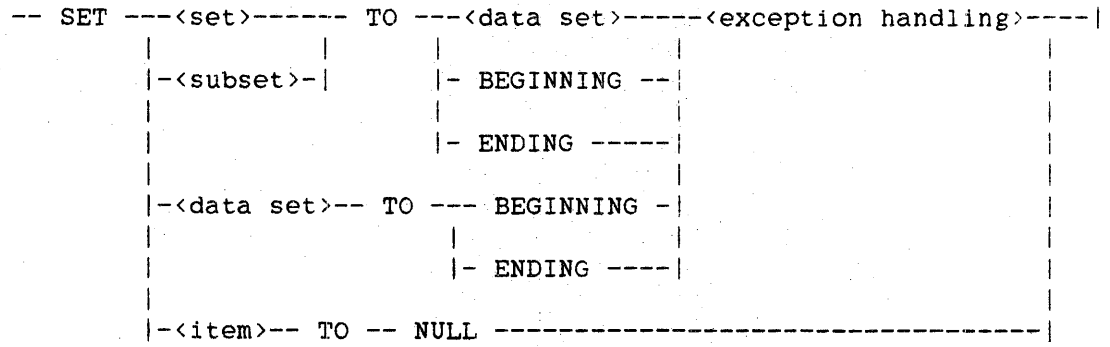
The BDMS "SET" statement alters the current path or changes the value of an item in the current record. Only the record area is affected. The data set is not affected until a subsequent STORE statement is executed.

The SET statement performs the following steps in order:

1. Frees the current path of the data set, set, or subset
2. Performs one of the following actions:
   a. Alters the current path of the data set, set, or subset to point to the beginning or the ending of the indicated structure
   b. Alters the set or subset path to point to the current path of another data set
   b. Assigns a NULL value to a particular item

Syntax

<BDMS set statement>



<item>

See also
<data set>. 703
<exception handling>. 768
<qualification> 693
<set> 703
<subset>. 704

Semantics

The constructs <data set>, <set>, or <subset> following the word "SET" specify the data set, set, or subset, respectively, whose path is altered.

If "TO <data set>" is specified, the current path of the set or subset is altered to point to the current record of the specified data set.

If "TO BEGINNING" is specified, the current path of the set, subset, or data set is altered to point to the beginning of the set, subset, or data set, respectively.

If "TO ENDING" is specified, the current path of the set, subset, or data set is altered to point to the ending of the set, subset, or data set, respectively.

The <item> construct specifies an item of the current record that is assigned a NULL value. The item cannot be a link item. NULL is the DASDL-declared NULL value if present; otherwise, it is the system default NULL value.

Pragmatics

After a "SET TO BEGINNING" form of the SET statement, the "FIND NEXT" and "FIND FIRST" forms of the FIND statement are equivalent; similarly, after a "SET TO ENDING", a "FIND PRIOR" and "FIND LAST" are equivalent.

Example

If the database DBASE is described in DASDL as follows:

```
D DATA SET (
   A ALPHA (20);
   B BOOLEAN;
   N NUMBER (2);
   R REAL;
   );
S SET OF D KEY (N):
SS SUBSET OF D WHERE (N = 3):
```

then the following BDMSALGOL program demonstrates different ways to use the SET statement.

```
BEGIN
   FILE CARD_FILE(KIND=READER),
        PRINT_FILE(KIND=PRINTER);
   DATABASE DBASE;
   BOOLEAN MB,RSLT;
   REAL MR;
   INTEGER MN;
   EBCDIC ARRAY MA[0:2];
   LABEL CLOSE_DATABASE;

   OPEN INQUIRY DBASE;
   SET SS TO BEGINNING :RSLT;
   IF RSLT THEN
      BEGIN
      WRITE(PRINT_FILE,<"** NO ENTRIES IN SS. **">);
      GO CLOSE_DATABASE;
      END;
   WHILE NOT READ(CARD_FILE,<I3>,MN) DO
      BEGIN
      FIND S AT N = MN;
      SET SS TO D :RSLT;
      IF RSLT THEN
         WRITE(PRINT_FILE,<I3," NOT IN SS.">,MN)
      ELSE
         BEGIN
         GET D(MA[0] := A,MB := B);
         IF MB THEN
            GET D (MR := R)
         ELSE
            MR := 0;
         WRITE(PRINT_FILE,<I3," ",A3," ",L5," ",E4.2>,
               MN,MA[0],MB,MR);
         END;
      END;

CLOSE_DATABASE:
   CLOSE DBASE;
END.
```

STORE STATEMENT

The STORE statement places a new or modified record into a data set or a global record area. The data from the user work area for the data set or global record is inserted into the data set or global record area.

The STORE statement performs the following actions depending on the prior operation.

1. After a CREATE or RECREATE statement:

    a. Checks the data in the user work area for validity if a VERIFY condition is specified in the DASDL

    b. Tests the record for validity for insertion in each set in the data set (for example, tests whether or not duplicates are allowed)

    c. Evaluates the WHERE condition for each automatic subset

    d. Inserts the record into all sets and automatic subsets if all conditions are satisfied

    e. Locks the new record

    f. Alters the data set path to point to the new record

2. After a BDMS "LOCK" or MODIFY statement:

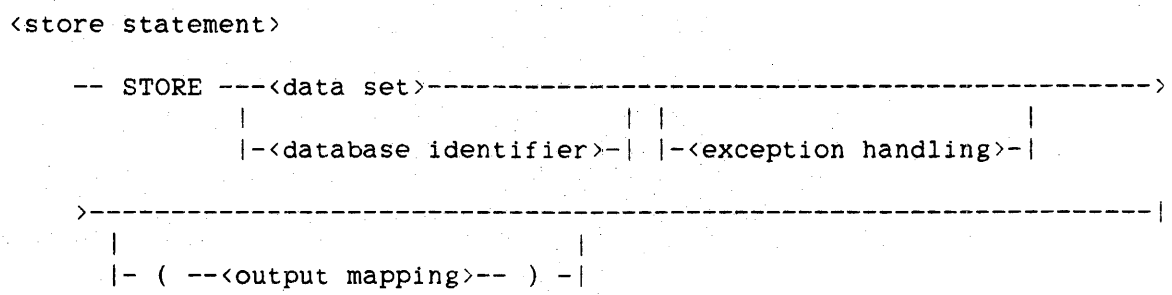
    a. Checks the data in the user work area for validity if a VERIFY condition is specified in the DASDL.

    b. Re-evaluates the conditions if items involved in the insertion conditions have changed. If the condition yields FALSE, the record is removed from each automatic subset that contains the record. If the condition yields TRUE, the record is inserted into each automatic subset that does not contain the record.

    c. Deletes and re-inserts the record in the proper position if a key used in the ordering of a set or automatic subset is modified so that the record must be moved within that set or automatic subset.

    d. Stores the record in a manual subset, but performs no reordering on that subset. The user is responsible for maintaining manual subsets. A subsequent reference to the record using that subset produces undefined results.

Syntax



See also

Semantics

If the <data set> form is used, the data in the user work area for the data set is returned to the specified data set.

If the <database identifier> form is used, the data in the user work area for the global data is returned to the global data record area. The global data record must be locked before a STORE statement references it; otherwise, the STORE statement is terminated with an exception.

An exception is returned and the record is not stored if the record does not meet any of the validity conditions.

An exception is returned if the data set path is valid and the current record is not locked, or if the global data record is not locked.

Example

If the database DBASE is described in DASDL as follows:

```
OPTIONS(AUDIT);
R RESTART DATA SET (
   P ALPHA (10);
   Q ALPHA (100);
   );
D DATA SET (
   A ALPHA (3);
   N NUMBER (3);
   );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the STORE statement can be used to place a record into the data set D.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   EBCDIC ARRAY MY_A[0:2];
   INTEGER MY_N;

   OPEN UPDATE DBASE;
   MY_N := 1;
   WHILE MY_N < 100 DO
      BEGIN
      CREATE D;
      PUT D (N := MY_N);
      BEGINTRANSACTION R;
      STORE D;
      ENDTRANSACTION R;
      MY_N := *+1;
      END;
   WHILE NOT READ(CARD_FILE,<I3,A3>,MY_N,MY_A[0]) DO
      BEGIN
      LOCK S AT N = MY_N;
      BEGINTRANSACTION R;
      PUT D (A := MY_A[0]);
      STORE D;
      ENDTRANSACTION R;
      END;
   CLOSE DBASE;
END.
```

9.4 BDMSALGOL FUNCTIONS

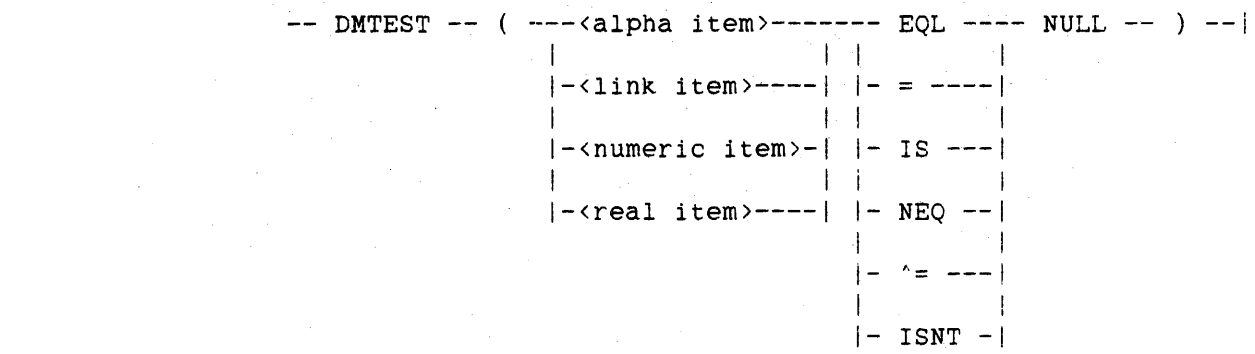
There are two data management functions available in the BDMSALGOL language: DMTEST and STRUCTURENUMBER. These functions are described in this section.

DMTEST FUNCTION

The DMTEST function determines whether an item is NULL. The function returns a Boolean value, which is TRUE if the value of the relationship expressed between the parentheses is TRUE and FALSE otherwise. No status value is associated with the DMTEST function.

Syntax

<dmtest function>



<alpha item>
<numeric item>
<real item>

See also
<link item> 705
<qualification> 693

Semantics

The <alpha item> construct specifies an alpha item declared in the DASDL. The alpha item contains a NULL value after a "SET <item> TO NULL" form of the BDMS "SET" statement, where <item> is the alpha item.

The <numeric item> construct specifies a numeric item declared in the DASDL. The numeric item contains a NULL value after a "SET <item> TO NULL" form of the BDMS "SET" statement, where <item> is the numeric item.

The <real item> construct specifies a real item declared in the DASDL. The real item contains a NULL value after a "SET <item> TO NULL" form of the BDMS "SET" statement, where <item> is the real item.

The <link item> construct specifies a link item declared in the DASDL. The link item contains a NULL value if either of the following is TRUE:

1. The link item does not point to a record.

2. No current record is present for the data set that contains the link item. This condition occurs following a BDMS "OPEN" statement, following the "SET TO BEGINNING" and "SET TO ENDING" forms of the BDMS "SET" statement, or when the record containing the link item has been deleted.

The link item contains a non-NULL value if the link item points to a record, even if that record has been deleted.

The word "NULL" represents the DASDL-defined NULL value.

Example

If the database DBASE is described in DASDL as follows:

```
 D DATA SET (
    A ALPHA (3);
    B BOOLEAN;
    N NUMBER (3);
    R REAL;
    );
S SET OF D KEY N;
```

then the following BDMSALGOL program demonstrates how the DMTEST function can be used to determine whether or not the alpha item A is NULL.

```
BEGIN
   FILE CARD_FILE(KIND=READER);
   DATABASE DBASE;
   INTEGER X;

   OPEN UPDATE DBASE;
   WHILE NOT READ(CARD_FILE,<I3>,X) DO
      BEGIN
      LOCK S AT N = X;
      IF DMTEST(A ISNT NULL) THEN
         DELETE D
      ELSE
         FREE D;
      END;
   CLOSE DBASE;
END.
```

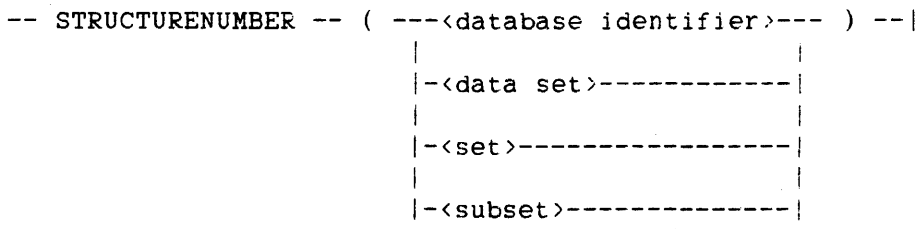
STRUCTURENUMBER FUNCTION

The STRUCTURENUMBER function allows the programmer to determine programmatically the structure number of a data set, set, subset, or of global data. This function can be used to analyze the result of exception conditions.

This capability is most useful when several sets span a data set and the previous operation against the data set yielded an exception. The program can determine which structure caused the exception from the corresponding structure number.

Syntax

<structurenumber function>



See also

Semantics

If the <database identifier> construct is used, the STRUCTURENUMBER function returns the structure number of the global data. Otherwise, the function returns the structure number of the data set, set, or subset specified, respectively, by the <data set>, <set>, or <subset> construct.

Pragmatics

When a partitioned structure is declared in DASDL, it is assigned one or more structure numbers, depending on <unsigned integer> in the "OPEN PARTITIONS = <unsigned integer>" form of the DASDL "OPEN" data set option. (For more information, refer to the "DMSII DASDL Reference Manual.") For example, if "OPEN PARTITIONS = 3" is specified, three structure numbers are assigned to the partitioned structure. The STRUCTURENUMBER function returns the smallest structure number assigned to the structure; however, DMSTRUCTURE, the value in the exception status word, can evaluate to any of these values; that is, it does not necessarily evaluate to the same structure number every time.

Example

```
REAL ERRORWORD;
IF STRUCTURENUMBER(D) = ERRORWORD.DMSTRUCTURE THEN
   REPLACE EA BY "D FAULT";
```

9.5 EXCEPTION PROCESSING

When executing BDMSALGOL statements, any one of several exception conditions, which prevent the operation from being performed as specified, can be encountered. These conditions result if the operation encounters a fault or does not produce the expected action. For example, execution of the statement

```
FIND S AT NAME = "JONES"
```

would result in an exception if there is no entry in S that has a value of "JONES" for the key item. If the operation terminates normally, no exception occurs.

A database status word is returned to the BDMSALGOL program at the conclusion of each BDMSALGOL statement. The value of this word indicates whether or not an exception has occurred and specifies the nature of the exception.

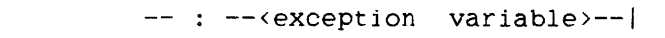
DATABASE STATUS WORD

In a BDMSALGOL statement, the user must specify the name of a real variable or Boolean variable in which the value of the database status word is stored at the completion of the BDMSALGOL statement. If no such variable is specified, the status value cannot be interrogated.

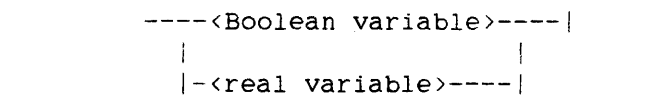
The <exception handling> construct is used in the syntax of the BDMSALGOL statements to denote those statements where a program variable can be designated to receive the value of the database status word.

Syntax

<exception handling>



<exception variable>



See also
<Boolean variable> 234
<real variable> 447

Semantics

A Boolean variable is a Boolean simple variable or an element of a Boolean array. A real variable is a real simple variable or an element of a real array.

Example

```
REAL ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
```

EXCEPTION HANDLING

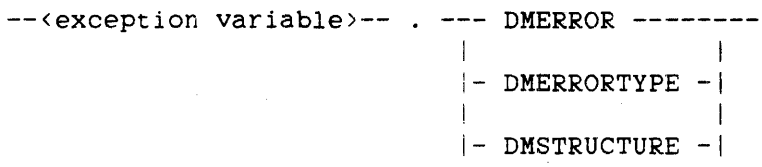
If the database status word is treated as a Boolean quantity, its value is TRUE if the operation containing it results in an exception; otherwise, it is FALSE.

If an exception results from a database operation, but the value of the database status word is not assigned to an exception variable in the program, the program is terminated. If the value is assigned to an exception variable, no other indication of the exception is given. The BDMSALGOL program is responsible for determining the nature of the exception and responding appropriately. Failure to do so can cause unpredictable results.

To determine the nature of an exception, the database status word is interrogated by specifying a period (.) and an attribute name following the exception variable. The attribute names are recognized by the BDMSALGOL compiler as representations of the appropriate fields of the database status word.

Syntax

<exception value>



Semantics

The attribute "DMERROR" yields a numeric value identifying a major category. Mnemonic names are also available to represent these numeric values. Either the category number or the category mnemonic can be used to test for a particular category.

The attribute "DMERRORTYPE" yields a numeric value identifying the subcategory of the major category.

The attribute "DMSTRUCTURE" yields a numeric value identifying the structure number of the structure involved in the exception. The structure numbers of all invoked structures are shown in the program listing if the program was compiled with the compiler control options LIST and LISTDB equal to TRUE.

Example

The following example illustrates one way of handling exceptions in a BDMSALGOL program:

```
REAL ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
IF BOOLEAN(ERRORWORD) THEN
   IF ERRORWORD.DMERROR = OPENERROR THEN
      IF ERRORWORD.DMERRORTYPE = 1 THEN
         DISPLAY("I/O ERROR ON ACCESSROUTINES CODE FILE");
```

If the exception variable is a Boolean variable, the preceding example is changed as follows:

```
BOOLEAN ERRORWORD;
OPEN UPDATE DBASE :ERRORWORD;
IF ERRORWORD THEN
   IF REAL(ERRORWORD).DMERROR = OPENERROR THEN
      IF REAL(ERRORWORD).DMERRORTYPE = 1 THEN
         DISPLAY("I/O ERROR ON ACCESSROUTINES CODE FILE");
```

9.6 BDMSALGOL COMPILER CONTROL OPTIONS

The following compiler control options are available in the BDMSALGOL language in addition to the options available in the ALGOL language. For information on the compiler control options available in ALGOL, refer to "Compiler Control Options" in the chapter "Compiling Programs."

See also
Compiler Control Options. 595

<datadictinfo option>

-- DATADICTINFO --|

(Type: Boolean, Default value: FALSE)

If the DATADICTINFO option is TRUE, information about the usage of database structures and items is placed into the object code file. This usage information shows which database structures and items were invoked by the program and whether they were read or written. This option cannot be assigned a value after the appearance of the first syntactical item in the program.

<listdb option>

-- LISTDB --|

(Type: Boolean, Default value: FALSE)

If both the LIST option and the LISTDB option are TRUE, the printer listing contains information about the invoked databases, structures, and items, including the declared database titles. If the LIST option is TRUE but the LISTDB option is FALSE, the printer listing does not contain this information. The value of LISTDB is ignored if the LIST option is FALSE.

<nodmdefines option>

```
-- NODMDEFINES --|
```

(Type: Boolean, Default value: FALSE)

If the NODMDEFINES option is TRUE, no defines are expanded in BDMSALGOL constructs.

When the NODMDEFINES option is FALSE, defines in BDMSALGOL constructs are expanded, including defines in the following situations:

1. A database item has the same identifier as a define.

2. An alphanumeric string that is part of a database item identifier (between two hyphens, before the first hyphen, or after the last hyphen) is the same as the identifier of a define.

<tracedb option>

```
-- TRACEDB --|
```

(Type: Boolean, Default value: FALSE)

If the TRACEDB option is TRUE, the printer listing contains a trace of the communication between the BDMSALGOL compiler and the DATABASE/INTERFACE program. The action of this option is independent of the value of the LIST option.

9.7 BINDING AND SEPCOMP OF DATABASES

BINDING

Programs that declare and reference databases can be bound together by the Binder program. The following example shows a BDMSALGOL host program that declares a database and an external procedure, and a separate procedure that is to be bound to the host and that declares the database in its global part.

The DASDL description of the database TESTDB is as follows:

```
DS DATA SET (
   NAME GROUP (
      LAST ALPHA (10);
      FIRST ALPHA (10);
      );
   AGE NUMBER (2);
   SEX ALPHA (1);
   SSNO ALPHA (9);
   );
NAMESET SET OF DS KEY (LAST, FIRST);
```

The following program, compiled with the name SEP/HOST, is the BDMSALGOL host program:

```
BEGIN
   DATABASE TESTDB;
   PROCEDURE P; EXTERNAL;
   OPEN UPDATE TESTDB;
   P;
   CLOSE TESTDB;
END.
```

The following separate procedure, P, compiled with the name SEP/P, is to be bound to the external procedure P of the host. Note how the database TESTDB is declared in the global part.

```
[DATABASE TESTDB;]
PROCEDURE P;
   BEGIN
   BOOLEAN EXCEPTIONWORD;
   EXCEPTIONWORD := FALSE;
   SET NAMESET TO BEGINNING;
   WHILE NOT EXCEPTIONWORD DO
      BEGIN
      FIND NEXT NAMESET AT LAST = "SMITH"
         AND FIRST = "JOHN" :EXCEPTIONWORD;
      % OTHER STATEMENTS
      END;
   END;
```

The separate procedure P in SEP/P can be bound to the host SEP/HOST using the following Work Flow Language (WFL) job. The resulting bound code file is named GLOBDB.

```
?BEGIN JOB BIND/GLOB;
 BIND GLOBDB WITH BINDER LIBRARY;
 BINDER DATA
 HOST IS SEP/HOST;
 BIND P FROM SEP/P;
?END JOB.
```

SEPCOMP

Programs that declare and use databases can also make use of the sepcomp facility of the compiler, as shown in the following example.

The DASDL description of the database TESTDB is as follows:

```
DS DATA SET (
   NAME GROUP (
      LAST ALPHA (10);
      FIRST ALPHA (10):
      ):
   AGE NUMBER (2):
   SEX ALPHA (1):
   SSNO ALPHA (9);
   );
NAMESET SET OF DS KEY (LAST, FIRST):
```

Because the MAKEHOST compiler control option is TRUE, the following program, compiled as MY/HOST, can be used as a host program for SEPCOMP.

```
$ SET MAKEHOST
BEGIN                                                    1
   DATABASE TESTDB;                                      2
   PROCEDURE P;                                          3
      BEGIN                                              4
      BOOLEAN EXCEPTIONWORD;                             5
      EXCEPTIONWORD := FALSE;                            6
      SET NAMESET TO ENDING;                             7
      WHILE NOT EXCEPTIONWORD DO                         8
         BEGIN                                           9
         FIND NEXT NAMESET AT LAST = "SMITH"            10
            AND FIRST = "JOHN": EXCEPTIONWORD:          11
         % OTHER STATEMENTS                             12
         END;                                           13
      END;                                              14
   OPEN UPDATE TESTDB;                                  15
   P;                                                   16
   CLOSE TESTDB;                                        17
END.                                                    18
```

The following source input invokes the SEPCOMP facility to change the record of the host MY/HOST with sequence number 7, recompile the procedure P, and bind the new P to the host.

```
$ SET SEPCOMP "MY/HOST"          % PATCH FOLLOWS
      SET NAMESET TO BEGINNING:
```

10 COMPILE-TIME FACILITY

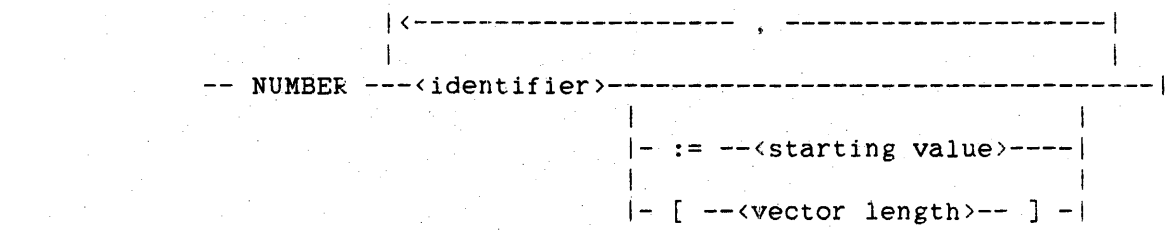
The compile-time facility is used to conditionally and iteratively compile ALGOL source data. This chapter describes the declaration and use of compile-time variables, compile-time identifiers, compile-time statements, an extension to the DEFINE declaration, and compiler control options that pertain to the compile-time facility.

The compile-time facility is available only in ALGOL compilers that have been compiled with the CTPROC compiler-generation option set to TRUE.

COMPILE-TIME VARIABLE

Syntax

<compile-time variable declaration>



<number identifier>

An <identifier> that is associated with a compile-time variable in a compile-time variable declaration.

<starting value>

--<compile-time arithmetic expression>--|

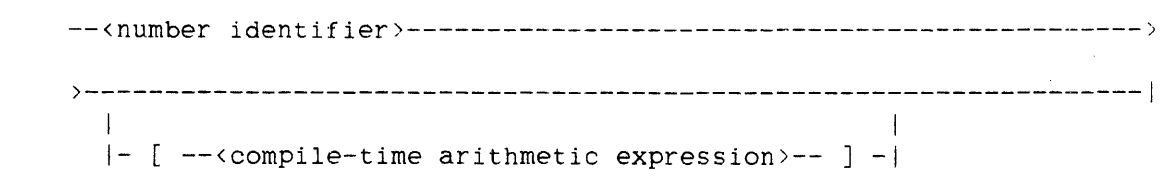
<compile-time arithmetic expression>

Any <arithmetic expression> that can be fully evaluated at compile time. A compile-time arithmetic expression consists of constants and <compile-time variable>s.

<vector length>

--<compile-time arithmetic expression>--|

<compile-time variable>



Semantics

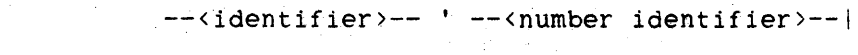
An identifier declared in a compile-time variable declaration is a "number variable" or an arithmetic compile-time variable. A compile-time variable represents a single-precision arithmetic value. It can be used wherever an arithmetic value is allowed and represents the value most recently assigned to it. The value of a compile-time variable can be changed at any time during compilation by using the compile-time 'LET statement. A compile-time variable can be declared with a starting value; if a starting value is not explicitly declared, the starting value is zero.

If a compile-time variable is declared with a vector length, a vector or "array" of compile-time variables is created. When an identifier declared in this way is used, it must be subscripted by a compile-time arithmetic expression with a value in the range 0 through (vector length - 1). The value of the vector length must be in the range 1 to 1023.

COMPILE-TIME IDENTIFIER

Syntax

<compile-time identifier>

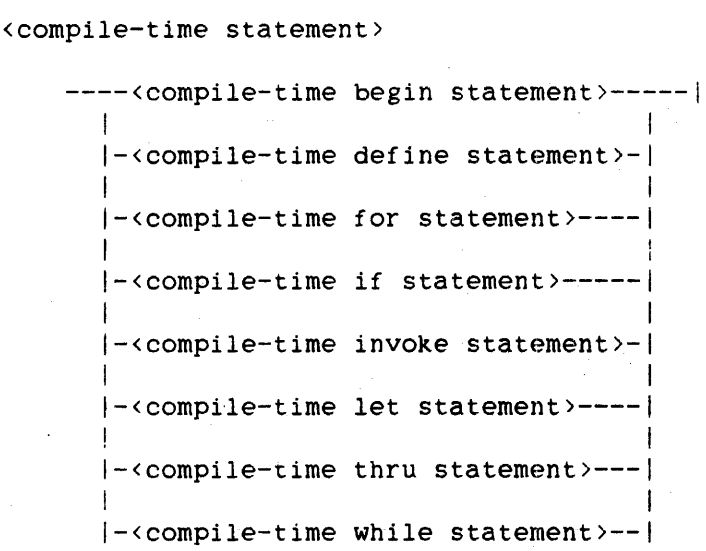


Semantics

A compile-time identifier can appear anywhere an identifier can be used, including declarations. No blank characters can appear between <identifier> and the apostrophe ('). The created identifier is the <identifier>, followed by an apostrophe, followed by numeric characters corresponding to the value of the number identifier, with leading zeros suppressed.

COMPILE-TIME STATEMENTS

Syntax



Semantics

Compile-time statements begin with an apostrophe ('), which distinguishes them from other ALGOL constructs. They are recognized at a very primitive level in the compiler and can, therefore, appear almost anywhere (for example, between any two ALGOL language components).

The compile-time statements are intended to provide a method for altering the normal flow of compilation, primarily by conditional and iterative compilation.

Compile-time statements are terminated in the same manner as ALGOL statements.

Note that through the use of compile-time variables, a compile-time arithmetic or Boolean expression can be fully evaluated at compile time and yet not have the same value each time it is evaluated.

'BEGIN Statement

Syntax

<compile-time begin statement>

```
-- 'BEGIN --<compile-time text>-- 'END --<end remark>--|
```

<compile-time text>

Any ALGOL source text, including complete compile-time statements.

Semantics

A 'BEGIN statement delimits a portion of ALGOL source text. It is normally used in conjunction with 'FOR statements, 'IF statements, and 'THRU statements. If the 'BEGIN statement is executed, the compiler processes all the delimited text; otherwise, the compiler skips (ignores) the text. Anything following the 'END up to the first special character, END, ELSE, 'END, 'ELSE, or UNTIL is considered to be an <end remark> and is ignored.

'DEFINE Statement

Syntax

<compile-time define statement>

```
-- 'DEFINE --<identifier>-- = --<compile-time statement>--|
```

<compile-time define identifier>

An <identifier> that is associated with a compile-time statement in a compile-time define statement.

Semantics

The 'DEFINE statement declares an identifier to represent a compile-time statement.

The compile-time statement is processed when it is referenced by the identifier in a subsequent 'INVOKE statement.

'FOR Statement

Syntax

<compile-time for statement>

```
-- 'FOR --<number identifier>-- := ------------------------------->

>-<compile-time arithmetic expression>-- STEP --------------------->

>-<compile-time arithmetic expression>-- UNTIL -------------------->

>-<compile-time arithmetic expression>-- DO ----------------------->

>-<compile-time statement>----------------------------------------|
```

Semantics

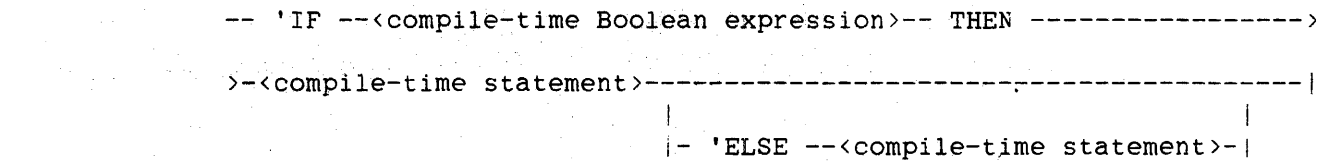
The 'FOR statement provides iterative compilation of ALGOL source input. The value of the compile-time arithmetic expression following "STEP" can be positive or negative but must not be equal to zero.

The action of this statement is similar to that of the non-compile-time FOR statement. One exception is that the compile-time arithmetic expressions following "STEP" and "UNTIL" are evaluated only once, at the beginning of the 'FOR statement, and are not re-evaluated, even though their compile-time components can change value.

'IF Statement

Syntax

<compile-time if statement>



<compile-time Boolean expression>

Any <Boolean expression> that can be fully evaluated at compile time. A compile-time Boolean expression consists of constants and <compile-time variable>s.

Semantics

The 'IF statement provides conditional compilation of ALGOL source input.

If the value of the compile-time Boolean expression is TRUE, then the compile-time statement following "THEN" is processed: if it is FALSE, the compile-time statement following "'ELSE" is processed, if present. In either case, compilation continues with the statement following the 'IF statement.

'INVOKE Statement

Syntax

<compile-time invoke statement>

```
-- 'INVOKE --<compile-time define identifier>--|
```

Semantics

The 'INVOKE statement causes the compile-time statement previously associated with the compile-time define identifier in a 'DEFINE statement to be processed.

'LET Statement

Syntax

<compile-time let statement>

```
-- 'LET --<compile-time variable>-- := ---------------------------->

>-<compile-time arithmetic expression>-----------------------------|
```

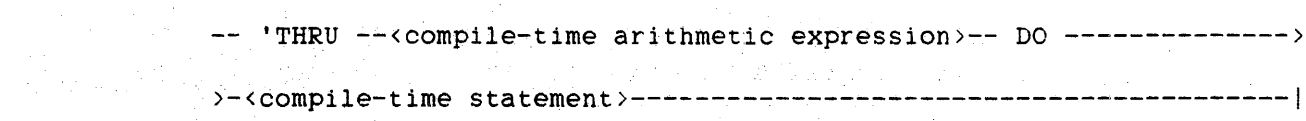
Semantics

The 'LET statement is used to modify the value of a compile-time variable. If the compile-time variable was declared using the <vector length> construct, it must be subscripted by a compile-time arithmetic expression with a value in the range 0 to (vector length - 1).

'THRU Statement

Syntax

<compile-time thru statement>



Semantics

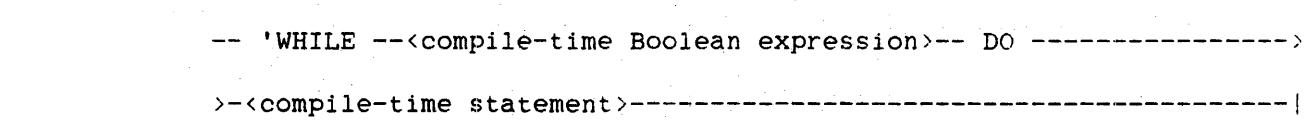
The 'THRU statement provides iterative compilation of ALGOL source input. The compile-time arithmetic expression must have a value greater than or equal to zero.

The compile-time statement following "DO" is processed <compile-time arithmetic expression> times. If this value is zero, the 'THRU statement is skipped.

'WHILE Statement

Syntax

<compile-time while statement>



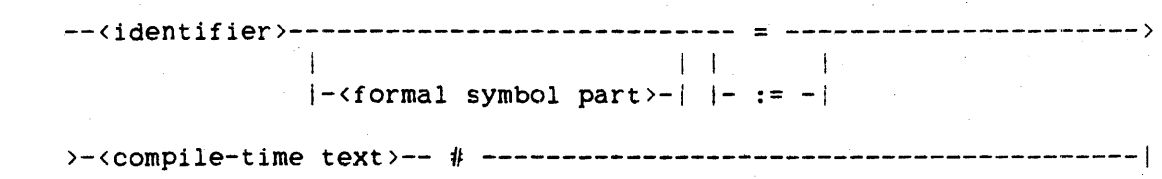
Semantics

The 'WHILE statement provides iterative compilation of ALGOL source input. The compile-time Boolean expression is evaluated at the beginning of the statement. If it is TRUE, the compile-time statement is processed, the compile-time Boolean expression is evaluated again, and this sequence is repeated. Whenever the compile-time Boolean expression is FALSE, the 'WHILE statement is finished, and compilation continues with the following statement.

EXTENSION TO THE DEFINE DECLARATION

The following extension to the DEFINE declaration is available when using the compile-time facility.

Syntax



See also
<formal symbol part>. 60

Semantics

If a define identifier is declared using the assignment operator (:=), then any compile-time statements in the compile-time text are evaluated once at the time of, and in the scope of, the DEFINE declaration. Otherwise, the compile-time items in the compile-time text are evaluated on each invocation of the define identifier.

COMPILE-TIME COMPILER CONTROL OPTIONS

The following compiler control options are available only in the compile-time facility. For more information on compiler control options and the printer listing file of the compiler, refer to the "Compiling Programs" chapter.

<ctlist option>

```
-- CTLIST --|
```

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CTLIST option are TRUE, all input records processed are written to the printer listing. In particular, during iterative compile-time statements, input records that are compiled repeatedly are listed repeatedly. These input records are identified by an asterisk (*) just to the left of the sequence number. If the LIST option is TRUE but the CTLIST option is FALSE, input records are written to the printer listing only the first time they are compiled. The value of the CTLIST option is ignored if the LIST option is FALSE.

<ctmon option>

```
-- CTMON --|
```

(Type: Boolean, Default value: FALSE)

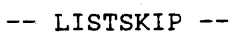
If both the LIST option and the CTMON option are TRUE, all assignments to compile-time variables are monitored and written to the printer listing. The current value of a compile-time variable when it is referenced and the new value when it is changed are listed. If the LIST option is TRUE and the CTMON option is FALSE, this monitor information does not appear in the printer listing. The value of the CTMON option is ignored if the LIST option is FALSE.

<cttrace option>

(Type: Boolean, Default value: FALSE)

If both the LIST option and the CTTRACE option are TRUE, values of certain expressions that are components of compile-time statements are written to the printer listing. If the LIST option is TRUE and the CTTRACE option is FALSE, this information does not appear in the printer listing. The value of the CTTRACE option is ignored if the LIST option is FALSE.

<listskip option>



(Type: Boolean, Default value: TRUE)

If both the LIST option and the LISTSKIP option are TRUE, all records are written to the printer listing whether or not they are skipped. Skipped records are denoted in the listing by the word "SKIP" to the right of the sequence number. If the LIST option is TRUE and the LISTSKIP option is FALSE, source records that are skipped by the compile-time facility are not written to the printer listing. The value of the LISTSKIP option is ignored if the LIST option is FALSE.

11 BATCH FACILITY

The batch facility allows more efficient use of system resources by enabling a group of programs possessing certain common characteristics to share many system overhead functions normally associated with individual jobs. Overhead functions that can be shared include job initialization, job termination, linking of intrinsics, memory allocation, and so on.

Programs submitted as part of a batch are compiled and executed according to the following restrictions:

1. The programs are compiled for syntax errors only or for one-time execution.

2. If a program declares more than one printer file, all output is directed to the same physical file. Likewise, if a program declares more than one card file, all input is taken from the same physical file.

3. If a program explicitly opens or closes the printer file, that program is discontinued, and the next program in the batch is initiated.

4. If a program requires operator intervention, it is discontinued, and the next program in the batch is initiated.

5. If a program attempts to initiate a task, it is discontinued, and the next program in the batch is initiated.

6. WAIT statements are allowed but have no effect.

7. The source records of all programs in the batch must use the same character type.

8. In FILE declarations, the values assigned to file attributes must not be variables or expressions involving variables.

9. The following compiler control options are invalid or ignored.

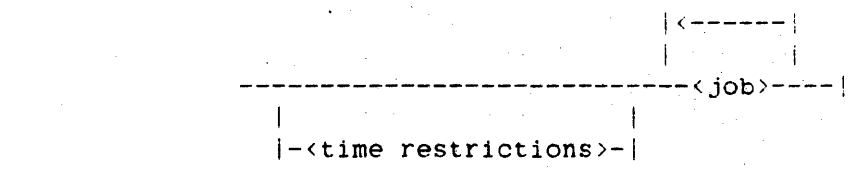
AUTOBIND	INTRINSICS	PURGE
BIND	LEVEL	SEGDESCABOVE
BINDER	LIBRARY	SEPCOMP
DUMPINFO	LINEINFO	SHARING
ERRLIST	LISTINCL	STOP
EXTERNAL	LOADINFO	TARGET
GO TO	MAKEHOST	TIME
HOST	MCP	USE
INCLNEW	MERGE	VERSION
INCLSEQ	NEW	XDECS
INCLUDE	NEWSEQERR	XREF
INITIALIZE	NOBINDINFO	XREFFILES
INSTALLATION	NOXREFLIST	XREFS

BATCH SOURCE INPUT

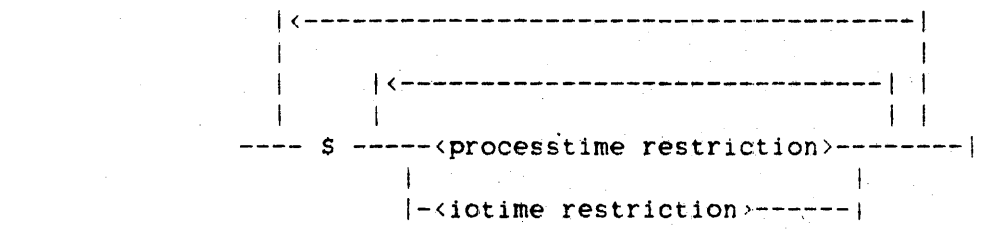
The ALGOL compiler performs batch compilation only if the compiler is initiated by the Work Flow Language (WFL), and the source input is in the form of <batch source input>. If a <job specifier> occurs in a CANDE-initiated compilation, the compilation is discontinued.

All batch source input must have the following form:

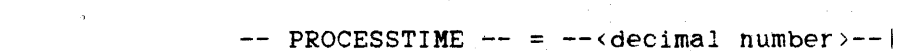
Syntax



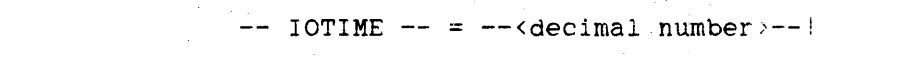
<time restrictions>



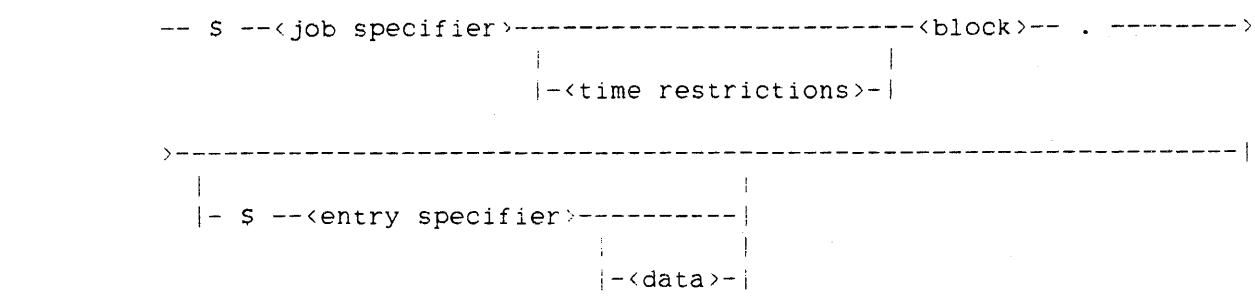
<processtime restriction>



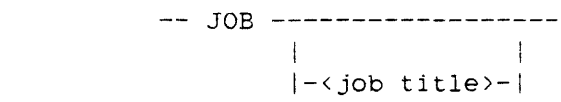
<iotime restriction>



<job>



<job specifier>



<job title>

Any valid file title.

<entry specifier>

<data>

Input data to be read by the program when it is executed.

Semantics

Batch source input consists of one or more individual programs or "jobs." Jobs are delimited in batch source input by "$ <job specifier>" records. Each job can be either compiled for syntax only or compiled and then executed. A job is executed if it contains a "$ ENTRY" record, and if the job needs input during its execution, this data follows the "$ ENTRY" record. Maximum processor times and I/O times can be specified for each job or for all jobs in a batch. Printed output for all jobs in a batch is written to the same file and is printed in one listing.

The <processtime restriction>, <iotime restriction>, <job specifier>, and <entry specifier> constructs are valid only for batch compilation and are not compiler control options.

The individual syntactic constructs of <batch source input> are described in more detail in the following paragraphs.

<time restrictions>

The <processtime restriction> and <iotime restriction> constructs specify the maximum processor time and maximum I/O time, respectively, allowed for the execution of a job before it is discontinued.

If the time restrictions appear before the first job in a batch, they apply to the execution of every job in the batch. If the time restrictions appear within a job, they apply to the execution of that job only, but they are meaningful only if they specify less time than the time restrictions, if any, appearing before the first job in the batch.

<job>

Each job in the batch source data consists of a complete program that is compiled for syntax only or for one-time execution. A job can include its own time restrictions and compiler control records.

<job specifier>

The first record in each job must be the "$ <job specifier>" record. If a job title is specified in the <job specifier> construct, it is used to identify the printed output produced by that job.

<block>

The <block> and any compiler control records that follow the "$ <job specifier>" record and precede the next "$ ENTRY" record, if any, or "$ <job specifier>" record make up the ALGOL program.

<entry specifier>

The "$ <entry specifier>" record specifies that the program is to be executed if no syntax errors are found and indicates the beginning of any input data to be used during execution. If no "$ <entry specifier>" record appears in the job, then the program is compiled for syntax errors only; it is not executed.

<data>

If a job needs input data during execution, the input data appears after the "$ ENTRY" record and before the next job, if any.

Example

```
?BEGIN JOB EXAMPLEBATCH;
?COMPILE BATCHA ALGOL;
?ALGOL DATA
$ PROCESSTIME = 5.5 IOTIME = 1
$ JOB A1
$ SET LIST SEQ
BEGIN
   FILE RDR(KIND = READER);
   INTEGER I;
   ARRAY Z[0:2];
   READ(RDR,/,FOR I := 0 STEP 1 UNTIL 2 DO Z[I]);
END.
$ ENTRY
2.315,3.71828,.5772,
$ JOB A2
BEGIN
   INTEGER A,B,C;
   A := B ++ C; % A SYNTAX ERROR
END.
$ ENTRY
$ JOB A3
BEGIN
   REAL PI;
   PI := 22/7;
END.
$ JOB A4
$ PROCESSTIME = 2.5 IOTIME = 0.5
BEGIN
   LABEL AGAIN;
   REAL X;
AGAIN:
   X := 355/133;
   GO TO AGAIN;
END.
$ ENTRY
?END JOB
```

This example shows a WFL job consisting of one batch compilation. The batch contains four programs or jobs. No job in this batch is allowed more than 5.5 seconds of processor time or 1 second of I/O time.

Job A1 compiles correctly and is executed using the one record of data immediately following its "$ ENTRY" record. Job A2 is not executed because it contains a syntax error but would be executed if it were free of errors. Job A3, though syntactically correct, is not executed because it does not contain a "$ ENTRY" record.

Job A4 is allowed only 2.5 seconds of processor time and 0.5 seconds of I/O time, compared to 5.5 seconds and 1 second, respectively, for the other three jobs in the batch. Job A4 compiles correctly and is executed; however, because it contains an infinite loop, it is discontinued after its allotted 2.5 seconds of processor time.

IMPLEMENTATION SCHEME

The goal of the batch facility is to eliminate as much system overhead as possible by reducing the number of tasks initiated within the running environment of A Series and B 5000/B 6000/B 7000 Series systems.

An entire batch is presented to the compiler as one file, thereby avoiding repeated initiation of the compiler. The compilation process for each individual job within a batch is virtually the same as that for nonbatched programs and yields equally efficient object code. When the compiler finishes compiling the batch, it generates special object code that links each job to the next one. When a job is not to be executed (the "$ ENTRY" record is not present or syntax errors occurred during compilation), it is not linked to the other jobs. When compilation is complete, the object code for all jobs in the batch resides in one disk file.

The printed output from the compiler is directed to a backup printer file with an altered BDNAME task attribute so that the output is not automatically printed. Logging information regarding the compilation is also saved in this file. The execution of the object code proceeds as follows:

1. The D2 stack is built.

2. The batchmonitor intrinsic is called and passed a procedure that serially calls each individual job.

3. The batchmonitor intrinsic processes the procedure passed to it. If any job causes a fatal run-time error, the batchmonitor intrinsic executes a procedure that goes automatically to the next individual job.

4. The batchmonitor intrinsic rewinds the two backup printer files, extracts the logging information, collates the output into a new printer file, and returns control to the Master Control Program (MCP).

All jobs share the same printer file and intrinsics and can even share the same object code segments. The individual jobs run serially and share the same stack space.

A job is protected from previous jobs by the batchmonitor intrinsic. If one job is discontinued with an error, execution is re-initiated by the batchmonitor intrinsic at the next individual job. If any job uses excessive I/O or processor time, the batchmonitor intrinsic terminates that job. Likewise, the batchmonitor intrinsic enforces the rule that no operator messages requiring responses are allowed by terminating any job that causes such a message.

The batchmonitor intrinsic extracts the logging information from the two printer files and summarizes it at the end of the output for each job. This summary procedure is easily modified to provide an interface with the accounting system of a given installation. Two words in each log record furnished by the compiler and the individual job are spares, thus facilitating any installation extensions.

A RESERVED WORDS

A ‹reserved word› in Extended ALGOL has the same syntax as an identifier. The reserved words are divided into three types.

A reserved word of type 1 can never be declared as an identifier; that is, it has a predefined meaning that cannot be changed. For example, because LIST is a type 1 reserved word, the declaration

```
ARRAY LIST[0:999]
```

is flagged with a syntax error.

A reserved word of type 2 can be redeclared as an identifier; it then loses its predefined meaning in the scope of that declaration. For example, because IN is a type 2 reserved word, the declaration

```
FILE IN(KIND = READER)
```

is legal, but in the scope of the declaration, the statement

```
SCAN P WHILE IN ALPHA
```

is flagged with a syntax error on the word "IN".

If a type 2 reserved word is used as a variable in a program but it is not declared as a variable, then the error message that results is not the expected "UNDECLARED IDENTIFIER." Instead, it may be "NO STATEMENT CAN START WITH THIS."

A reserved word of type 3 is context-sensitive. It can be redeclared as an identifier, and if it is used where the syntax calls for that reserved word, it carries the predefined meaning; otherwise, it carries the user-declared meaning. The different meanings for the type 3 reserved word STATUS are illustrated in the following example.

```
BEGIN
   TASK T;
   REAL STATUS;
   % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
   STATUS := 4.5;
   % IN THE NEXT STATEMENT, "STATUS" IS A TASK ATTRIBUTE
   IF T.STATUS = VALUE(TERMINATED) THEN
      % IN THE NEXT STATEMENT, "STATUS" IS A REAL VARIABLE
      STATUS := 10.0;
END.
```

Type 3 reserved words include the following:

- File attribute names
- Task attribute names
- Library attribute names
- Direct array attribute names
- Mnemonics for attribute values

All file attributes, direct array attributes, and mnemonics described in the "I/O Subsystem Reference Manual" are type 3 reserved words in ALGOL. All task attributes and mnemonics described in the "Work Flow Language (WFL) Reference Manual" are type 3 reserved words in ALGOL.

RESERVED WORDS LIST

The following is an alphabetical list of reserved words for Extended ALGOL. The number in parentheses following each word indicates the type of the reserved word. For example, "FOR (1)" indicates that FOR is a type 1 reserved word.

ABS (2)
ACCEPT (2)
ACTUALNAME (3)
ALL (3)
ALPHA (1)
ALPHA6 (3)
ALPHA7 (3)
ALPHA8 (3)
AND (2)
ANYFAULT (3)
ARCCOS (2)
ARCSIN (2)
ARCTAN (2)
ARCTAN2 (2)
ARRAY (1)
ARRAYS (3)
ARRAYSEARCH (2)
AS (3)
ASCII (2)
ASCIITOBCL (3)
ASCIITOEBCDIC (3)
ASCIITOHEX (3)
ATANH (2)
ATTACH (2)
AVAILABLE (2)
BACKUPPREFIX (3)
BASE (3)
BCL (2)
BCLTOASCII (3)
BCLTOEBCDIC (3)
BCLTOHEX (3)
BEGIN (1)
BOOLEAN (1)
BREAKPOINT (2)
BY (2)
BYFUNCTION (3)
BYTITLE (3)
CABS (2)
CALL (2)
CALLING (2)
CANCEL (2)
CASE (2)
CAT (2)
CAUSE (2)
CAUSEANDRESET (2)
CCOS (2)
CEXP (2)
CHANGEFILE (2)
CHARGECODE (3)
CHECKPOINT (2)
CHECKSUM (2)
CLASS (3)
CLN (2)
CLOSE (2)
CODE (3)
COMMENT (1)
COMPILETIME (2)
COMPILETYPE (3)
COMPLEX (2)
CONJUGATE (2)
CONTINUE (1)
COREESTIMATE (3)
COS (2)
COSH (2)
COTAN (2)
CRUNCH (3)
CSIN (2)
CSQRT (2)
DABS (2)
DAND (2)
DARCCOS (2)
DARCSIN (2)
DARCTAN (2)
DARCTAN2 (2)
DBS (3)

Reserved Words, Continued

DCOS (2)
DCOSH (2)
DEALLOCATE (2)
DECIMAL (2)
DECIMALPOINTISCOMMA (3)
DECLAREDPRIORITY (3)
DEFINE (2)
DELINKLIBRARY (2)
DELTA (2)
DEQV (2)
DERF (2)
DERFC (2)
DETACH (2)
DEXP (2)
DGAMMA (2)
DICTIONARY (2)
DIGITS (2)
DIMP (2)
DINTEGER (2)
DIRECT (1)
DISABLE (2)
DISCARD (3)
DISK (3)
DISKPACK (3)
DISPLAY (2)
DIV (2)
DLGAMMA (2)
DLN (2)
DLOG (2)
DMAX (2)
DMIN (2)
DNABS (2)
DNOT (2)
DO (1)
DONTWAIT (2)
DOR (2)
DOUBLE (1)
DROP (2)
DSCALELEFT (2)
DSCALERIGHT (2)
DSCALERIGHTT (2)
DSIN (2)
DSINH (2)
DSQRT (2)
DTAN (2)
DTANH (2)
DUMP (2)
EBCDIC (2)
EBCDICTOASCII (3)
EBCDICTOBCL (3)
EBCDICTOHEX (3)
ELAPSEDTIME (3)
ELSE (1)
EMPTY (2)
EMPTY4 (2)
EMPTY7 (2)
EMPTY8 (2)
ENABLE (2)
END (1)
ENTIER (2)
EQL (2)
EQV (2)
ERF (2)
ERFC (2)
EVENT (1)
EXCEPTIONEVENT (3)
EXCEPTIONTASK (3)
EXCHANGE (2)
EXP (2)
EXPONENTOVERFLOW (3)
EXPONENTUNDERFLOW (3)
EXPORT (2)
EXTERNAL (2)
FALSE (1)
FAMILY (3)
FILE (1)
FILECARDS (3)
FILES (3)
FILL (2)
FIRST (2)
FIRSTONE (2)
FIRSTWORD (2)
FIX (2)
FOR (1)
FORMAL (2)
FORMAT (1)
FORWARD (2)
FREE (2)
FREEZE (2)
FUNCTIONNAME (3)
GAMMA (2)
GEQ (2)
GO (1)
GTR (2)
HAPPENED (2)
HEAD (2)
HEX (2)
HEXTOASCII (3)
HEXTOBCL (3)
HEXTOEBCDIC (3)
HISTORY (3)
IF (1)
IMAG (2)
IMP (2)
IN (2)
INITIATOR (3)
INTEGER (1)
INTEGEROVERFLOW (3)
INTEGERT (2)
INTERRUPT (2)
INTNAME (3)
INVALIDADDRESS (3)
INVALIDINDEX (3)
INVALIDOP (3)
INVALIDPROGRAMWORD (3)
IS (2)
ISNT (2)
JOBNUMBER (3)
LABEL (1)
LB (2)
LENGTH (2)
LEQ (2)
LIBACCESS (3)
LIBERATE (2)
LIBPARAMETER (3)
LIBRARIES (3)
LIBRARY (2)
LINE (2)
LINENUMBER (2)
LINKLIBRARY (2)
LIST (1)
LISTLOOKUP (2)
LN (2)
LNGAMMA (2)
LOCK (2)
LOCKED (3)
LOG (2)
LONG (1)
LOOP (3)
LSS (2)
MASKSEARCH (2)
MAX (2)
MAXCARDS (3)
MAXIOTIME (3)
MAXLINES (3)
MAXPROCTIME (3)
MEMORYPARITY (3)
MEMORYPROTECT (3)
MERGE (2)
MESSAGESEARCHER (2)
MIN (2)
MOD (2)
MONITOR (2)

Reserved Words, Continued

MUX (2)
MYJOB (2)
MYSELF (2)
NABS (2)
NAME (3)
NEQ (2)
NO (2)
NORMALIZE (2)
NOT (2)
NUMERIC (2)
OF (2)
OFFER (3)
OFFSET (2)
ON (2)
ONES (2)
OPEN (2)
OPTION (3)
OR (2)
ORGUNIT (3)
OUT (3)
OUTPUTMESSAGE (2)
OWN (1)
PACK (3)
PAGED (3)
PARTNER (3)
PERMANENT (3)
PICTURE (2)
POINTER (1)
POTC (2)
POTH (2)
POTL (2)
PRIVATELIBRARIES (3)
PROCEDURE (1)
PROCESS (2)
PROCESSID (2)
PROCESSIOTIME (3)
PROCESSTIME (3)
PROCURE (2)
PROGRAMDUMP (2)
PROGkAMMEDOPERATOR (3)
PURGE (2)
RANDOM (2)
RB (2)
READ (2)
READLOCK (2)
REAL (1)
REEL (3)
REFERENCE (1)
REMAININGCHARS (2)
REMOVEFILE (2)
REPEAT (2)
REPLACE (2)
RESET (2)
RESIZE (2)
RESTART (3)
RETAIN (3)
REWIND (2)
RUN (2)
SCALELEFT (2)
SCALERIGHT (2)
SCALERIGHTF (2)
SCALERIGHTT (2)
SCAN (2)
SCANPARITY (3)
SDIGITS (2)
SECONDWORD (2)
SEEK (2)
SET (2)
SETACTUALNAME (2)
SIBS (3)
SIGN (2)
SIN (2)
SINGLE (2)
SINH (2)
SIZE (2)
SKIP (2)
SORT (2)
SPACE (2)
SQRT (2)
STACKER (2)
STACKNO (3)
STACKSIZE (3)
STARTTIME (3)
STATION (2)
STATUS (3)
STEP (1)
STOP (2)
STOPPOINT (3)
STRING (2)
STRINGPROTECT (3)
STRING4 (2)
STRING7 (2)
STRING8 (2)
SUBFILE (2)
SUBSPACES (3)
SWITCH (1)
TADS (3)
TAIL (2)
TAKE (2)
TAN (2)
TANH (2)
TARGETTIME (3)
TASK (1)
TASKATTERR (3)
TASKFILE (3)
TASKVALUE (3)
TEMPORARY (3)
THEN (1)
THRU (2)
TIME (2)
TIMELIMIT (2)
TIMES (2)
TITLE (3)
TO (2)
TRANSLATE (2)
TRANSLATETABLE (1)
TRUE (1)
TRUTHSET (1)
TYPE (3)
UNTIL (1)
USERCODE (3)
VALUE (1)
WAIT (2)
WAITANDRESET (2)
WHEN (2)
WHILE (1)
WITH (2)
WORDS (2)
WRITE (2)
ZERODIVIDE (3)
ZIP (1)

RESERVED WORDS BY TYPE

Type 1 Reserved Words

ALPHA
ARRAY
BEGIN
BOOLEAN
COMMENT
CONTINUE
DIRECT
DO
DOUBLE
ELSE
END
EVENT
FALSE
FILE
FOR
FORMAT
GO
IF
INTEGER
LABEL
LIST
LONG
OWN
POINTER
PROCEDURE
REAL
REFERENCE
STEP
SWITCH
TASK
THEN
TRANSLATETABLE
TRUE
TRUTHSET
UNTIL
VALUE
WHILE
ZIP

Type 2 Reserved Words

ABS
ACCEPT
AND
ARCCOS
ARCSIN
ARCTAN
ARCTAN2
ARRAYSEARCH
ASCII
ATANH
ATTACH
AVAILABLE
BCL
BREAKPOINT
BY
CABS
CALL
CALLING
CANCEL
CASE
CAT
CAUSE
CAUSEANDRESET
CCOS
CEXP
CHANGEFILE
CHECKPOINT
CHECKSUM
CLN
CLOSE
COMPILETIME
COMPLEX
CONJUGATE
COS
COSH
COTAN
CSIN
CSQRT
DABS
DAND
DARCCOS
DARCSIN
DARCTAN
DARCTAN2
DCOS
DCOSH
DEALLOCATE
DECIMAL
DEFINE
DELINKLIBRARY
DELTA
DEQV
DERF
DERFC
DETACH
DEXP
DGAMMA
DICTIONARY
DIGITS
DIMP
DINTEGER
DISABLE
DISPLAY
DIV
DLGAMMA
DLN
DLOG
DMAX
DMIN
DNABS
DNOT
DONTWAIT
DOR
DROP
DSCALELEFT
DSCALERIGHT
DSCALERIGHTT
DSIN
DSINH
DSQRT
DTAN
DTANH
DUMP
EBCDIC
EMPTY
EMPTY4
EMPTY7

Type 2 Reserved Words, Continued

EMPTY8
ENABLE
ENTIER
EQL
EQV
ERF
ERFC
EXCHANGE
EXP
EXPORT
EXTERNAL
FILL
FIRST
FIRSTONE
FIRSTWORD
FIX
FORMAL
FORWARD
FREE
FREEZE
GAMMA
GEQ
GTR
HAPPENED
HEAD
HEX
IMAG
IMP
IN
INTEGERT
INTERRUPT
IS
ISNT
LB
LENGTH
LEQ
LIBERATE
LIBRARY
LINE
LINENUMBER
LINKLIBRARY
LISTLOOKUP
LN
LNGAMMA
LOCK
LOG
LSS
MASKSEARCH
MAX
MERGE
MESSAGESEARCHER
MIN
MOD
MONITOR
MUX
MYJOB
MYSELF
NABS
NEQ
NO
NORMALIZE
NOT
NUMERIC
OF
OFFSET
ON
ONES
OPEN
OR
OUTPUTMESSAGE
PICTURE
POTC
POTH
POTL
PROCESS
PROCESSID
PROCURE
PROGRAMDUMP
PURGE
RANDOM
RB
READ
READLOCK
REMAININGCHARS
REMOVEFILE
REPEAT
REPLACE
RESET
RESIZE
REWIND
RUN
SCALELEFT
SCALERIGHT
SCALERIGHTF
SCALERIGHTT
SCAN
SDIGITS
SECONDWORD
SEEK
SET
SETACTUALNAME
SIGN
SIN
SINGLE
SINH
SIZE
SKIP
SORT
SPACE
SQRT
STACKER
STATION
STOP
STRING
STRING4
STRING7
STRING8
SUBFILE
TAIL
TAKE
TAN
TANH
THRU
TIME
TIMELIMIT
TIMES
TO
TRANSLATE
WAIT
WAITANDRESET
WHEN
WITH
WORDS
WRITE

Type 3 Reserved Words

ACTUALNAME
ALL
ALPHA6
ALPHA7
ALPHA8
ANYFAULT
ARRAYS
AS
ASCIITOBCL
ASCIITOEBCDIC
ASCIITOHEX
BACKUPPREFIX
BASE
BCLTOASCII
BCLTOEBCDIC
BCLTOHEX
BYFUNCTION
BYTITLE
CHARGECODE
CLASS
CODE
COMPILETYPE
COREESTIMATE
CRUNCH
DBS
DECIMALPOINTISCOMMA
DECLAREDPRIORITY
DISCARD
DISK
DISKPACK
EBCDICTOASCII
EBCDICTOBCL
EBCDICTOHEX
ELAPSEDTIME
EXCEPTIONEVENT
EXCEPTIONTASK
EXPONENTOVERFLOW
EXPONENTUNDERFLOW
FAMILY
FILECARDS
FILES
FUNCTIONNAME
HEXTOASCII
HEXTOBCL
HEXTOEBCDIC
HISTORY
INITIATOR
INTEGEROVERFLOW
INTNAME
INVALIDADDRESS
INVALIDINDEX
INVALIDOP
INVALIDPROGRAMWORD
JOBNUMBER
LIBACCESS
LIBPARAMETER
LIBRARIES
LOCKED
LOOP
MAXCARDS
MAXIOTIME
MAXLINES
MAXPROCTIME
MEMORYPARITY
MEMORYPROTECT
NAME
OFFER
OPTION
ORGUNIT
OUT
PACK
PAGED
PARTNER
PERMANENT
PRIVATELIBRARIES
PROCESSIOTIME
PROCESSTIME
PROGRAMMEDOPERATOR
REEL
RESTART
RETAIN
SCANPARITY
SIBS
STACKNO
STACKSIZE
STARTTIME
STATUS
STOPPOINT
STRINGPROTECT
SUBSPACES
TADS
TARGETTIME
TASKATTERR
TASKFILE
TASKVALUE
TEMPORARY
TITLE
TYPE
USERCODE
ZERODIVIDE

B DATA REPRESENTATION

FIELD NOTATION

The notation "[m:n]" is used in this manual to describe fields within data words. The 48 accessible bits of a data word are considered to be numbered, with the leftmost bit numbered as bit 47 and the rightmost bit numbered as bit 0. In the notation [m:n], "m" denotes the number of the leftmost bit of the field being described, and "n" denotes the number of bits in the field. For example, the field indicated by the shaded area in Figure B-1 (bits 28 through 24) is described as [28:5].

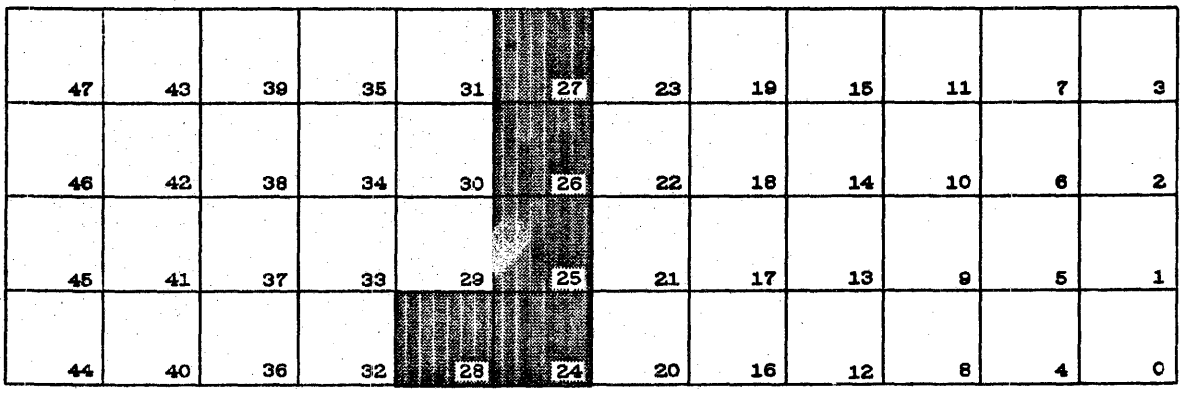


Figure B-1. Field Notation, [28:5]

All data words have a field associated with them that is called the "tag" of the word. The tag identifies the type of the data word; that is, whether the word is an operand, descriptor, and so on. The tag is not accessible to ALGOL programs.

Hexadecimal format is used extensively in this manual to indicate word contents. This format is particularly suited to describe the value of a data word, because each hexadecimal digit indicates the contents of a 4-bit field. Such fields can be visualized as the columns in Figure B-1.

CHARACTER REPRESENTATION

Characters are stored in fields of one, two, three, four, six, or eight bits. In the table below, the Character Type column lists the valid ALGOL character types. The Field Size and Bits Used columns show the field size in bits and the number of bits within that field, respectively, that are used for storing each type of character. The Valid Constructs column shows the character-oriented ALGOL constructs that are used to manipulate each character type.

Character Type	Field Size	Bits Used	Valid Constructs
EBCDIC	8	8	pointer, character array, string, string literal
ASCII	8	7	pointer, character array, string, string literal
BCL	6	6	pointer, character array, string literal
Hexadecimal (HEX)	4	4	pointer, character array, string, string literal
Octal	3	3	string literal
Quaternary	2	2	string literal
Binary	1	1	string literal

Figures B-2 through B-5 illustrate how the various character types are stored within a data word.

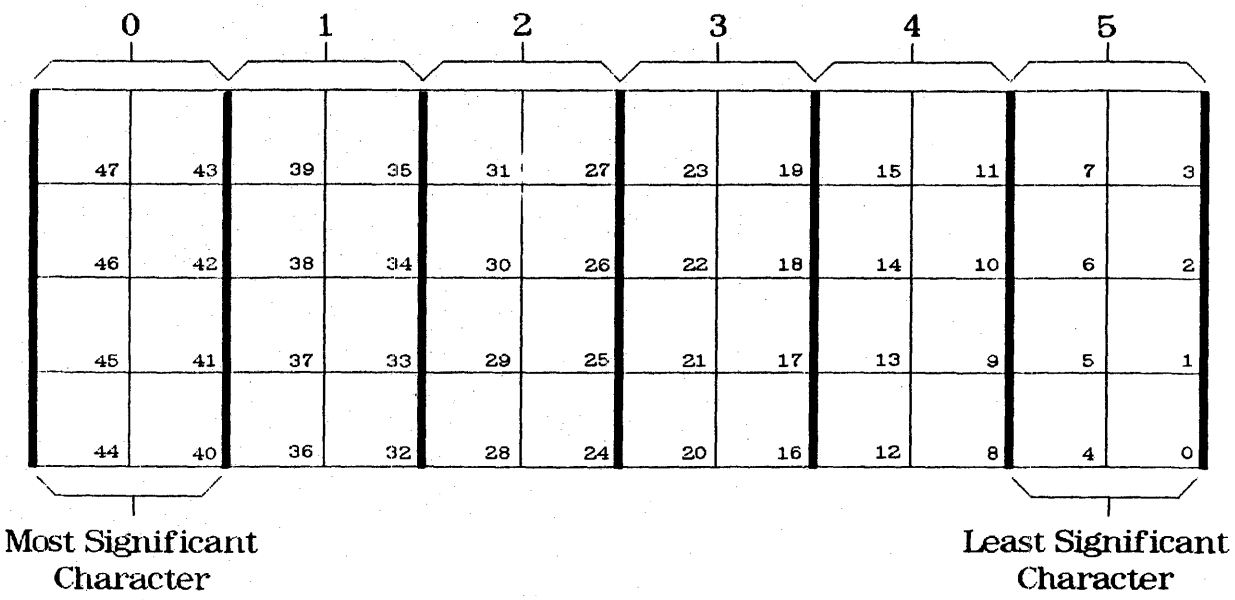


Figure B-2. EBCDIC Characters (8-Bit Fields)

Figure B-3 shows that ASCII characters, which are 7-bit characters, are stored in 8-bit fields (as are EBCDIC characters). The zeros shown for bits 7, 15, 23, 31, 39, and 47 indicate that these bits are 0 when ASCII characters are stored.

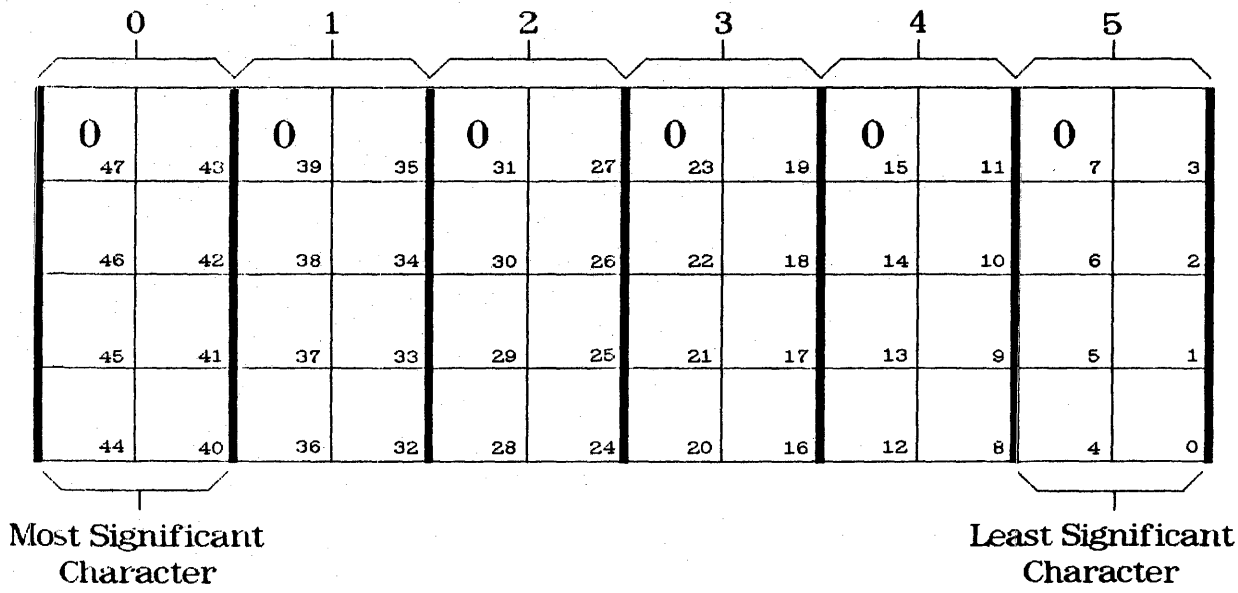


Figure B-3. ASCII Characters (8-Bit Fields)

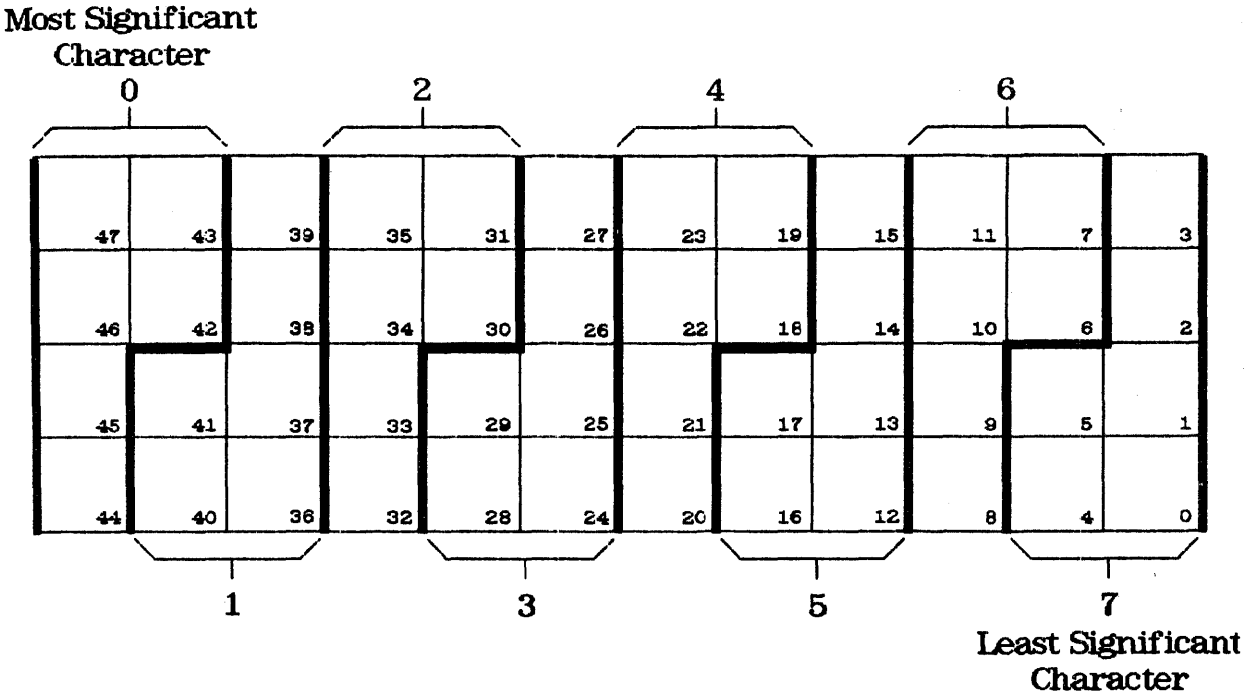


Figure B-4. BCL Characters (6-Bit Fields)

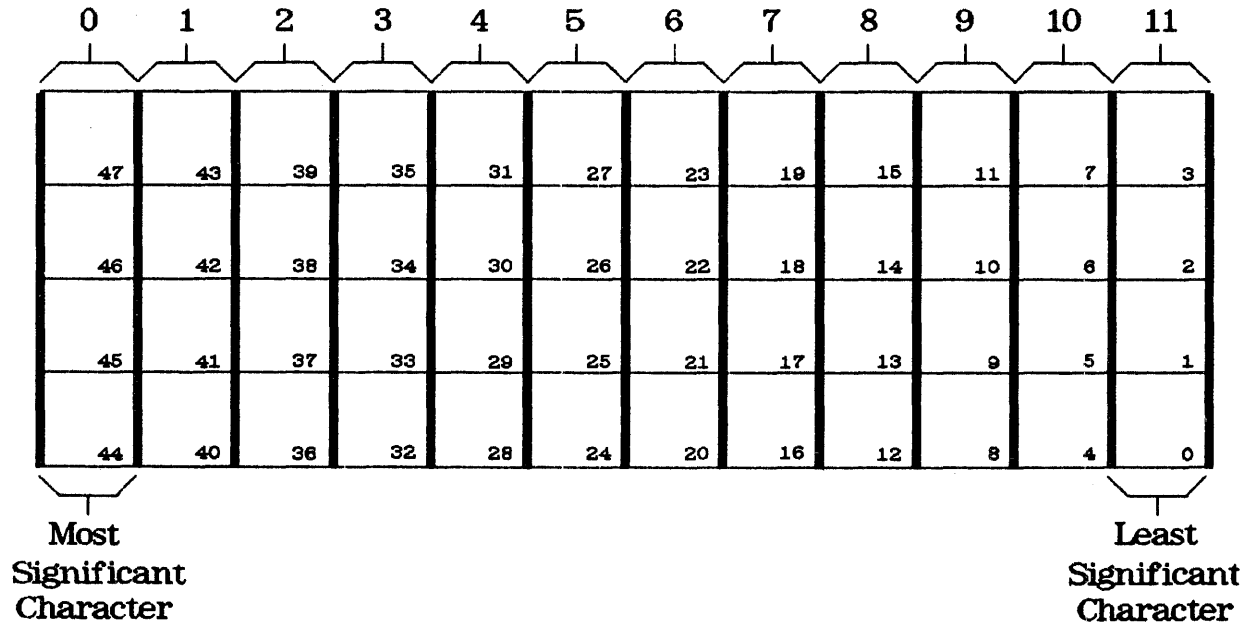


Figure B-5. Hexadecimal Characters (4-Bit Fields)

Character Values and Graphics

The following is a list of the 256 EBCDIC values represented in binary, octal, decimal, and hexadecimal formats. The corresponding BCL, ASCII, and EBCDIC graphics (as they appear when printed using an EBCDIC96 print train) are also shown. In the EBCDIC column, corresponding standard mnemonics appear for some nonprintable EBCDIC values.

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
00000000	000	0	00	0		NUL
00000001	001	1	01	1		SOH
00000010	002	2	02	2		STX
00000011	003	3	03	3		ETX
00000100	004	4	04	4		
00000101	005	5	05	5		HT
00000110	006	6	06	6		
00000111	007	7	07	7		DEL
00001000	010	8	08	8		
00001001	011	9	09	9		
00001010	012	10	0A	#		
00001011	013	11	0B	@		VT
00001100	014	12	0C	?		FF
00001101	015	13	0D	:		CR
00001110	016	14	0E	>		SO
00001111	017	15	0F	'		SI
00010000	020	16	10	+		DLE
00010001	021	17	11	A		DC1
00010010	022	18	12	B		DC2
00010011	023	19	13	C		DC3
00010100	024	20	14	D		
00010101	025	21	15	E		NL
00010110	026	22	16	F		BS
00010111	027	23	17	G		
00011000	030	24	18	H		CAN
00011001	031	25	19	I		EM
00011010	032	26	1A	.		
00011011	033	27	1B	[		
00011100	034	28	1C	&		FS
00011101	035	29	1D	(		GS
00011110	036	30	1E	<		RS
00011111	037	31	1F	!		US
00100000	040	32	20	}		
00100001	041	33	21	J	!	
00100010	042	34	22	K	"	
00100011	043	35	23	L	#	
00100100	044	36	24	M	$	
00100101	045	37	25	N	%	LF
00100110	046	38	26	O	&	ETB
00100111	047	39	27	P	'	ESC
00101000	050	40	28	Q	(	

Character Values and Graphics, continued

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
00101001	051	41	29	R	)	
00101010	052	42	2A	$	*	
00101011	053	43	2B	*	+	
00101100	054	44	2C	-	,	
00101101	055	45	2D	)	-	ENQ
00101110	056	46	2E	;	.	ACK
00101111	057	47	2F	^	/	BEL
00110000	060	48	30		0	
00110001	061	49	31	/	1	
00110010	062	50	32	S	2	SYN
00110011	063	51	33	T	3	
00110100	064	52	34	U	4	
00110101	065	53	35	V	5	
00110110	066	54	36	W	6	
00110111	067	55	37	X	7	EOT
00111000	070	56	38	Y	8	
00111001	071	57	39	Z	9	
00111010	072	58	3A	,	:	
00111011	073	59	3B	%	;	
00111100	074	60	3C	_	<	DC4
00111101	075	61	3D	=	=	NAK
00111110	076	62	3E	]	>	
00111111	077	63	3F	"	?	SUB
01000000	100	64	40		@	SP (blank)
01000001	101	65	41		A	
01000010	102	66	42		B	
01000011	103	67	43		C	
01000100	104	68	44		D	
01000101	105	69	45		E	
01000110	106	70	46		F	
01000111	107	71	47		G	
01001000	110	72	48		H	
01001001	111	73	49		I	
01001010	112	74	4A		J	[
01001011	113	75	4B		K	.
01001100	114	76	4C		L	<
01001101	115	77	4D		M	(
01001110	116	78	4E		N	+
01001111	117	79	4F		O	!
01010000	120	80	50		P	&
01010001	121	81	51		Q	
01010010	122	82	52		R	
01010011	123	83	53		S	
01010100	124	84	54		T	
01010101	125	85	55		U	
01010110	126	86	56		V	
01010111	127	87	57		W	
01011000	130	88	58		X	
01011001	131	89	59		Y	

Character Values and Graphics, continued

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
01011010	132	90	5A		Z	]
01011011	133	91	5B		[	$
01011100	134	92	5C		\	*
01011101	135	93	5D		]	)
01011110	136	94	5E		^	;
01011111	137	95	5F		_	^
01100000	140	96	60		'	-
01100001	141	97	61		a	/
01100010	142	98	62		b	
01100011	143	99	63		c	
01100100	144	100	64		d	
01100101	145	101	65		e	
01100110	146	102	66		f	
01100111	147	103	67		g	
01101000	150	104	68		h	
01101001	151	105	69		i	
01101010	152	106	6A		j	|
01101011	153	107	6B		k	,
01101100	154	108	6C		l	%
01101101	155	109	6D		m	_
01101110	156	110	6E		n	>
01101111	157	111	6F		o	?
01110000	160	112	70		p	
01110001	161	113	71		q	
01110010	162	114	72		r	
01110011	163	115	73		s	
01110100	164	116	74		t	
01110101	165	117	75		u	
01110110	166	118	76		v	
01110111	167	119	77		w	
01111000	170	120	78		x	
01111001	171	121	79		y	'
01111010	172	122	7A		z	:
01111011	173	123	7B		{	#
01111100	174	124	7C		|	@
01111101	175	125	7D		}	'
01111110	176	126	7E		~	=
01111111	177	127	7F			"
10000000	200	128	80			
10000001	201	129	81			a
10000010	202	130	82			b
10000011	203	131	83			c
10000100	204	132	84			d
10000101	205	133	85			e
10000110	206	134	86			f
10000111	207	135	87			g
10001000	210	136	88			h
10001001	211	137	89			i
10001010	212	138	8A			

Character Values and Graphics, continued

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
10001011	213	139	8B			
10001100	214	140	8C			
10001101	215	141	8D			
10001110	216	142	8E			
10001111	217	143	8F			
10010000	220	144	90			
10010001	221	145	91			j
10010010	222	146	92			k
10010011	223	147	93			l
10010100	224	148	94			m
10010101	225	149	95			n
10010110	226	150	96			o
10010111	227	151	97			p
10011000	230	152	98			q
10011001	231	153	99			r
10011010	232	154	9A			
10011011	233	155	9B			
10011100	234	156	9C			
10011101	235	157	9D			
10011110	236	158	9E			
10011111	237	159	9F			
10100000	240	160	A0			
10100001	241	161	A1			~
10100010	242	162	A2			s
10100011	243	163	A3			t
10100100	244	164	A4			u
10100101	245	165	A5			v
10100110	246	166	A6			w
10100111	247	167	A7			x
10101000	250	168	A8			y
10101001	251	169	A9			z
10101010	252	170	AA			
10101011	253	171	AB			
10101100	254	172	AC			
10101101	255	173	AD			
10101110	256	174	AE			
10101111	257	175	AF			
10110000	260	176	B0			
10110001	261	177	B1			
10110010	262	178	B2			
10110011	263	179	B3			
10110100	264	180	B4			
10110101	265	181	B5			
10110110	266	182	B6			
10110111	267	183	B7			
10111000	270	184	B8			
10111001	271	185	B9			
10111010	272	186	BA			
10111011	273	187	BB			

Character Values and Graphics, continued

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
10111100	274	188	BC			
10111101	275	189	BD			
10111110	276	190	BE			
10111111	277	191	BF			
11000000	300	192	C0			{
11000001	301	193	C1			A
11000010	302	194	C2			B
11000011	303	195	C3			C
11000100	304	196	C4			D
11000101	305	197	C5			E
11000110	306	198	C6			F
11000111	307	199	C7			G
11001000	310	200	C8			H
11001001	311	201	C9			I
11001010	312	202	CA			
11001011	313	203	CB			
11001100	314	204	CC			
11001101	315	205	CD			
11001110	316	206	CE			
11001111	317	207	CF			
11010000	320	208	D0			}
11010001	321	209	D1			J
11010010	322	210	D2			K
11010011	323	211	D3			L
11010100	324	212	D4			M
11010101	325	213	D5			N
11010110	326	214	D6			O
11010111	327	215	D7			P
11011000	330	216	D8			Q
11011001	331	217	D9			R
11011010	332	218	DA			
11011011	333	219	DB			
11011100	334	220	DC			
11011101	335	221	DD			
11011110	336	222	DE			
11011111	337	223	DF			
11100000	340	224	E0			\
11100001	341	225	E1			
11100010	342	226	E2			S
11100011	343	227	E3			T
11100100	344	228	E4			U
11100101	345	229	E5			V
11100110	346	230	E6			W
11100111	347	231	E7			X
11101000	350	232	E8			Y
11101001	351	233	E9			Z
11101010	352	234	EA			
11101011	353	235	EB			
11101100	354	236	EC			

Character Values and Graphics, continued

Binary	Octal	Decimal	Hexadecimal	BCL	ASCII	EBCDIC
11101101	355	237	ED			
11101110	356	238	EE			
11101111	357	239	EF			
11110000	360	240	F0			0
11110001	361	241	F1			1
11110010	362	242	F2			2
11110011	363	243	F3			3
11110100	364	244	F4			4
11110101	365	245	F5			5
11110110	366	246	F6			6
11110111	367	247	F7			7
11111000	370	248	F8			8
11111001	371	249	F9			9
11111010	372	250	FA			
11111011	373	251	FB			
11111100	374	252	FC			
11111101	375	253	FD			
11111110	376	254	FE			
11111111	377	255	FF			

Default Character Type

The default character type is the character type assumed by pointers, string variables, string literals, and so on when a character type is not explicitly specified. The default character type is also used as the default value of the INTMODE file attribute. For more information on INTMODE, refer to the "I/O Subsystem Reference Manual."

Two compiler control options affect the default character type: the ASCII option and the BCL option. (For more information, refer to the ASCII option and the BCL option under "<ASCII option>" and "<BCL option>," respectively, in the "Compiling Programs" chapter.) If the ASCII option is TRUE, the default character type is ASCII. If the BCL option is TRUE, the default character type is BCL. If neither the ASCII option nor the BCL option is TRUE, the default character type is EBCDIC.

Free-field and formatted I/O generate either EBCDIC or BCL data depending on the default character type. If the default character type is BCL, BCL data is generated. If the default character type is EBCDIC or ASCII, EBCDIC data is generated.

The example programs below demonstrate the effects of different default character types.

See also
<ASCII option>. 603
<BCL option>. 605

Examples

```
%%% PROGRAM 1 %%%
BEGIN
% OPTION RECORD
   FILE F(KIND=DISK,NEWFILE=TRUE);
   ARRAY A[0:9];
   POINTER P;
   STRING S;        % DECLARATION OF S
   OPEN(F);
   P := POINTER(A);
   S := "ABC";      % STRING ASSIGNMENT
END.
```

If this example is compiled and executed as is, the default character type is EBCDIC. Thus, the INTMODE attribute of file F is EBCDIC, the character size of pointer P is eight bits, the string type of string S is EBCDIC, and, after the string assignment is executed, S contains the EBCDIC string "ABC".

If "% OPTION RECORD" on the third line is replaced by the compiler control record "$ SET ASCII", and the program is compiled and executed, the default character type is ASCII. Thus, the INTMODE of file F is ASCII, the character size of pointer P is eight bits, the string type of string S is ASCII, and, after the string assignment is executed, S contains the ASCII string "ABC".

If program 1 is compiled after "% OPTION RECORD" is replaced by the compiler control record "$ SET BCL", the default character type is BCL and the program is flagged with two syntax errors--one on the declaration of string S and one on the string assignment--because string variables cannot be of type BCL.

```
%%% PROGRAM 2 %%%
BEGIN
$ SET BCL
   FILE F(KIND=DISK,NEWFILE=TRUE);
   ARRAY A[0:9];
   POINTER P;
   OPEN(F);
   P := POINTER(A);
   REPLACE P BY "ABC";    % REPLACE STATEMENT
END.
```

Program 2 contains the compiler control record "$ SET BCL" and has no string constructs. It compiles without error. If program 2 is compiled and executed, the default character type is BCL. Thus, the INTMODE of file F is BCL, the character size of pointer P is six bits, and, after the REPLACE statement is executed, P references the BCL characters "ABC" stored in array A.

Program 2 is flagged with a warning message when it is compiled, because the BCL character type is not supported on all A Series and B 5000/B 6000/B 7000 Series systems.

Signs of Numeric Fields

Certain operations in ALGOL require an indication of a numeric sign in character data. The sign of a numeric field is represented as follows:

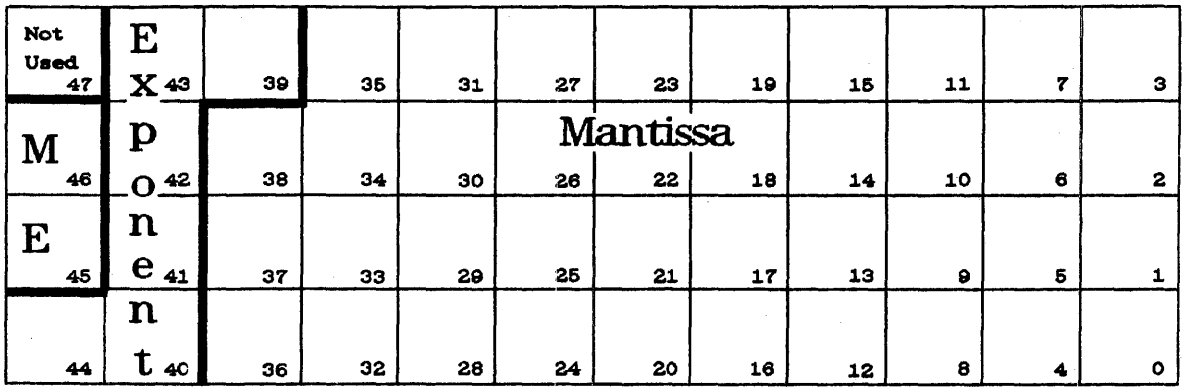
8-bit characters	The sign is in the zone field of the least significant character (field [7:4] of the character). A bit configuration of 1"1101" (4"D") indicates a negative number; any other bit configuration indicates a positive number.
6-bit characters	The sign is in the zone field of the least significant character (field [5:2] of the character). A bit configuration of 1"10" indicates a negative number; any other bit configuration indicates a positive number.
4-bit digits	The sign is carried as a separate digit, and it is the most significant digit of the field. A bit configuration of 1"1101" (4"D") indicates a negative number; any other bit configuration indicates a positive number.

ONE-WORD OPERAND

Real, integer, and Boolean operands each require one word of storage.

Real Operand

The internal structure of a real operand is illustrated in Figure B-6.



Field	Use
Tag	0
[47:1]	Not used
[46:1]	Sign of mantissa: 1 = negative, 0 = positive
[45:1]	Sign of exponent: 1 = negative, 0 = positive
[44:6]	Exponent
[38:39]	Mantissa

Figure B-6. Real Operand

Real (single-precision, floating-point) values are represented internally in signed-magnitude, mantissa-and-exponent notation. The sign of the mantissa is contained in bit 46, and the sign of the exponent is contained in bit 45. A minus sign is denoted by a 1 in the appropriate sign bit. The magnitude of the exponent is contained in field [44:6]; hence, the maximum absolute value of an exponent in a real operand is 2**6-1 = 63. The magnitude of the mantissa is contained in field [38:39]; hence, the maximum absolute value of a mantissa in a real operand is 2**39-1. There is an implied radix point to the right of bit zero.

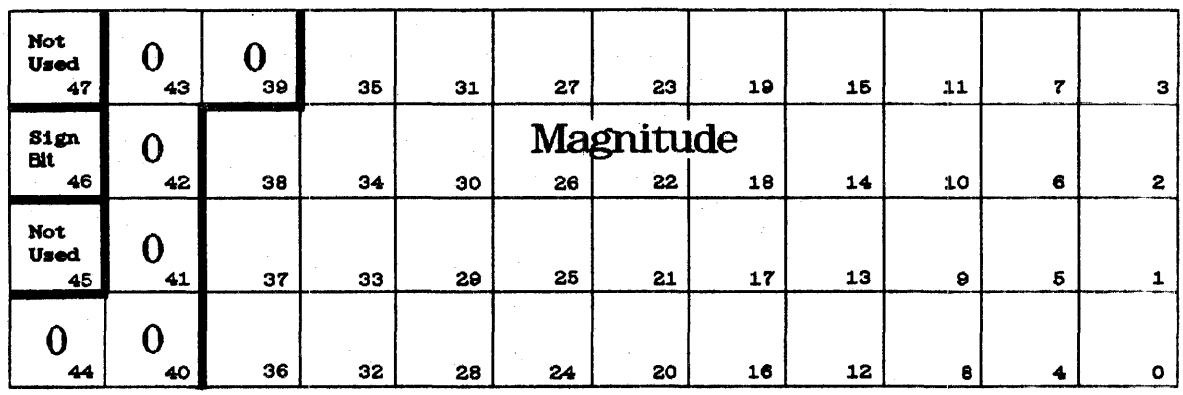
The value represented by a real operand can be obtained by the following formula:

```
(mantissa) * 8**(exponent)
```

Certain operations, such as the NORMALIZE function, return a "normalized" real operand as their result. A normalized real operand is a real operand in which the leftmost octade (3-bit field) of the mantissa is nonzero. For example, the real value 0.5, in normalized form, is represented internally as 4"26C000000000". In this example, the mantissa is 4 * 8**12 and the exponent is -13.

Integer Operand

The internal structure of an integer operand is illustrated in Figure B-7.



Field	Use
Tag	0
[47:1]	Not used
[46:1]	Sign: 1 = negative, 0 = positive
[45:1]	Not used
[44:6]	0
[38:39]	Magnitude

Figure B-7. Integer Operand

Integer values are represented internally in signed-magnitude notation. The sign of the value is denoted by bit 46 of the data word. A minus sign is denoted by a 1 in bit 46. The magnitude of the value is stored in field [38:39]. The maximum absolute value of an integer operand is 2**39-1. There is an implied radix point to the right of bit zero.

For example, the internal representation of the integer 10 is

```
4"00000000000A"
```

The internal representation of -10 is

```
4"40000000000A"
```

The internal representation of 99,999,999,999 is

```
4"00174876E7FF"
```

The internal representation

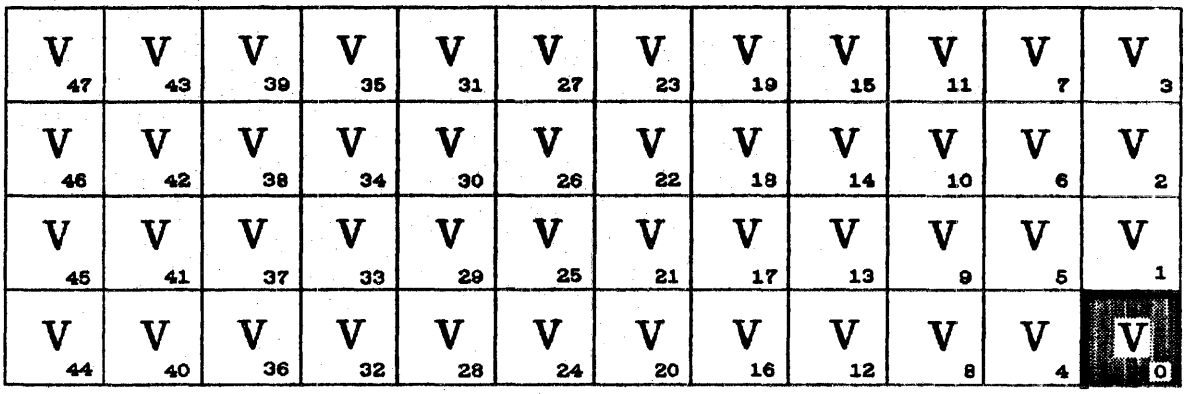
```
4"007FFFFFFFFF"
```

represents the decimal value 549,755,813,887 (the maximum value an integer operand can contain). A larger value would have to be stored in a real operand or a double-precision operand.

Note that the internal format of an integer operand is the same as the internal format of a real operand with an exponent of zero.

Boolean Operand

The internal structure of a Boolean operand is illustrated in Figure B-8.



Field	Use
Tag	0
[47:47]	Any bit or field within this field can be referenced as a Boolean value using the <partial word part> or <concatenation> constructs.
[0:1]	The Boolean value of the operand as a whole.

Figure B-8. Boolean Operand

Boolean operations are performed on a bit-by-bit basis on all 48 bits of a Boolean operand. The one exception is the NOT operation performed on an arithmetic relation, where NOT is performed on the low-order bit (bit zero), but not necessarily on the other 47 bits. However, when a Boolean operand is referenced as a whole, only the low-order bit (bit zero) is significant (0 is FALSE, 1 is TRUE).

In Figure B-8, the shaded V's indicate that each bit of the entire 48-bit Boolean operand can be used to store an individual Boolean value. The individual bits can be referenced by using the <partial word part> and <concatenation> constructs.

TWO-WORD OPERAND

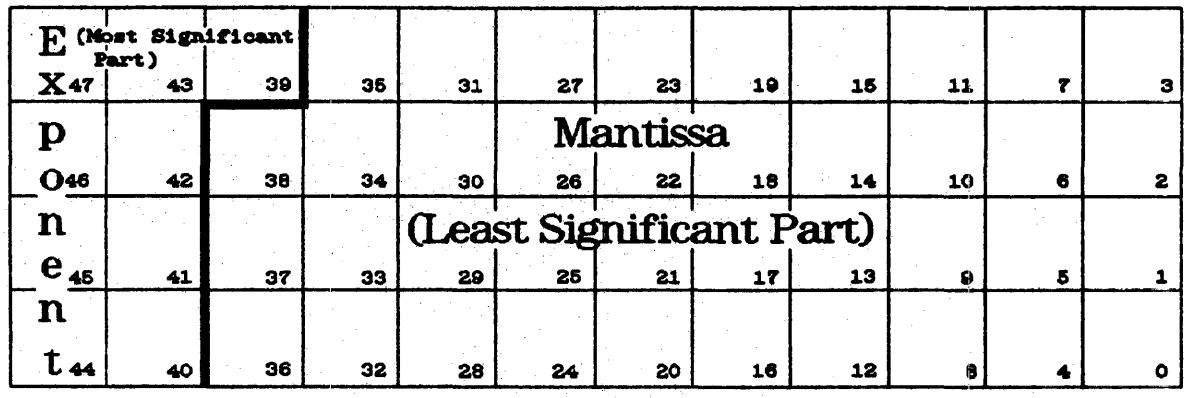
Double-precision and complex operands each require two words of storage.

Double-Precision Operand

The internal structure of a double-precision operand is illustrated in Figures B-9 and B-10.

Field	Use
Tag	2
[47:1]	Not used
[46:1]	Sign of mantissa: 1 = negative, 0 = positive
[45:1]	Sign of exponent: 1 = negative, 0 = positive
[44:6]	Least significant portion of the exponent
[38:39]	Most significant portion of the mantissa

Figure B-9. First Word, Double-Precision Operand



Field	Use
Tag	2
[47:9]	Most significant portion of the exponent
[38:39]	Least significant portion of the mantissa

Figure B-10. Second Word, Double-Precision Operand

Double-precision values are represented internally in signed-magnitude, mantissa-and-exponent notation. The sign of the mantissa is contained in bit 46 of the first data word, and the sign of the exponent is contained in bit 45 of the first data word. A minus sign is denoted by a 1 in the appropriate sign bit.

The magnitude of the exponent of a double-precision operand is contained in a total of 15 bits. The most significant nine of these 15 bits are contained in field [47:9] of the second data word. The least significant six of these 15 bits are contained in field [44:6] of the first data word.

The magnitude of the mantissa of a double-precision operand is represented by a total of 78 bits. The most significant 39 bits are contained in field [38:39] of the first data word. The least significant 39 bits are contained in field [38:39] of the second data word. There is an implied radix point to the right of bit zero of the first data word; that is, between the most significant and least significant parts of the mantissa.

The value represented by a double-precision operand can be obtained by the following formula:

```
(MM + LM * 8**(-13)) * 8**(ME * 2**6 + LE)
```

where

- MM is the most significant part of the mantissa.
- LM is the least significant part of the mantissa.
- ME is the most significant part of the exponent.
- LE is the least significant part of the exponent.

Certain operations return a "normalized" double-precision operand as their result. A normalized double-precision operand is a double-precision operand in which the leftmost octade (3-bit field) of the full 78-bit mantissa is nonzero. For example, the double-precision value 1@@0, in normalized form, is represented internally as

```
4"261000000000", 4"000000000000"
```

In this example, the mantissa is 1 * 8**12 and the exponent is -12.

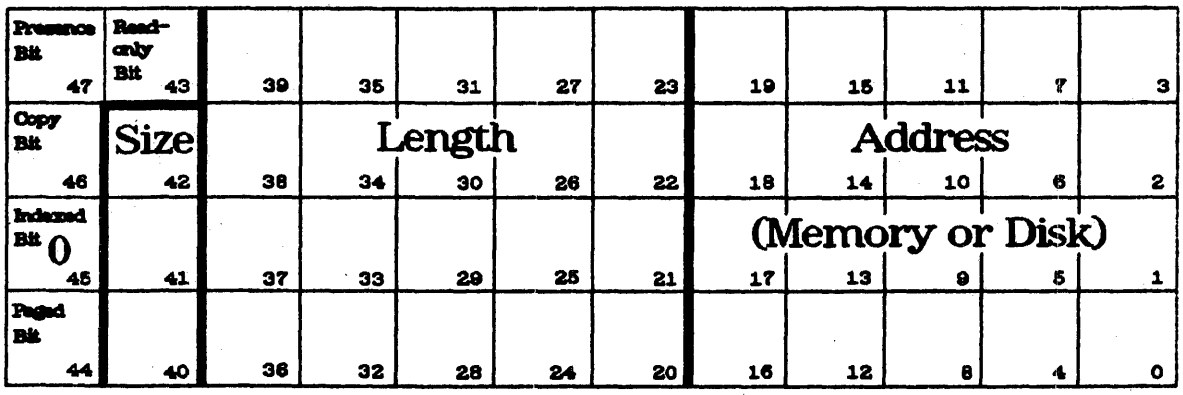
Complex Operand

A complex operand requires two words of storage. The first word contains the real part of the complex value; the second word contains the imaginary part. The internal structure of both the real and imaginary parts of the complex value is identical to the internal structure of a real operand. (For more information, refer to "Real Operand" in this appendix.)

DATA DESCRIPTORS AND POINTER

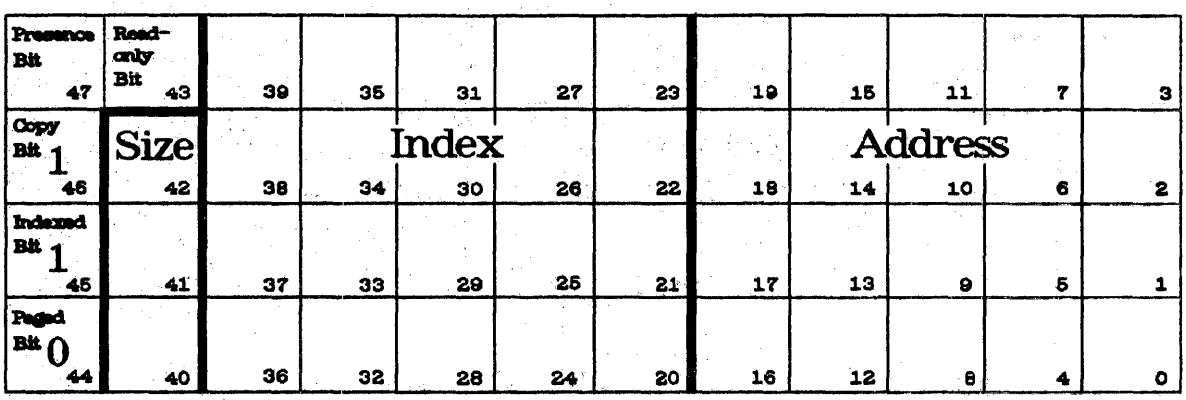
An unindexed data descriptor is the mechanism used on A Series and B 5000/B 6000/B 7000 Series systems to represent the contents of an array. An indexed data descriptor is used to reference one element of a word array. A pointer references one character within a word or character array.

The internal structures of unindexed data descriptors, indexed data descriptors, and pointers are illustrated in Figures B-11, B-12, and B-13, respectively.



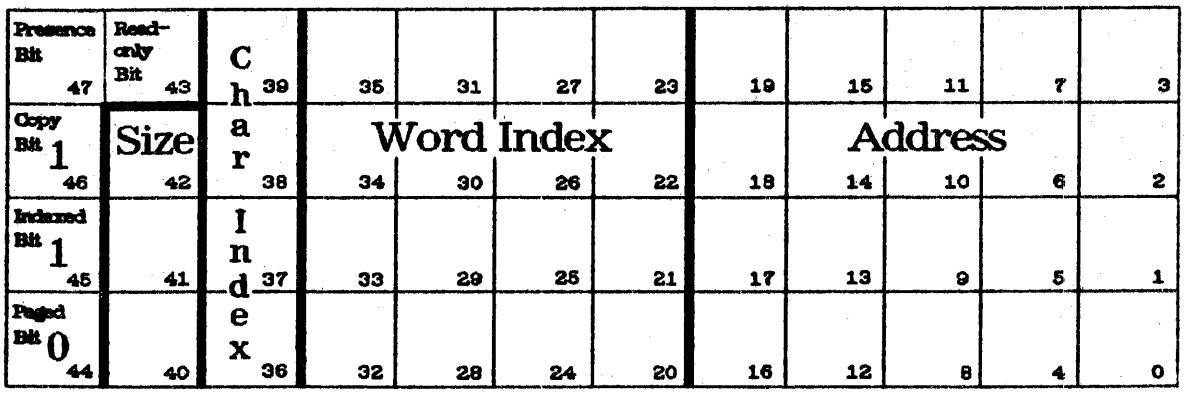
Field	Use
Tag	5
[47:1]	Presence bit: 1 = present, 0 = absent
[46:1]	Copy bit: 1 = copy, 0 = mom (original)
[45:1]	Indexed bit: 0 = unindexed
[44:1]	Paged bit: 1 = paged, 0 = unpaged
[43:1]	Read-only bit: 1 = read-only, 0 = read/write
[42:3]	Element size (type of the array): 0 = single precision, 1 = double precision, 2 = hexadecimal, 3 = BCL, 4 = EBCDIC/ASCII
[39:20]	Length (number of elements in the array)
[19:20]	Address: - When present, the memory address of the beginning of the array - When absent and mom, the software-encoded address of the array in storage - When absent and copy, the memory address of the associated mom descriptor

Figure B-11. Unindexed Data Descriptor



Field	Use
Tag	5
[47:1]	Presence bit: 1 = present, 0 = absent
[46:1]	Copy bit: 1 = copy
[45:1]	Indexed bit: 1 = indexed
[44:1]	Paged bit: 0 = unpaged
[43:1]	Read-only bit: 1 = read-only, 0 = read/write
[42:3]	Element size (type of the array): 0 = single precision, 1 = double precision
[39:20]	Index of the element in the array (relative to zero)
[19:20]	Address: - When present, the memory address of the beginning of the array - When absent, the memory address of the associated unindexed mom data descriptor

Figure B-12. Indexed Data Descriptor



Field	Use
Tag	5
[47:1]	Presence bit: 1 = present, 0 = absent
[46:1]	Copy bit: 1 = copy
[45:1]	Indexed bit: 1 = indexed
[44:1]	Paged bit: 0 = unpaged
[43:1]	Read-only bit: 1 = read-only, 0 = read/write
[42:3]	Element size (type of the pointer): 2 = hexadecimal, 3 = BCL, 4 = EBCDIC/ASCII
[39:4]	Character index: the zero-relative index in the word of the referenced character
[35:16]	Word index: the zero-relative index in the array of the referenced word
[19:20]	Address: - When present, the memory address of the beginning of the array - When absent, the memory address of the associated unindexed mom data descriptor

Figure B-13. Pointer

C RUN-TIME FORMAT-ERROR MESSAGES

Free-field Input

The meanings of the format-error numbers pertaining to free-field input are given in the following table.

Number	Error Message
400	An error occurred on free-field input.
416	Evaluation of a list element caused an I/O action on the current file.
420	Input from the ‹core-to-core file part› required more records than allowed by the ‹core-to-core blocking part›.
442	The input data corresponding to a single-precision list element consisted of a hexadecimal string of more than 12 significant digits.
443	The input data contained a hexadecimal string containing nonhexadecimal characters.
444	The input data corresponding to a double-precision list element consisted of a hexadecimal string of more than 24 significant digits.
462	The next list element was a pointer. and the corresponding input data consisted of a quoted string that had been only partially assigned.
467	The input data contained a value greater than the maximum value allowed for the corresponding list element.
473	The input data contained a string with no trailing quotation mark character.
484	An expression was used as a list element on input.

Formatted Output

The meanings of the format-error numbers pertaining to formatted output are given in the following table.

Number	Error Message
100	An error occurred on formatted output.
102	The editing phrase letter was V, and the data specified by the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, U, X, or Z in the appropriate character position.
103	The editing phrase was of the form rV, and the resulting specifier required a <field width>.
104	The editing phrase was of the form rV, and the resulting specifier required a <field width> and <decimal places>.
105	The editing phrase was of the form Fw.d, and d was less than zero.
106	The editing phrase used was Fw.d, and d was less than zero.
107	The editing phrase used was Ew.d or Dw.d, and d was less than zero.
109	The editing phrase used was Zw, and the corresponding list element was not of type INTEGER or BOOLEAN.
110	The type of the input data was not compatible with the type of the corresponding list element.
111	The editing phrase was of the form Zw.d. The phrase chosen to edit the output was Ew.d, but d was less than zero.
113	The editing phrase used was Ew.d or Dw.d, and w was less than or equal to d.
114	The value of a dynamic w or d part was greater than the maximum allowable integer (549,755,813,887).
116	Evaluation of a list element caused an I/O action on the current file (recursive I/O).
117	An attempt was made to write a number of characters greater than the record size.
120	Output to the <core-to-core file part> required more records than allowed by the <core-to-core blocking part>.

Run-Time Format-Error Messages

Formatted Output, continued

Number	Error Message
131	The value of a dynamic r part was greater than the maximum allowable real value (4.31359146673 * 10**68).
132	The value of a dynamic w part was greater than the maximum allowable integer (549,755,813,887).
133	The value of a dynamic d part was greater than the maximum allowable integer (549,755,813,887).
163	The record size was not large enough to allow the free-field write specified.

Formatted Input

The meanings of the format-error numbers pertaining to formatted input are given in the following table.

Number	Error Message
200	An error occurred on formatted input.
202	The editing phrase letter was V, and the data specified by the list element did not produce an A, C, D, E, F, G, H, I, J, K, L, O, P, R, S, T, U, X, or Z in the appropriate character position.
203	The editing phrase was of the form rV, and the resulting specifier required a <field width>.
204	The editing phrase was of the form rV, and the resulting specifier required a <field width> and <decimal places>.
205	The editing phrase was of the form rVw, and the resulting specifier required <decimal places>.
206	The editing phrase used was Fw.d, and d was less than zero.
207	The editing phrase used was Ew.d or Dw.d, and d was less than zero.
209	The editing phrase used was Zw, and the corresponding list element was not of type INTEGER or BOOLEAN.
210	The type of a list element was incompatible with the corresponding editing phrase.
213	The editing phrase used was Ew.d or Dw.d, and w was less than or equal to d.
214	The value of a dynamic w or d part was greater than the maximum allowable integer (549,755,813,887).
216	Evaluation of a list element caused an I/O action on the current file (recursive I/O).
217	An attempt was made to read a number of characters greater than the record size.
218	The editing phrase letter was H or K, but the input data contained nonblank, nonhexadecimal characters or nonblank, nonoctal characters, respectively.

Run-Time Format-Error Messages

Formatted Input, continued

Number	Error Message
220	Input from the <core-to-core file part> required more records than allowed by the <core-to-core blocking part>.
231	The value of a dynamic r part was greater than the maximum allowable real value (4.31359146673 * 10**68).
232	The value of a dynamic w part was greater than the maximum allowable integer (549,755,813,887).
233	The value of a dynamic d part was greater than the maximum allowable integer (549,755,813,887).
250	Input was attempted using a U editing phrase.
271	Input was attempted using a $ or P format modifier.
281	The input data was invalid for an I editing phrase.
284	An expression was used as a list element on input.
285	The list element was of type REAL, but the corresponding input data contained a value greater than the maximum allowable real value (4.31359146673 * 10**68).
286	The list element was of type INTEGER or BOOLEAN, but the input data contained a value greater than the maximum allowable integer (549,755.813,887).
291	The input data corresponding to a numeric editing phrase contained a nondigit character in the exponent part following at least one valid digit.
292	The input data corresponding to a numeric editing phrase contained more than one sign in the exponent part.
293	The input data corresponding to a numeric editing phrase contained an invalid character after the exponent sign and before the exponent value.
294	The input data corresponding to a numeric editing phrase contained an invalid character after the decimal point.
295	The input data corresponding to a numeric editing phrase contained more than one sign in the mantissa.

UNDERSTANDING RAILROAD DIAGRAMS

WHAT IS A RAILROAD DIAGRAM?

A railroad diagram is a way of representing the syntax of a command or statement graphically. It shows which items are required or optional, the order in which they should appear, how often you can repeat them, and any required punctuation.

HOW TO READ A RAILROAD DIAGRAM

Normally, you read a railroad diagram from left to right. However, there are some exceptions; in those cases, arrows indicate a right-to-left direction.

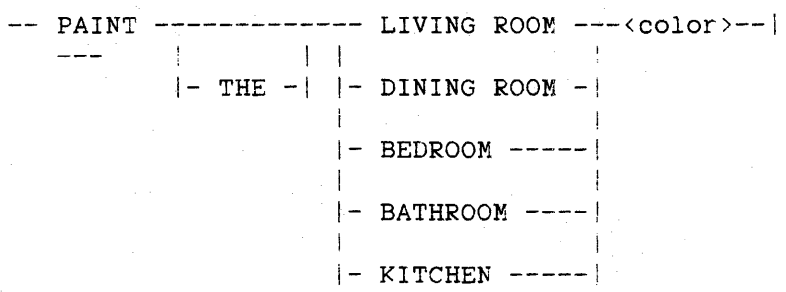
If a diagram is too long to fit on one line and must continue on the next, a right arrow (>) appears at the end of the first line and another at the beginning of the next line, like this:

```
-------------------------------------------------------------------------->

>-----------
```

The end of a railroad diagram is denoted by a vertical bar (|) or percent sign (%). The vertical bar means the command or statement can be followed by a semicolon and another command or statement. The percent sign means the command or statement must be on a line by itself.

CONSTANTS AND VARIABLES

Consider a hypothetical command for giving instructions to a house painter:



This command tells the painter to paint a designated room in the color you specify.

The example introduces two important features of railroad diagrams:

- Constants
- Variables

Constants

Constants are items that you cannot vary. You must enter a constant as it appears in the diagram, either in full or abbreviated. If you abbreviate a constant, you must enter everything that is underlined in the railroad diagram, optionally followed by one or more of the remaining characters.

You can recognize constants in railroad diagrams by the fact that they are never enclosed in angle brackets.

In the example, the word PAINT is a constant. You could enter PAINT in full or abbreviate it to PAI or PAIN, but not to PA or PAN. If no part of the constant is underlined, you cannot abbreviate it at all.

Variables

Variables are items that you can replace with other data to suit a particular situation; that is, you can vary the information you enter in place of the variable, subject to rules defined for the particular command or statement.

Variables appear in a railroad diagram enclosed in angle brackets (<>).

In the example, <color> is a variable item. If the description of the PAINT command defines the allowable colors as BLUE, GREEN, PINK, and YELLOW, you can enter any one of these in your command.

FOLLOWING THE PATHS OF A RAILROAD DIAGRAM

The paths of a railroad diagram lead you through the diagram from beginning to end. They are represented by horizontal and vertical lines.

A path shows the allowable syntax. Some diagrams have just one path that goes from the beginning to the end of the diagram. Others contain several paths, each covering a part of the diagram. A path shows which items you can include in a command or statement, which you can omit, and the number of times you can include a particular item or group of items.

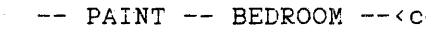
To follow a path through a railroad diagram, you need to understand the items you may encounter along the way. These items are

- Required items and punctuation
- Optional items
- Loops

A description of each item follows.

Required Items and Punctuation

Required items and punctuation must be entered in the command or statement: you cannot omit them. A required item appears by itself in a path (horizontal line). A required item can be either a constant or a variable. For example, if a railroad diagram indicates



the words PAINT and BEDROOM are required constants, and <color> is a required variable. You could correctly enter

```
PAINT BEDROOM BLUE
```

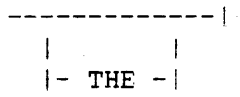
but not

```
PAINT BEDROOM
```

because the required item <color> would be missing.

Optional Items

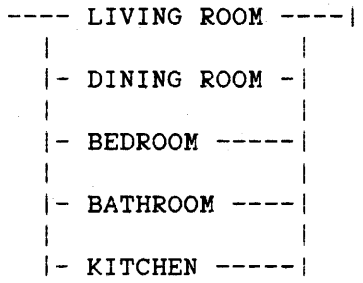
Optional items appear one below another in a vertical list. You can choose any one of the items in the list. If the list also contains an empty path (all dashes), you can omit the item entirely. An optional item can be either a variable or a constant. The PAINT command in the example contains two lists. The first is



which gives you two options:

- Enter the constant THE
- Omit it (because there is an empty path)

The second list has five optional constants:



You must enter one of the optional items (LIVING ROOM, DINING ROOM, BEDROOM, BATHROOM, or KITCHEN) because there is no empty path in this list.

Loops

A loop is an item or group of items that you can repeat. The number of repetitions allowed is controlled by an item called the bridge.

A loop can span all or part of a railroad diagram. It always consists of at least two horizontal lines, one below the other, like this:

```
      |<------- <return character> ---------|
      |                                      |
    -----<bridge>--<content of the loop>-------
```

or

```
      |<-- <bridge> -- <return character> --|
      |                                     |
    --------- <content of the loop> -----------
```

The bridge shows the maximum number of times you can go through the loop. The bridge can precede the contents of the loop, or it can precede the return character on the upper line of the loop to specify the number of times the right-to-left path can be traversed. The bridge is an integer enclosed in sloping lines, / \, for example, /4\. Not all loops have bridges. Those that do not can be repeated any number of times.

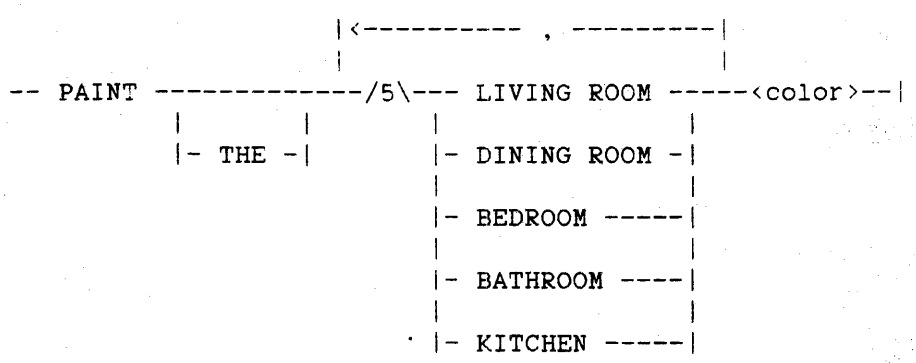
The top line is a right-to-left path that contains information about repeating the loop. The return character is the character to use before each repetition of the loop (often, a comma). Not all loops contain a return character; if none is shown, just enter one or more spaces before repeating the loop.

The other lines show the content of the loop (the data you enter each time you go through the loop). This can be any combination of optional items, required items, lists, and even other loops. The content of a loop can range from simple (one item), to very complex (many items, lists, and loops).

Example 1. A Simple Loop

The PAINT command as first shown is of limited usefulness. To tell the painter to do several rooms, you need a separate command for each room. It would be much easier if you could tell him to do several rooms in one command.

You can do that by making the list of rooms into a loop. The command would then look like this:



The bridge has a value of 5, so you can go through the loop up to five times, for a total of five rooms. The return character is a comma, which you must enter before repeating the loop content.

You can now enter

```
PAINT THE LIVING ROOM, BEDROOM, KITCHEN YELLOW
```

or

```
PAINT DINING ROOM, BEDROOM, BATHROOM BLUE
```

or

```
PAINT BEDROOM PINK
```

or

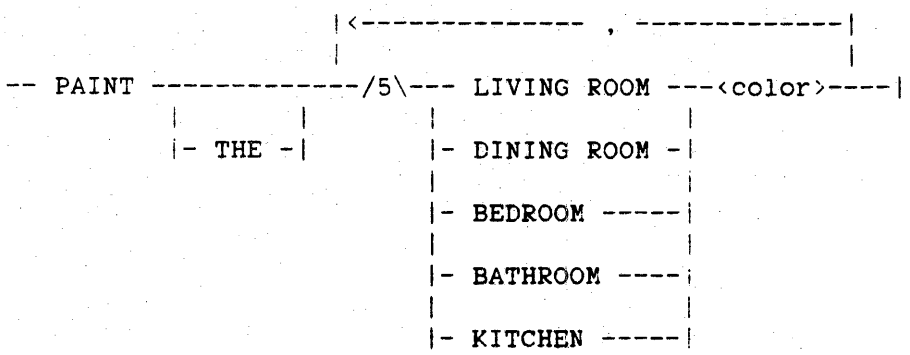
```
PAINT BEDROOM, BATHROOM, BEDROOM, BEDROOM BLUE
```

or any other valid combination.

This simple loop makes the PAINT command more versatile, but a significant drawback remains. Although you can include up to five rooms in a command, you cannot specify different colors.

Example 2. A More Complex Loop

If the content of the loop were to include the color, you could specify a different color for each room.



The content of the loop now consists of the

- List of optional constants that indicate rooms
- Required variable <color>

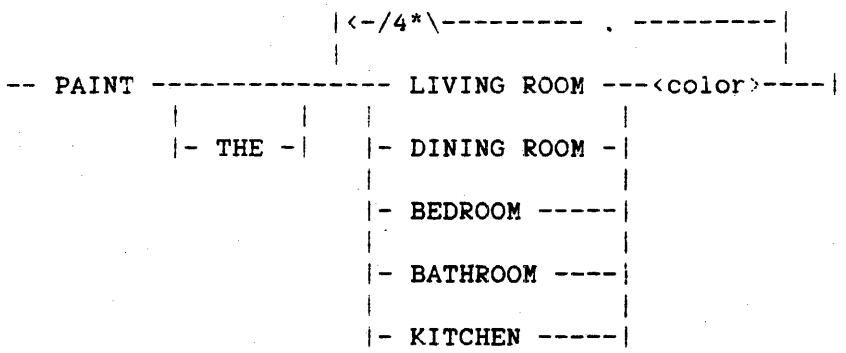
The bridge value is 5, and the return character is a comma. Given this railroad diagram, some valid PAINT commands would be

```
PAINT THE BEDROOM PINK

PAINT THE LIVING ROOM BLUE, DINING ROOM GREEN, KITCHEN YELLOW

PAINT BEDROOM GREEN, KITCHEN BLUE
```

and so on.

Example 3. Another Loop

In some bridges an asterisk follows the number. For example,



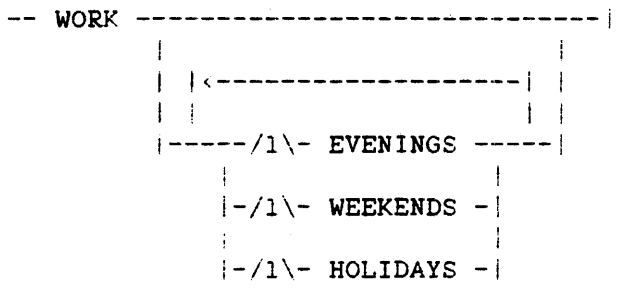
The asterisk means you must take the right-to-left path at least once. You cannot, for example, enter PAINT BEDROOM BLUE; you must tell the painter at least two rooms to paint. The maximum number of rooms to be painted is still five: the first time through the loop with up to four repetitions.

A valid form of the command would be

```
PAINT BEDROOM BLUE, KITCHEN YELLOW
```

Example 4. Another Use of the Bridge

A bridge can also control the number of times you take an individual path within a loop. For example, another command to the painter might be:



Each bridge /1\ indicates you can take that path once or not at all. That is, you can enter each of the items EVENINGS, WEEKENDS, and HOLIDAYS once at most. Some valid commands are

```
WORK EVENINGS WEEKENDS HOLIDAYS

WORK WEEKENDS

WORK HOLIDAYS EVENINGS
```

but

```
WORK EVENINGS EVENINGS
```

is invalid.

A FINAL WORD

To familiarize you with railroad diagrams, this explanation describes the elements of the diagrams and gives a few simplified examples. Some of the actual diagrams you will encounter in a book may be considerably more complex.

However, the principles are the same no matter how complex the diagram. The more you work with railroad notation, the more easily you will understand even the most complex diagrams.

GLOSSARY

accidental entry

See "thunk."

address couple

A representation of the address of an item in a program. An address couple consists of two numbers: the first number is a lexical level, and the second number is a displacement (offset) within that lexical level.

address equation

The process of declaring an identifier to have the same address as a previously declared identifier.

ASCII

American Standard Code for Information Interchange. A 7- or 8-bit code representing a set of 128 control and graphic characters.

asynchronous process

A procedure initiated by a program as a separate task that is dependent on the initiating program for globals but otherwise runs independent of and in parallel with the initiating program.

bad go to

A GO TO statement in an inner block that transfers control to a label that is global to that block. A necessary side effect of a "bad go to" is that the block in which it occurs is exited abruptly and local variables are deallocated immediately.

BCL

See "Burroughs Common Language."

Burroughs Common Language (BCL)

An obsolete code using 6-bit character representation. BCL is not available on A Series or on most B 5000/B 6000/B 7000 Series systems (such as B 5900 systems).

call-by-name

A method of passing a parameter to a procedure where every reference to the formal parameter within the procedure body causes the actual parameter to be evaluated. Any change made to the value of the formal parameter within the procedure body also changes the value of the corresponding actual parameter outside the procedure body.

call-by-reference

A method of passing a parameter to a procedure where the actual parameter's address is evaluated once and passed to the formal parameter. Every reference to the formal parameter within the procedure body thus references this address. Any change made to the value of the formal parameter within the procedure body also changes the value of the corresponding actual parameter outside the procedure body.

call-by-value

A method of passing a parameter to a procedure where the value of the actual parameter is assigned to the formal parameter, which is thereafter handled as a variable that is local to the procedure body. Any change made to the value of a call-by-value formal parameter has no effect outside the procedure body.

CANDE

Command AND Edit. A Burroughs Message Control System (MCS) that provides generalized file preparation and updating capabilities and task control in an interactive, terminal-oriented environment.

character array

An array whose elements are ASCII, BCL, EBCDIC, or hexadecimal characters.

code segment descriptor

A descriptor that references a code segment.

control variable

The variable that controls the repetitive execution of an iterative statement.

copy descriptor

A descriptor that has a one in the copy bit (bit 46). A copy descriptor is derived from a mom descriptor, and there can be multiple copy descriptors to the same data segment.

coroutine

A process in which two programs execute in parallel, but not simultaneously, passing control back and forth to each other and running alternately.

critical block

The block within a program initiating an asynchronous process or a coroutine that must not be exited until the process or coroutine finishes executing. As long as the critical block is not exited, the process or coroutine has access to necessary globals in the initiating program.

descriptor

A word, distinguished by its format, that is used to refer to data segments and code segments.

D1 stack (D[1] stack)

A stack created for each object code file that contains code segment descriptors and descriptors to read-only arrays. The D1 stack is also referred to as the "segment dictionary."

D2 stack (D[2] stack)

A stack initiated for each executing program that is used for storage of items declared in the outer block and other items allocated at lexical level two.

EBCDIC

Extended Binary Coded Decimal Interchange Code. An 8-bit code, capable of representing 256 graphic and control characters that are the native character set of A Series and B 5000/B 6000/B 7000 Series systems.

equivalent array

An array that is declared to refer to the same data as another array.

external sign flip-flop (EXTF)

A hardware flip-flop (a register that has two states) that is assigned values by and affects the actions of certain hardware operators. The state of EXTF reflects information about the signs of numeric quantities.

EXTF

See "external sign flip-flop."

fault

An error encountered by a hardware operator.

FIB

See "File Information Block."

file equation

A mechanism for specifying the values of file attributes when a program is compiled or executed.

File Information Block (FIB)

A data structure in an object code file that contains information describing a file.

float flip-flop (FLTF)

A hardware flip-flop (a register that has two states) that is assigned values by and affects the actions of certain hardware operators. The state of FLTF reflects information about the insertion of floating characters into character strings.

FLTF

See "float flip-flop."

fully specified formal procedure

A procedure parameter (formal procedure) that is declared with the word "FORMAL" in its declaration. With such procedures, the compiler checks the parameters of the actual procedure passed to it at compile time.

global identifier

Within a given block, an identifier that is declared in an outer block. A global identifier retains its values and characteristics as the blocks to which it is global are entered and exited.

Halt/Load

A system-initialization procedure that loads a fresh version of the Master Control Program (MCP) from disk or pack to main memory.

host program

A program to which separately compiled procedures can be bound by the Binder program or by using the sepcomp facility.

lex level

See "lexical level."

lexical level (lex level)

A number that indicates the relative level of a local addressing space within the stack of an executing program. The outer block of an ALGOL program is allocated at lexical level two. Procedures declared at the outer block level execute at lexical level three, procedures declared within those procedures execute at lexical level four, and so on, up to lexical level 15 or 31, depending on the computer family. The first number in an address couple is the lexical level of the item.

library

A program containing one or more procedures or "entry points" that can be called by other programs. Unlike a regular program, which is always entered at the beginning, a library can be entered at any of these entry points.

local identifier

Within a given block, an identifier that is declared in that block. The value or values associated with that identifier inside the block are not associated with that identifier outside the block. In other words, on entry to a block, the values of local identifiers are undefined; on exit from the block, the values of local identifiers are lost.

Master Control Program (MCP)

The operating system on A Series and B 5000/B 6000/B 7000 Series systems: the program that controls the operational environment of the system. This control includes memory management, job selection, peripheral management, system utilization, program segmentation, subroutine linkage, and error logging.

MCP

See "Master Control Program."

mom descriptor

A descriptor that has a zero in the copy bit (bit 46). For every data segment in a program, there is one and only one mom descriptor.

OFFF

See "overflow flip-flop."

original array

An array that is declared with a bound pair list. Each original array is distinct from all other original arrays.

overflow flip-flop (OFFF)

A hardware flip-flop (a register that has two states) that is assigned values by and affects the actions of certain hardware operators. The state of OFFF reflects whether or not an overflow occurred when a numeric quantity was converted to a character string of fixed length.

paged array

An array that is automatically divided ("paged" or "segmented") at run time into segments of 256 words each.

primary coroutine

A program that initiates a procedure as a coroutine.

ready queue

A queue of tasks that are capable of running. Tasks that are to begin executing are taken from this queue by the system.

referred array

An array identifier that--through array row equivalence, an array reference assignment, or because it is a formal array--refers to data in another array.

save memory

An area of memory that cannot be overlaid as long as the item with which it is associated is allocated.

scope

The portion of an ALGOL program in which an identifier can successfully be used to denote its corresponding values and characteristics.

secondary coroutine

A procedure that is initiated as a coroutine by a program.

segment dictionary

See "D1 stack."

segmented array

See "paged array."

separately compiled procedure

A procedure that is compiled on its own, rather than as part of a program, so that it can be bound into a host program by the Binder program.

SIB

See "Structure Information Block."

stack

A contiguous area in memory assigned to a task during its execution.

Structure Information Block (SIB)

On Mark 3.4 and earlier releases, a "pseudo-stack" that contains addressing environments for each structure of a database. SIBs do not exist on Mark 3.5 and later releases.

subroutine

A procedure to which program control is transferred when it is invoked and that transfers control back to the statement following the invocation statement when it is exited.

task

A single, complete unit of work performed by the system, such as compiling or executing a program or copying a file from one disk to another. Tasks are initiated by a job, by another task, or directly by a user.

thunk

A compiler-generated typed procedure that calculates and returns the value of an expression passed to a call-by-name formal parameter of a procedure. The value of the expression is calculated and returned each time the formal parameter is used. A thunk is also referred to as "accidental entry."

touched array

An array that has been referenced by a statement.

unpaged array

An array that is not automatically divided ("paged" or "segmented") at run time into segments of 256 words each. Arrays smaller than 1024 words are always unpaged.

unsegmented array

See "unpaged array."

up-level event

The situation that arises when either of the following is true:

1. The block containing an event is exited before the block containing the interrupt attached to the event is exited.

2. The block containing the finished event for a direct I/O statement is exited before the block containing the direct array is exited.

up-level pointer assignment

Any construct that could result in a pointer pointing to an array declared at a higher lexical level than that at which the pointer is declared. Such a construct is disallowed by the compiler, because the array can be deallocated, leaving the pointer pointing to an invalid portion of memory.

WFL

See "Work Flow Language."

word

A unit of computer memory. On Burroughs A Series and B 5000/B 6000/B 7000 Series systems, a word consists of 48 bits used for storage plus a tag field used to indicate how the word is interpreted.

word array

An array whose elements are single- or double-precision operands.

Work Flow Language (WFL)

The Burroughs language used to write jobs that control the flow of programs and tasks on the operating system.

zone field

The leftmost (high order) four bits of an ASCII or EBCDIC character or the leftmost two bits of a BCL character.