SMARTdata UTILITIES

SC26-7092-00

Data Description and Conversion
A Data Language Reference

Release 1

IBM

SMARTdata UTILITIES

SC26-7092-00

Data Description and Conversion
A Data Language Reference

Release 1

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page vii.

First Edition August 1995

This edition applies to the SMARTdata UTILITIES Release 1 function, and to all subsequent releases and modifications until otherwise indicated in new editions. Make sure you are using the correct edition for the level of the product.

Publications are *not* stocked at the address below. Requests for IBM publications should be made to your IBM representative or the IBM branch office serving your locality.

You can order by calling IBM Software Manufacturing Solutions at 1-800-879-2755.

A form for reader comments is provided at the back of this publication. If the form has been removed, address your comments to:

International Business Machines Corporation
RCF Processing Department
5600 Cottle Road
San Jose, CA 95193-0000
U.S.A.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

Trademarks and service marks

The following terms, denoted by an asterisk (*) in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

IBM	MVS/ESA
Operating System/2	OS/2
Operating System/400	OS/400
VM/ESA	

About This Book

This reference is written for application programmers who want to use the A Data Language (ADL) data description language to produce specifications of data to be converted. It explains how to use ADL to define the data encodings of various systems and programming languages. It also describes how to define the plans that convert data from one format into another.

The first part of the book explains the ADL concepts. The subsequent chapters explain the elements of the ADL declarations and plans and how they are used to describe a data types. You are shown how to convert between various numeric and alphanumeric data types. Appendix A gives some examples of ways to use ADL for data conversion.

Other appendixes explain the use of the DD&C User Exit, list the CCSID values used in converting data between languages of different countries, describe the formal ADL specification upon which this DD&C version of ADL is based, and demonstrate the Bachus Naur Form Syntax used in this book.

What You Should Know

This book assumes that you are familiar with Data Description and Conversion (DD&C) and that you have knowledge of the physical representation of data in the programming environments you intend to use.

Notation

The syntactic notation used in this book to describe ADL is an extended Backus Naur (or Normal) Form (BNF), as follows:

- Angle brackets (<>) enclose the identifier of a syntactic variable.
- ADL keywords are specified in uppercase.
- Syntactic units are specified as symbols, words, or literals.
- The symbol ::= means “is defined as”.
- The symbol | indicates an alternative.
- Ellipses (...) indicate elements that can be repeated one or more times.
- Square brackets ([ ]) indicate a group of one or more optional elements.
- Braces ({ }) indicate a group of one or more elements. For example:

  `{A | B | C } D` can be one of the following:

  ```
  A D
  B D
  C D
  ```

- The symbol m..n specifies a single integer value within the range m to n inclusive. Using 1..3 as an example, 1 is the first valid value in the range and 3 is the last valid value in the range.

- The symbol ! following a group, as in {A B}!, or an optional group, as in [A B]!, indicates an unordered list in which each of the elements in the group is specified once. For example, {A B}! can consist of:

  ```
  A B
  B A
  ```

  Optional elements need not be specified. For example, {[A] [B]}! can consist of:

  ```
  A
  B
  A B
  B A
  (neither of the two elements)
  ```

  If two or more optional groups are alternatives, all the elements of an optional group must be specified. For example, [A B]! | [A C]! can be one of the following:

  ```
  A B
  B A
  A C
  C A
  (neither of the two elements)
  ```

If you specify a list within a list, all items can be specified in any order. The following are examples of BNF notation that allow `A B C D` to be specified in any order:

```
{A B {C D}! }!
{{A B}! C D}!
{{A B}! {C D}!}!
{A {B C}! D}!
{A B C D}!
```

- Expressions enclosed in brackets or braces are evaluated first, before they are considered in relation to surrounding expressions.
- In an expression containing ellipses, a range, or alternatives, the priority of evaluation is range, ellipses, alternatives.

In the BNF syntax, a production symbol <A> is defined to *contain* a production symbol <B> if <B> occurs somewhere in the expansion of <A>. If <A> contains <B>, then <A> is the *containing* production symbol for <B>.

A production symbol <A> *immediately* contains a production symbol <B> if <A> contains <B> and if <B> occurs in the first expansion of <A>.

Conventions

The following conventions are used in this book:

1. The syntactic elements of ADL are specified in terms of:

Function	The element's purpose.
Syntax	The BNF definition of the element's syntax.
Syntax rules	Additional syntactic constraints not expressed in BNF that the element must satisfy.
General rules	Sequence of actions that define the semantics of the language.
Examples	Sample ADL descriptions that illustrate the syntactic elements described.

2. The term *space* refers to a contiguous sequence of storage locations in which data is represented.
3. Numeric expressions used in ADL rules or descriptions conform to the following conventions:
   - Operators are used between two values to indicate a mathematical operation between the two. Operators cannot be placed at the end of an expression.
   - Valid operators in numeric expressions are:

+	Addition
—	Subtraction
*	Multiplication
/	Division
**	Exponentiation.

- Except for the prefix operators – (unary minus), which indicates the negative value of a variable, +, which indicates a positive value, operators cannot be placed as the first character in an expression.
- Numeric expressions are evaluated from left to right. The precedence rule in evaluating numeric expressions is according to the priorities and governing rules in Table 1.

Table 1. Precedence rules for numeric expressions

Priority	Operator
1	**, unary -
2	*, /
3	+, -

  – Expressions enclosed in parentheses () are evaluated first, before they are considered in relation to surrounding operators.

  – Priority 1 has the highest priority and priority 3 the lowest.

  – All operators at the same priority level have the same priority.

  – For priority level 1, if two or more operators appear in the same expression, the order of priority is from right to left within the expression, that is, the rightmost exponentiation or unary minus operator has the highest priority, the next operator from the right has the next highest priority, and so on.

  – For all other priority levels, if two or more operators of the same level appear in an expression, the order of priority is from left to right within the expression.

- The following functions are also used:

CEIL	The ceiling integer value of a number. The value of this function is the smallest integer value that is greater than or equal to the number.
FLOOR	The floor integer value of a number. The value of this function is the largest integer value that is less than or equal to the number.
MIN	The value of this function is the lowest of the arguments specified.
MOD	The value of this function is the remainder of the first argument divided by the second argument.

For Y > 0 and X >= 0:

```
MOD(X,Y) = X - (Y * FLOOR(X/Y))
```

For Y > 0 and X < 0:

```
MOD(X,Y) = X - (Y * CEIL(X/Y))
```

PRODUCT The number or quantity that results from a multiplication of the arguments of the function.

- The keyword name of an attribute must be specified whenever reference is made to the current value of the attribute. For example, the current value of the <PRECISION attribute> is indicated by PRECISION in the following expression:

  ```
  (2**PRECISION)-1
  ```

Bibliography

You can order books by calling IBM Software Manufacturing Solutions at 1-800-879-2755.

Table 2. SMARTdata UTILITIES for AIX Publications

Publication Title	Order Number
SMARTdata UTILITIES for AIX Set	SBOF-6132
SMARTdata UTILITIES for AIX: VSAM in a Distributed Environment	SC26-7064
SMARTdata UTILITIES: SMARTsort for OS/2 and AIX	SC26-7099
SMARTdata UTILITIES for AIX: Data Description and Conversion	SC26-7066
SMARTdata UTILITIES: Data Description and Conversion A Data Language Reference	SC26-7092

Table 3. SMARTdata UTILITIES for OS/2 Publications

Publication Title	Order Number
SMARTdata UTILITIES for OS/2 Set	SBOF-6131
SMARTdata UTILITIES for OS/2: VSAM in a Distributed Environment	SC26-7063
SMARTdata UTILITIES: SMARTsort for OS/2 and AIX	SC26-7099
SMARTdata UTILITIES for OS/2: Data Description and Conversion	SC26-7091
SMARTdata UTILITIES: Data Description and Conversion A Data Language Reference	SC26-7092

Table 4 (Page 1 of 2). Other Publications

Publication Title	Order Number
DDM Architecture: Specifications for ADL	SC21-8286
Character Data Representation Architecture, Level 2	SC09-1390
IBM Systems Journal: Volume 31, No. 3, 1992	G321-5483
IBM Dictionary of Computing	SC20-1699
Compilers–Principles, Techniques, and Tools: by the Addison–Wesley Publishing Company	
IEEE Standard for Binary Floating–Point Arithmetic: ANSI/IEEE STANDARD	754-1985
INTEL 387™ DX User	
IBM Distributed Data Management: General Information	GC21-9527
IBM Distributed Data Management: Reference Guide	SC21-9526
Using Distributed Data Management for the IBM Personal Computer	SC21-9643
AS/400 Communications: Distributed Data Management Guide	SC21-9600
CICS/Distributed Data Management:User's Guide	SC33-0695

Table 4 (Page 2 of 2). Other Publications

Publication Title	**Order Number**
IBM 4680 Store Systems: Distributed Data Management: User's Guide	SC30-4915
DFSMS/MVS Version 1 Release 2 Distributed FileManager/MVS Guide and Reference	SC26-4915
AIX SNA Server/6000: Configuration Reference	SC31-7014
AIX SNA Server/6000: User's Guide	SC31-7002
AIX DCE Administration Guide	SC23-2475
Encina Server Administration: System Administrator's Guide and Reference for AIX	SC23-2461
Encina Structured File Server Administrator's Guide and Reference for AIX	SC23-2468

Chapter 1. Understanding ADL Concepts

A Data Language (ADL) describes the way data is encoded in various programming languages, in various formats for various operating systems. It assists in performing data conversion between disparate operating systems in a shared data environment. Using ADL, you create a **module** that bridges differences between data types and data encodings defined by a variety of programming languages.

For example, an MVS system whose programs are written in COBOL can communicate with an AIX or OS/2 system whose programs are coded in C language only when the data encodings can be understood by both systems. Figure 1 illustrates the difficulties of interlanguage and intersystem communication. It is the differences in programming languages and data types that make data conversion and ADL necessary.

```
C Data Description                      COBOL Data Description

 struct                                      01 RECORD.
    {                                            05 SALARY PICTURE 9(4) USAGE IS COMP -3.
       long salary;                              05 NAME PICTURE X(4) USAGE DISPLAY.
       char name[5];
    } RECORD;

OS/2 C Data Representation              MVS COBOL Data Representation

     |E0|2E|00|00|4A|4F|45|00|00|            |12|00|0C|D1|D6|C5|40|
     └──────────┘ └──────────────┘            └───────┘ └──────────┘
        SALARY         NAME                     SALARY      NAME
```

Figure 1. An example of the diversity of data encodings. For even a simple data structure, the data encodings used by different programming languages for different systems can vary widely. Many programming languages support the declaration of much more complex data structures.

Components

An ADL module consists of **declarations** and **plans**. Each ADL declaration provides a complete and consistent description of how data is encoded in a particular **representation domain**. Different declarations can exist in a module for the same data, with each description appropriate to a single representation domain. An ADL plan combines these alternative descriptions to create a conversion plan between two types of data encodings.

This manual tells you how to write ADL declarations to describe data formats, and how to write ADL plans that describe how to convert data from one format to another.

Following are descriptions of the components involved in data conversion. Refer to Figure 2 on page 4 for a visual representation of how the components fit into the total picture.

Representation Domain

A representation domain is defined by the ways the data types of a programming language are encoded in an implementation of the programming language for a particular machine architecture. Different implementations of a programming language result in different representation domains. Different representation domains usually represent different types of systems as well as different programming languages.

ADL Module

An ADL module contains the declarations that describe the data and the plans that describe how to convert the data from the format of one representation domain to that of another.

Declarations
: Declarations describe how the data is encoded in a particular implementation of a programming language in a representation domain. Using the description of ADL provided in this book, you can create ADL descriptions of data from existing programming language descriptions. For example, the ADL description of a SEQUENCE of fields can be derived from the COBOL description of a STRUCTURE. However, the representation of a COBOL STRUCTURE by a particular COBOL implementation must also be known. To create ADL data descriptions, therefore, you require detailed knowledge about how data is encoded in any particular representation domain.

  ViewRec describes the data format after conversion has been performed.

  BaseRec describes the data format before conversion.

Plans
: An ADL plan describes how to convert data from the format of the source data to the format of the target data, as described in their respective data declarations.

  The getplan and putplan are always described from the point of view of the local system.

  getplan describes the conversion of data from the remote to the local format.

  putplan describes the conversion of data from the local to remote format.

  A conversion plan can be carried out whenever data is to be transported between the exporting and importing representation domains. The data may be stored in a file or database between the time it is exported by one program and the time it is imported by another program. That data can be imported by programs from different representation domains, as long as a conversion plan exists for the representation domains involved.

A number of scenarios are described in Appendix A, “Scenarios” on page 175, including some that include using CALL statements to use conversion plans. Refer to Chapter 3 for a detailed description of the CALL statement.

DD&C ADL Declaration Translator

The declaration translator is invoked to **parse** declaration source files to create the ADLDCLSPC and ADLPLNSPC spaces. It can also **generate** the source text statements from existing ADLDCLSPC and ADLPLNSPC spaces.

FMTPRS (Parse)	The FMTPRS function parses the ADL data descriptions and ADL plans to produce the binary encodings of the source files. These will be used to perform data conversion. The encoded ADL statements are stored in areas of memory called ADLDCLSPC and ADLPLNSPC. These spaces are subsequently passed as input to the conversion plan builder component of DD&C.
FMTGEN (Generate)	The FMTGEN function generates ADL text from ADL declaration spaces and ADL plan spaces. This function can be used as a tool to check the contents of the spaces when they are to be updated or an error is suspected.

The DD&C Conversion Plan Builder

The conversion plan builder uses the ADLDCLSPC and ADLPLNSPC as input to create a conversion program. It uses the plan to map the conversion of the source data format to the format required by the target system.

DD&C Conversion Executor

When invoked, the conversion executor performs the actual data conversion, using as input the original data from the target system and the conversion plan that was created by the conversion plan builder. The resulting converted data is then available to the source system.

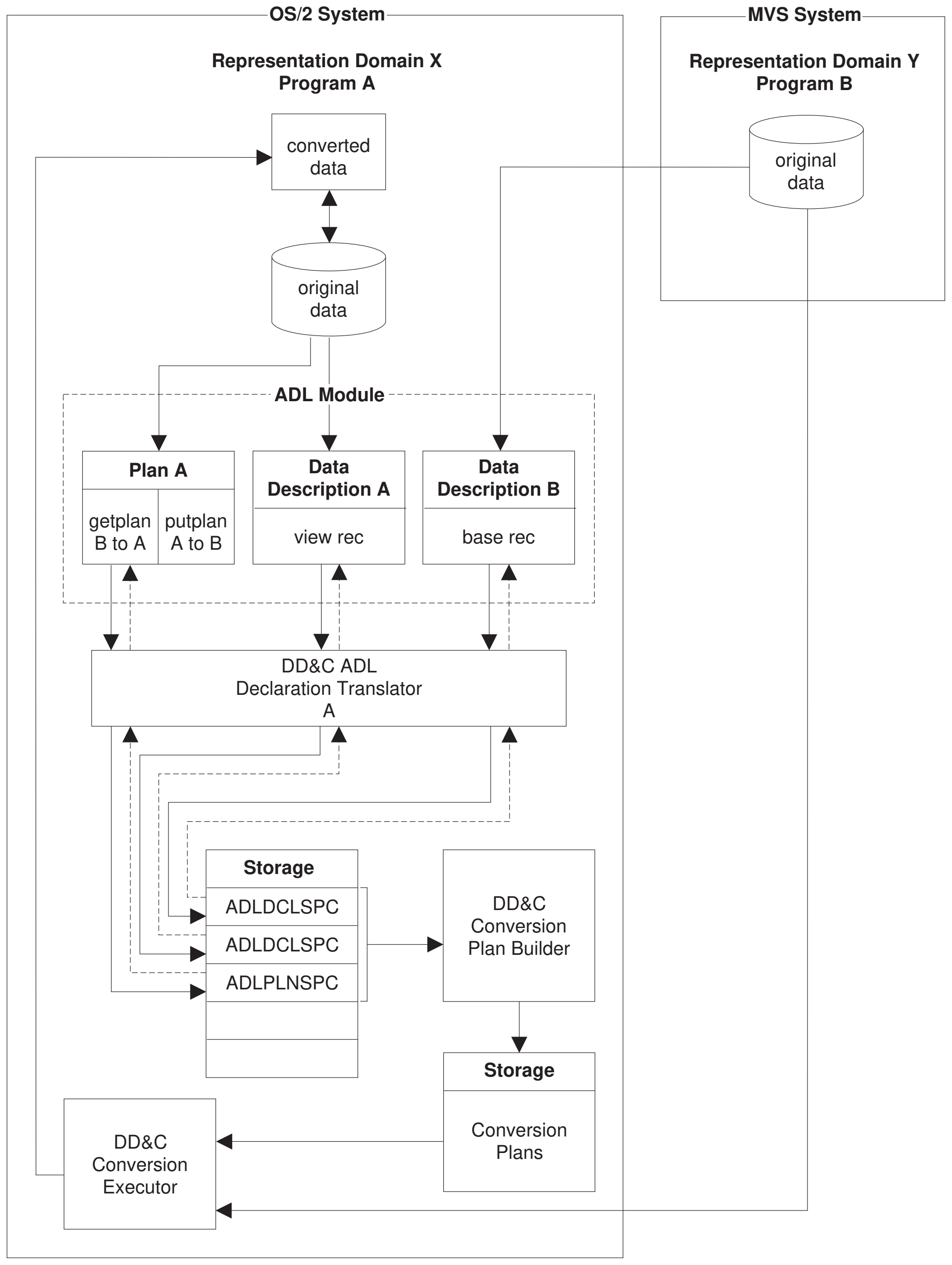


Figure 2. Components involved in DD&C data conversion

Example of ADL Use

The following example uses an MVS COBOL system and an OS/2 C language system to demonstrate how you use ADL to define the data formats.

The data:

```
12000, "JOE"
```

might be described in an MVS COBOL program as:

```
01 RECORD.
    05 SALARY PICTURE 9(4) USAGE IS COMP -3.
    05 NAME PICTURE X(4) USAGE DISPLAY.
```

and in a C program as follows:

```
struct
   {
      long salary;
      char name[5];
   } RECORD;
```

The following steps are necessary to convert records stored in an MVS system to an appropriate format for OS/2:

1 For both data descriptions, the record layout must first be described in ADL. This is done using two data declaration statements:

- BaseRec, describing the original MVS data record
- ViewRec, describing the data record for OS/2

The BaseRec data declaration is as follows:

```
DECLARE
BEGIN;
  BaseRec:  SEQUENCE
  BEGIN;
    salary: PACKED PRECISION(5);
    name:   CHAR LENGTH(4) CCSID(500);
  END;
END;
```

The ViewRec data declaration is as follows:

```
DECLARE
BEGIN;
  ViewRec:  SEQUENCE
  BEGIN;
    salary: BINARY PRECISION(31) BYTRVS(TRUE) ;
    name:   CHARSFX MAXALC(TRUE) MAXLEN(5) CCSID(437);
  END;
END;
```

Both the COBOL and the C records are represented as SEQUENCEs in ADL. Each SEQUENCE contains two fields. Since ADL can be used to describe many different data representations, each field is described using both its data type (CHAR, PACKED, CHARSFX, and BINARY in the example) and a number of *attributes*, which define the fields for each ADL data type more precisely.

2 To convert the records, an ADL plan must be defined. The plan describes how the records are converted from one format to the other. As DD&C contains the necessary conversion routines to convert ADL fields and structures, the plan describes the conversion at an abstract level only.

```
/* Conversion specification from base to view */

getPlan: PLAN (BaseRec: INPUT, ViewRec: OUTPUT)
         BEGIN;
            ViewRec <- BaseRec;
         END;
```

The plan consists of a reference to the declarations of the base and view record, and one assignment statement (`ViewRec <- BaseRec`). This assignment statement instructs DD&C to perform conversion of the structures.

For each field of the view record, the base record is searched for a field with the same name. If found, the field is converted.

In this example, the record consists of a character field and a numeric field. Conversion is performed separately for each data type:

- On the MVS system, the numeric field is represented as a packed field, whereas in OS/2 it is stored as a 4-byte binary number in byte-reversed format.
- The representation of each character of the character field is different on both systems. On the MVS system, characters are represented in an EBCDIC code page, whereas on OS/2, an ASCII code page is used. Also, the structure of the character field is different. On the MVS system, the string is padded with blanks, whereas on OS/2 the string is null-terminated according to the C convention.

3 The Parse function of the ADL declaration translator API is called to compile the ADL source text into a declaration space and a plan space. These spaces are input for the conversion plan builder component, which is then called to produce a conversion plan space. This space is used by the conversion plan executor component to convert the records.

ADL Syntax

The declarations and plans of an ADL module are specified as text in a source file. This text consists of a sequence of statements constructed according to the rules of ADL syntax. Following is an overview of ADL concepts and terminology as they are used in Chapter 2 through Chapter 7.

Statements

Each ADL statement has a body, and a terminator. Some types of statements can also have identifiers. For example, in the statement:

```
parts_count: BINARY PRECISION(15) SCALE(0);
     where:
```

the identifier is	`parts_count`
the body is	`BINARY PRECISION(A1) SCALE(0)`
the terminator is	**;**

you can use spaces and comments between the identifiers, keywords and punctuation of ADL statements. However, none are required except when they are used to separate identifiers and keywords from other tokens.

Comments

Comments can be placed in ADL text anywhere where a space could be placed. Comments consist of the character sequence:

```
/* comment */
```

The body of the comment can include any sequence of characters allowed by the source file's coded character set identifier (CCSID) , including non-syntactic characters. However, the character sequence */ cannot appear within a comment. Likewise, the sequence /* cannot appear within a comment.

Literals

You can specify literals appropriate to each of the data types defined by ADL as ADL tokens, including:

- numeric literals
- boolean literals
- character literals

The body of a character literal can include any sequence of characters allowed by the system's CCSID (including non-syntactic characters).

Identifiers

You can specify user-defined identifiers for all types of ADL statements. These identifiers assign a name to a statement so that it can be referenced by other statements. Identifiers consist of a sequence of uppercase or lowercase letters, numbers, and the special characters question mark (?), percent (%), ampersand (&), and underscore (_).

Each identifier must be unique in the statement in which you use it, but you can use the same identifier in different contexts in other statements.

Contexts in which you can use an identifier include:

- Within a module, the identifiers of all DECLARE statements must be unique.
- Within a module, the identifiers of all PLAN statements must be unique.

- Within a declaration, the identifiers of all constants and subtypes, and the fully-qualified identifiers of all data declarations must be unique.
- Within a sequence or case declaration, the identifiers of all data declarations must be unique.
- Within a plan statement, the identifiers of all statements must be unique.
- Within a declare statement, no fully-qualified identifier of a data declaration can be the same as the partly-qualified identifier of another data declaration.

Some ADL statements support references to constants, subtypes, or data declarations. These references are specified in terms of the identifiers associated with the referenced entity. Whenever there is a possibility of ambiguity in a reference, you must specify it with enough high level identifiers as qualifiers to eliminate the ambiguity. The qualifiers are specified in the same sequence as the nested contexts and are separated by periods (.).

For example, given the hierarchy:

```
A
  B
    C
    D
  E
    C
    F
```

If an ADL statement refers to A.B.C, then at least B.C must be specified to avoid any ambiguity with E.C.

INCLUDE Statement

The INCLUDE statement is a special type of ADL statement.

An ADL INCLUDE statement can be specified anywhere an ADL statement is allowed. The INCLUDE statement names a file containing ADL text. The included ADL text must be valid in the context of the type of statement in which it is specified. For example, CONSTANT statements cannot be included in a plan statement.

The included ADL text can itself contain INCLUDE statements, but there is a limit to the number of loops of INCLUDE statements that can be specified.

Data Types

There are many different methods used to encode numeric, character and other types of data. Each instance of a data type is represented by an encoded string of bits. It is the programming language and system environment that determines the method of encoding data in a bit string. ADL uses a set of attributes to define the data types that correspond to the methods of encoding data used by various programming languages. Data attributes are described in detail in Chapter 5, “Attributes” on page 103. The data types are described in the following topics. Not all ADL data types incorporate all of the attributes described below. Likewise, not all programming languages require the use of all the data types.

Field Data Types

ADL defines the following data types for describing how *field* data is encoded:

ASIS	A string of bits whose encoding is unknown or inconvertible by an ADL module.
BINARY	A fixed-length string of bits that encodes a fixed precision number as a base 2 integer. Among the attributes that can be specified are the precision, scaling factor, and radix of the encoded number, and whether its encoding is byte reversed.
BIT	A string of bits whose encoding is application defined.
BITPRE	A variable-length string of bits whose encoding is application defined. The actual length of the string is specified by a length field that precedes the bit string.
BOOLEAN	A bit string that encodes either TRUE or FALSE.
CHAR	A string of bits that encodes characters.
CHARPRE	A variable-length string of bits that encodes characters. The actual length of the string is specified by a prefix that precedes the string of bits.
CHARSFX	A variable-length string of bits that encodes characters. The actual length of the string is determined by scanning for a suffix string of bits used as a terminator.
ENUMERATION	The binary encoding of one of a set of integers, where each integer is associated with an identifier as a constant.
FLOAT	A fixed-length string of bits that encodes an approximate number by means of a characteristic and a significand. The FORM attribute specifies the format of the encoded string, and implies its length.
PACKED	A fixed-length string of bits that encodes a number by means of a sequence of hexadecimal representations of decimal digits. Among the attributes that can be specified are the precision and scaling of the field.
ZONED	A fixed-length string of bits that encodes a number by means of a sequence of hexadecimal representations of decimal digits and zones. Among the attributes that can be specified are the precision and scaling of the field.

ADL Constructor Data Types

ADL defines the following *constructor* data types that specify how data elements are combined to form more complex entities:

SEQUENCE A collection of fields and constructors, each encoded according to its ADL data type.

The elements of a SEQUENCE can be of any ADL field data type or constructor data type. Also, the SKIP statement can be used to skip bits between the elements of a SEQUENCE to account for the data alignments methods of various programming languages, or to account for filler fields allowed by some programming languages.

ARRAY A collection whose elements are uniquely associated with one or more integer values, called the dimensions of the array.

Each dimension of an array is declared in terms of either the lowest and the highest integer values, or the lowest integer value and the number of elements in the dimension. These values can be specified as numeric literals or by references to other variables. In the latter case, the overall size of the array can vary for each instance of the array, or the array can contain slots for all possible elements with only elements within the specified bounds containing data.

The elements of an array must all be the same length, but they need not be of the same data type. If an ARRAY of type CASE is declared, all elements of the array can be of the same data type, but vary from instance to instance of the array. Alternatively, each element can be of a different data type if each element is a SEQUENCE containing its own discriminant field.

CASE A set of alternative data declarations, one of which determines the interpretation of a string of bits, depending on the value of one or more discriminant fields.

A CASE contains an ordered set of WHEN statements and an optional OTHERWISE statement. Each WHEN statement consists of a condition to be tested and either a data declaration or a SKIP statement. The condition to be tested is specified by predicates connected by the logical operators AND, OR, and NOT.

A WHEN statement or an OTHERWISE statement consists of a data declaration, a SKIP statement, or a REJECT statement:

- The data declaration of an OTHERWISE statement can be of any ADL field or constructor type.
- The SKIP statement can be specified to indicate that only filler data exists.
- The REJECT statement can be specified to reject the input data and terminate the conversion plan.

If an OTHERWISE statement is specified for a CASE, it is selected if no condition of a WHEN statement evaluates to TRUE. If no OTHERWISE statement is specified in a CASE, and no WHEN statement condition evaluates to TRUE, then no instance of the CASE exists.

Data Type Attributes

The attributes of ADL data types specify how data is actually encoded within a representation domain. That is, the encoding of a data item is described by specifying an ADL data type and selected values of its attributes. For example, the byte reversal attribute (BYTRVS) of a BINARY field specifies how the bytes of the binary encoding are ordered.

You can specify values for all of the attributes of a data type when declaring a field of that type, but this is both tedious and error prone. Therefore, for each data type, ADL specifies:

- Attributes that assume a default value if a value is not specified. ADL defines default values for each of these attributes. However, you can change the defaults for all uses of an attribute in a declaration by using the DEFAULT statement. This makes it possible for one programmer to specify the required set of defaults for a given representation domain and for other programmers to adopt those defaults by using the INCLUDE statement.
- Attributes that are ignored if a value is not specified.

The following rules define the relationships between system-defined attributes, user-defined attributes, and the attributes specified in a data declaration statement. For each data type and each attribute of this data type, the following sequence of priorities applies:

1. The attribute is defined in the data declaration statement.
2. The data type is a subtype instance and the attribute is defined in the SUBTYPE declaration.
3. The data type is a subtype instance of another subtype. In this case, step through the chain of subtypes and take the first occurrence of the attribute.
4. There is a DEFAULT statement for this data type and attribute.
5. There is an ADL-defined value for this data type and attribute.

If none of these conditions apply, the attribute is undefined.

If two mutually exclusive attributes for a data type are specified at different levels of priority, then the attribute at the higher level is used.

Declarations

A declaration describes data in terms of the data types and encoding methods of a single representation domain. A declaration consists of a DECLARE statement that contains:

- DEFAULT statements that set the default attributes of ADL types.
- CONSTANT statements that associate an identifier with an ADL literal. The identifier associated with a CONSTANT statement can be referenced by SUBTYPE statements and data declaration statements in the declaration section.
- SUBTYPE statements that declare subtypes of ADL types. The identifier associated with a SUBTYPE statement can be referenced by all SUBTYPE statements and data declaration statements in the declaration.
- Data declaration statements that describe application data. The identifier associated with a data declaration statement can be referenced by other statements in declarations and plans.

DECLARE Statements

Each declaration of a module is specified by using a DECLARE statement, such as:

```
ProgramA: DECLARE
          BEGIN;
            DEFAULT statements
            SUBTYPE statements
            CONSTANT statements
            Data declaration statement
          END;
```

where ProgramA is a name assigned to the declaration and declaration statements are specified between BEGIN and END statements.

DEFAULT Statements

A value can be specified for a defaultable attribute of an ADL data type by using a DEFAULT statement, such as:

```
DEFAULT BINARY
        BYTRVS(FALSE)
        RADIX(10)
       SIGNED(FALSE);
```

which specifies default values for the BYTRVS, RADIX, and SIGNED attributes of the ADL BINARY data type.

Each ADL data type whose attributes are to be defaulted must have a separate DEFAULT statement.

The defaults established by a DEFAULT statement for an attribute apply to the entire declaration.

Only one DEFAULT statement can be specified for an ADL type. If a DEFAULT statement is not specified for an ADL type, then the default ADL attributes for that type are used.

If a data declaration specifies attributes that are mutually exclusive with the attributes specified in their respective DEFAULT statement, the attributes on the data declaration statement take precedence over the default ones.

CONSTANT Statements

A literal is a specification of a data value in ADL text. ADL defines how literals are to be specified for numeric, character, boolean, and other ADL data types. Literals are used to specify the values of attributes, in assignment statements, and in predicates. However, it is often desirable to associate an identifier with a literal and then refer to the identifier whenever the value of the literal is required. This is accomplished by means of a CONSTANT statement, such as:

```
base: CONSTANT 10;
population: BINARY RADIX(base)...;
```

where `base` is associated with the literal value 10, and the RADIX attribute is set to the value 10 by referring to the identifier `base`.

SUBTYPE Statements

The SUBTYPE statement can be used to declare a subtype of an ADL type or of another subtype. A data item can then be declared to be an instance of the subtype.

Subtypes of both field and constructor types can be defined. Field subtypes allow the subtype identifier to be associated with a specific ADL type and an associated set of attributes. For example, in:

```
partno: SUBTYPE OF PACKED
        PRECISION(9) SIGNED(FALSE) TITLE('Part Number');
```

some other identifier can be declared to be an instance of the `partno` subtype, as in:

```
assembly: partno TITLE('Part Number of the whole assembly.');
```

In this case, `assembly` is declared to be an instance of the `partno` subtype and thereby of the ADL PACKED type with the `PRECISION(9)` attribute. However, `assembly` specifies its own TITLE attribute, overriding the TITLE attribute specified for the `partno` subtype.

Subtypes of constructors are similar, but their instances also inherit the components of the constructor subtype. In the following example, `name` is the identifier of a subtype of the ADL SEQUENCE constructor type. This subtype declares `name` to have the components `last`, `first`, and `initial`. `namerec` is a SEQUENCE consisting of a set of `names`, each of which includes `last`, `first`, and `initial` components.

```
name: SUBTYPE OF SEQUENCE TITLE('General Name Structure')
                 BEGIN;
                    last: CHAR LENGTH(12);
                    first: CHAR LENGTH(12);
                    initial: CHAR;
                 END;
namerec: SEQUENCE TITLE('Employee Contacts Record')
         BEGIN;
            employee: name;
            spouse: name;
            broker: name;
            attorney: name;
            parole_officer: name;
            clergy: name;
         END;
```

Two kinds of subtypes can be defined. One kind of subtype is related to the data types of other programming languages. Subtypes can be assigned identifiers similar to the data type names of a programming language, with attributes specified as required by the representation domain. For example, a subtype named `short` could be defined by a C representation domain to match the way a C implementation represents instances of the C `short` data type. A programmer specifying ADL declarations for that representation domain can then declare data to be of the `short` subtype, as in:

```
short: SUBTYPE OF BINARY
                  BYTRVS(TRUE)
                  PRECISION(15);
counter: short;
```

All declared instances of the `short` subtype are represented in 16 bits and encoded with byte reversal. `counter` is declared to be an instance of the subtype `short`, and therefore of the ADL BINARY type.

Application specific subtypes can also be defined. For example, if `name` is declared as a subtype, then a personnel record could declare `employee_name`, `spouse_name`, and `child_name` to all be instances of the `name` subtype. If the same application subtypes are required in several declare statements, they can be specified in an ADL text file that is included in those declaration statements.

Data Declaration Statements

The application data exported and imported by programs is described by data declaration statements, such as:

```
retail_price: ZONED PRECISION(5) SCALE(2);
```

where `retail_price` is the identifier associated with all instances of the data, `ZONED` specifies the data is of the ADL ZONED data type, and `PRECISION` and `SCALE` are attributes of the instances.

Plans

A plan statement specifies a program for converting data from one form to another, where both forms are described by data declaration statements. These programs can be invoked whenever data becomes available for the input parameters of the plan and converted data is required.

PLAN Statements

Each plan of a module is specified by a PLAN statement, for example:

```
ReadPlan: PLAN (A: INPUT, B: OUTPUT)
          BEGIN;
            assignment statements
            CALL statements
          END;
```

where `ReadPlan` is a name assigned to the plan, `A` and `B` are the INPUT and OUTPUT parameters of the plan, and the assignment and CALL statements of the plan are specified between `BEGIN` and `END` statements.

The names specified in the parameter list of the plan are references to data declarations. A plan assumes that these declarations describe the data areas passed to the plan as parameters when the plan is called. The caller of the plan passes a data area containing original data and a data area to receive converted data. The assignment statements of the plan move and convert data from the fields of the original data area to the converted data area.

The LENGTH and CCSID attributes can be specified as attributes of INPUT parameters, and the MAXLEN and CCSID attributes can be specified as attributes of OUTPUT parameters. These attributes provide additional information about the parameters that can be used within the plan. For example, the MAXLEN attribute of an OUTPUT attribute specifies the maximum length of the data that can be returned by the plan. These attributes can be specified as literals or as references to other parameters. This allows their values to be specified by the caller of the plan.

Assignment Statements

The assignment statements of a plan cause data to be moved from a *source* variable, typically an INPUT parameter, to a *target* variable, typically an OUTPUT parameter. When the source and target are both fields, the data types of the source and target, along with their accompanying attributes, are compared. It may be that the two data types are compatible and that no conversion is required. If necessary, however, the data of the source field must be converted to the type and attributes of the target field.

For fields, DD&C supports the following conversions:

- From source to target fields of the same type, DD&C performs whatever conversions are required by the attributes of the source and target.
- Any numeric type can be converted to any other numeric type.
- ENUMERATION data types can be converted to or from BINARY.

- Any character type can be converted to any other character type if the character data representation architecture (CDRA) has defined conversions from the source CCSID to the target CCSID.
- Any data type can be converted to or from the ASIS data type. In these cases, the source and target fields are treated as bit strings and moved, with padding or truncation. The results may or may not be valid data for the target field.

For constructors, DD&C supports the following conversions:

- When the source and target are both ARRAYs, the dimension list attributes of the source and target are compared to determine if they are compatible. Both arrays must have the same number of dimensions, and both arrays must have the same number of elements in each dimension. Conversions are performed as each source element is assigned to a target element.
- When the source and target are both SEQUENCEs, the identifier of each element of the target is used to locate a matching identifier of the source. Conversions are performed as the data of the source element is assigned to the target element. The target elements need not be in the same order as the source elements, and not all target elements need be selected. Because of this, the target declaration can both select and reorder source elements.
- When the source and target are both CASEs, the condition clauses of the WHEN statements of the source are evaluated to determine which data declaration applies to the source data. The identifier of the selected WHEN statement is then used to locate a WHEN statement of the target CASE with a matching identifier, and its data declaration applies to the target data. Conversions are performed as the source data is assigned to the target. If no source WHEN statement is selected, then the OTHERWISE statement of both the source and target are selected.

If a target data declaration includes a WHEN clause, then the condition specified must be TRUE or an exception occurs, thereby causing the plan to be terminated. This allows the plan to act as a filter on the input data passed to it. For example, if the source data is being read from a file, then only data that passes through the filter is converted and passed to the caller of the plan. Thus, only a subset of the file's data is actually presented to the reader of the file. And if the target is a file to which data is being written, then only data that passes through the filter is written to the file.

CALL Statements

A CALL statement calls a named program external to the program containing the conversion plan and passes it a list of parameters. The called program can operate on the input parameters passed to it and return values in the output parameters for further use by the conversion plan.

While CALL statements can be used for any application purpose, the anticipated use of CALL statements is to convert data that cannot otherwise be described or converted by ADL.

Workspace Variables and Storage Allocation

Associated with each data declaration statement of an ADL module is a block of storage onto which the declared variable is mapped. If the declared variable is specified in the parameter list of a PLAN statement, then the corresponding block of storage has been allocated by the caller of the plan, either as input to the plan or as an area for output from the plan. This kind of variable is called a *parameter variable*. Otherwise, the block is contained in a workspace allocated by the module. This kind of variable is called a *workspace variable*.

When subsequently initializing the conversion plan, the conversion executor creates a module workspace. This module workspace remains in existence until execution of the conversion plan is complete. This means that workspace variables persist over several calls to convert the same or different conversion plans of a conversion plan space. Workspace variables can therefore be seen as global variables for each conversion with a conversion plan. You should therefore always ensure that workspace variables are correctly initialized before they are used.

Since there is only one persisting module workspace, all references by plans or declarations to workspace variables are to the same workspace blocks. This allows workspace variables to be used for communications between invocations of different plans or between different invocations of the same plan.

Exception Handling

A variety of conditions can cause an exception to occur when running a conversion plan. All exceptions result in the conversion plan being terminated and the error code being returned to it's caller. ADL defines an *ADL communications area (FMTADLCA)* that contains this diagnostic information.

Chapter 2. Common Elements

This chapter describes the syntax and semantics of elements that are common to many ADL statements. The elements are described in alphabetical order.

Constants

The following shows the function, syntax, rules, and examples:

Function

Specify the constants used in the DD&C implementation of ADL.

Syntax

```
<max7>  ::= 127

<max8>  ::= 255

<max15> ::= 32,767

<max28> ::= 268,435,455

<max31> ::= 2,147,483,647

<min7>  ::= -128

<min31> ::= -2,147,483,648
```

Syntax rules

None.

General rules

None.

Examples

None.

<character>

The following shows the function, syntax, rules, and examples:

Function

Specify character units.

Syntax

```
 <character> ::=
   <digit> |
   <letter> |
   <special character> |
   <underscore> |
   <space>

 <digit> ::=
   0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

 <letter> ::=
   <upper case letter> | <lower case letter>

 <upper case letter> ::=
   A | B | C | D | E | F | G | H | I |
   J | K | L | M | N | O | P | Q | R |
   S | T | U | V | W | X | Y | Z

 <lower case letter> ::=
   a | b | c | d | e | f | g | h | i |
   j | k | l | m | n | o | p | q | r |
   s | t | u | v | w | x | y | z

 <special character> ::=
   . | , | : | ; | ? | ( | ) | ' | " |
   / | - | & | + | % | = | * | > | <

 <underscore> ::= _

 <source character> ::=
```

Syntax rule

A <source character> is any character allowed by the <CCSID attribute> of the ADL source file.

General rules

None.

Examples

None.

<comparison predicate>

The following shows the function, syntax, rules, and examples:

Function

Specify a comparison of two values.

Syntax

```
<comparison predicate> ::=
  <value expression>
  <comparison operator>
  <value expression>

<comparison operator> ::=
  = | < | > | <> | <= | >=
```

Syntax rules

The data types of the first <value expression> and the second <value expression> are comparable. The <value expression>s are comparable if the result of the second <value expression> can be converted to the data types of the result of the first <value expression> as defined by the matrix in Figure 6 on page 136.

General rules

1. Let x denote the value of the first <value expression> and let y denote the value of the second <value expression>.

   "x <comparison operator> y" is either TRUE or FALSE:

   "x = y" is TRUE if and only if x and y are equal.
   "x < y" is TRUE if and only if x is less than y.
   "x > y" is TRUE if and only if x is greater than y.
   "x <> y" is TRUE if and only if x and y are not equal.
   "x <= y" is TRUE if and only if x is not greater than y.
   "x >= y" is TRUE if and only if x is not less than y.

2. It is only possible to compare ADL data types that belong to the same class.

   In DD&C, four classes of ADL data types are defined. ADL data types not belonging to any of the following classes cannot be used for comparison purposes:

   a. Character data types.

      This class consists of the ADL data types CHAR, CHARPRE, CHARSFX, <character literal>, and <encoded hex literal>. For this class, only the <comparison operator>s = and <> are allowed.

      Comparisons are performed as follows:

      1) For CHARPRE, only the character specified by its prefix length field is used. The prefix does not participate in the comparison.

      2) For CHARSFX, the character preceding the suffix is used. The suffix does not participate in the comparison.

3) If the lengths of the two <value expression>s are not equal, then the shorter <value expression> is considered to be extended on the right with space characters defined by the <CCSID attribute> of the shorter <value expression>.

4) The comparison is performed from left to right, comparing the individual characters of the <value expression>s. If all such individual comparisons are equal, then the two <value expression>s are equal; otherwise, the result of the comparison of the <value expression>s is defined to be unequal.

b. Numeric data types.

This class consists of the ADL data types BINARY, ENUMERATION, PACKED, ZONED, <positive integer>, and <signed integer>.

The attributes of these data types must have the value COMPLEX(FALSE).

Only numeric values that can be expressed as a 32-bit signed integer are allowed.

Comparison of numeric values is always exact, without truncation or rounding. If an operand has SCALE not equal to zero, comparison can lead to an overflow situation under certain circumstances.

c. Bit fields.

This class consists of the ADL data types ASIS, BIT, BITPRE, <bit literal>, and <hex literal>. The comparison rules are:

1) If the lengths of the two <value expression>s are not equal, the shorter <value expression> is considered to be extended on the right with `B'0'` bits.

2) The comparison is performed from left to right, comparing the individual bits of the <value expression>. If all such individual comparisons are equal, then the two <value expression>s are equal; otherwise, the result of the comparison of the <value expression>s is defined to be the result of comparing the first unequal bit encountered.

3) A bit value of `B'1'` is greater than a bit value of `B'0'`.

d. Boolean fields

This class consists of the ADL data types <BOOLEAN> and <boolean literal>.

A truth value of TRUE is greater than a truth value of FALSE.

Examples

1. empno = 123456
2. salary > 10000
3. legalspeed <= 55
4. name <> 'Elaine'

<condition>

The following shows the function, syntax, rules, and examples:

Function

Specify a condition that is TRUE or FALSE depending on the result of applying boolean operators to specified conditions.

Syntax

```
<condition> ::=
  <boolean term> |
  <condition> OR <boolean term>

<boolean term> ::=
  <boolean factor> |
  <boolean term> AND <boolean factor>

<boolean factor> ::=
  [NOT]<boolean primary>

<boolean primary> ::=
  <boolean literal> |
  <qualified identifier> |
  <predicate> |
  (<condition>)
```

Syntax rule

If a <qualified identifier> is specified for the <boolean primary>, the <qualified identifier> refers to a BOOLEAN field.

General rules

1. The result is derived by the application of the specified boolean operators (AND, OR, NOT) to the conditions that result from the application of each specified <predicate> to a given field. If boolean operators are not specified, then the result of the <condition> is the result of the specified <predicate>.
2. NOT(TRUE) is FALSE, and NOT(FALSE) is TRUE.
3. AND and OR operators are defined in the following truth table:

file

Values		**Results**	
A	**B**	**A and B**	**A or B**
TRUE	TRUE	TRUE	TRUE
TRUE	FALSE	FALSE	TRUE
FALSE	TRUE	FALSE	TRUE
FALSE	FALSE	FALSE	FALSE

4. Expressions within parentheses are evaluated first and when the order of evaluation is not specified by parentheses, <predicate>s are evaluated first, NOT applied before AND, AND applied before OR, and operators at the same precedence level applied from left to right.

Examples

None.

<identifier>

The following shows the function, syntax, rules, and examples:

Function

Specify an identifier for an entity.

Syntax

```
<identifier> ::=
  <identifier string>
  | " <identifier string> "

<identifier string> ::=
  {<letter> | <digit> | <identifier character>}...

<identifier character> ::=
  ? | % | & | <underscore>
```

Syntax rules

1. An <identifier> consists of a maximum of <max8> characters.
2. If an <identifier string> matches an ADL key word, then it must be enclosed within double quotation marks (").
3. The same <identifier> that is used to identify a <data declaration statement> cannot be used to define a <CONSTANT> data type.
4. At least one character of an <identifier> must be a <letter> or an <identifier character>.
5. The case of a <letter> is significant when specified in an identifier. For example, abc and aBc are different <identifier>s.

General rules

None.

Examples

1. The following are valid application-defined identifiers:

```
empser     EmpSer     go123    “BINARY”    123GO
_empser    empser_    emp%ser  &123GO    emp_ser
```

2. The following are not valid application-defined identifiers:

```
empser#    #EmpSer    1234    EMPLOYEE-SERIAL-NUMBER
```

<literal>

The following shows the function, syntax, rules, and examples:

Function

Specify a string of characters representing a value.

Syntax

```
<literal> ::=
  <character literal> |
  <noncharacter literal>

<character literal> ::=
  <character string>[<separator><character string>]...
  [<CCSID attribute>]

<character string> ::=
  '[<source character> | <quote representation>]...'

<quote representation> ::= ''

<noncharacter literal> ::=
  <bit literal> |
  <signed integer> |
  <positive integer> |
  <hex literal> |
  <boolean literal> |
  <encoded hex literal>

<bit literal> ::=
  {b | B}<bit string> [<separator><bit string>]...

<bit string> ::= '[0 | 1]...'

<signed integer> ::= [+ | -] <digit>...

<positive integer> ::= [+] <digit>...

<hex literal> ::=
  {x | X}<hex string> [<separator><hex string>]...

<hex string> ::= '[<hex digit>]...'

<encoded hex literal> ::= <hex literal> <CCSID attribute>

<hex digit> ::= <digit> | A | B | C | D | E | F

<boolean literal> ::= FALSE | TRUE
```

Syntax rules

1. If the <CCSID attribute> is specified for a <character literal>, it must match the CCSID of the source file.
2. An even number of <hex digit>s must be specified for an <encoded hex literal>.

General rules

1. For <bit literal>s,
   a. The data type is BIT with ADL defined default attributes.
   b. The length is the number of bits that the <bit literal> contains. The maximum length of a <bit literal> in DD&C is 32760 bits.
   c. The value is the sequence of bits that the <bit literal> contains.
   d. If a <bit literal> is specified as a sequence of <bit string>s, the <bit string>s are concatenated to form a single <bit string> without any intervening apostrophes (').
2. For <boolean literal>s,
   a. The data type is BOOLEAN with ADL-defined default attributes.
   b. The value is TRUE or FALSE.
3. For <character literal>s,
   a. The data type is CHAR with ADL-defined default attributes.
   b. The length is the number of bytes in the string. Each <quote representation> in a <character literal> represents a single quotation mark character in both the value and the length of the <character literal>. The maximum length of a <character literal> in DD&C is 32760 single-byte characters.
   c. The value is the sequence of characters the <character literal> contains.
   d. If a <character literal> is specified as a sequence of <character string>s, the <character string>s are concatenated to form a single <character string> without any intervening apostrophes (').
   e. A <character literal> cannot contain a <newline> token.
   f. If no CCSID is specified with the <character literal>, the system CCSID is taken as the CCSID value of the <character literal>. The system CCSID is the CCSID derived from the code page used by the OS/2 process in which the Parse function of DD&C (FMTPRS) is running.
   g. Any CCSID value specified must be a constant greater than 0.
4. For <encoded hex literal>s,
   a. The data type is CHAR with ADL-defined default attributes.
   b. The length is the number of bytes in the string. The maximum length of an <encoded hex literal> in DD&C is 32760 single-byte characters.
   c. The value is the sequence of characters represented by the hexadecimal digits the <encoded hex literal> contains, in the CCSID specified.
   d. If an <encoded hex literal> is specified as a sequence of <hex string>s, the <hex string>s are concatenated to form a single <hex string> without any intervening apostrophes (').
   e. Any CCSID value specified must be a constant greater than 0.

5. For <hex literal>s,

   a. The data type is ASIS with ADL-defined default attributes.

   b. The length is the number of bits represented by the hexadecimal digits the <hex literal> contains. The maximum length of a <hex literal> in DD&C is 32760 nibbles.

   c. The value is the sequence of hexadecimal digits that the <hex literal> contains.

   d. If a <hex literal> is specified as a sequence of <hex string>s, the <hex string>s are concatenated to form a single <hex string> without any intervening apostrophes (').

6. For <positive integer>s and <signed integer>s:

   a. The data type is BINARY with ADL-defined default attributes. Use BYTRVS (TRUE) for OS/2, and BYTRVS(FALSE) for AIX.

   b. The value must be in the range between <min31> and <max31>.

   c. A positive sign is assumed if a plus (+) is not specified.

   d. The numeric value is derived from the normal mathematical interpretation of signed positional decimal notation.

Examples

1. Examples of valid literals are:

```
55      'abcd'     B'0110011001'     'abcd' CCSID(500)
TRUE    'it''s'    x'A98B'           x'21' CCSID(437)
```

2. The following is a single character literal broken into segments:

```
'Whose woods these are, I think I know '
'His house is in the village, though. '
'He will not see me stopping here '
'To watch his woods fill up with snow. '
```

<module>

The following shows the function, syntax, rules, and examples:

Function

Specify an ADL module.

Syntax

```
<module> ::= <Parse unit>...
```

The term <module> as used in DD&C is not part of the formal ADL language specification. A module is a collection of one or more <parse unit>s.

Syntax rule

At least one <DECLARE statement> and one <PLAN statement> must be specified in a <module>.

General rules

1. Two classes of data storage are associated with a <module>:
   a. Data storage passed as arguments to the parameters of a plan, called parameter variables.
   b. Data storage allocated and managed by the <module>, called workspace variables.
2. Parameter variables are allocated and managed by the caller of the plan. Workspace variables are allocated as required by the variables declared in the <DECLARE statement>s of the <module>. For every such variable that is not referenced as a <parameter> of a plan, a single area of storage is allocated for each processing thread that invokes the PLANs of the <module>. All invocations of any of the plans of the <module> by the same thread must be provided with access to this area.

Examples

See Appendix A, "Scenarios" on page 175.

<parse unit>

The following shows the function, syntax, rules, and examples:

Function

Specify an ADL parse unit. A parse unit is defined as the amount of ADL text that is parsed with one call of the parse function of DD&C. It consists of declarations, plans or both.

Syntax

```
<parse unit> ::= {<DECLARE statement> | <PLAN statement>}...
```

Syntax rule

At least one <DECLARE statement> or one <PLAN statement> must be specified in a <parse unit>.

General Rules

None.

Example

Each ADL source file, together with all included ADL files, is a <parse unit>.

<positional identifier>

The following shows the function, syntax, rules, and examples:

Function

Specify an entity in terms of its position within its containing entity.

Syntax

```
<positional identifier> ::= " <digit>... "
```

Syntax rules

None.

General rules

1. A <positional identifier> is implicitly associated with all <DECLARE statement>s and with all entities contained within a <DECLARE statement>, <SEQUENCE>, or <CASE>, starting with 1 for each, and increasing by 1 for each entity it contains up to <max31>. A <positional identifier> is not assigned to a <SKIP statement>, however.
2. The position of a <DECLARE statement> depends on the order in which ADLDCLSPC objects are specified in the parameter list of the conversion plan builder component. The first <DECLARE statement> of the first ADLDCLSPC in the list is assigned the position 1. If the first ADLDCLSPC object contains n <DECLARE statement>s, therefore, the first <DECLARE statement> of the second ADLDCLSPC object has the position n+1.
3. A <positional identifier> can only be referred to from within a <PLAN statement>.

Examples

1. The <positional identifier> of the first entity in a DECLARE statement is “1”.
2. The <positional identifier> of the second entity in a module is “2”.

<predicate>

The following shows the function, syntax, rules, and examples:

Function

Specify a condition that can be evaluated to give a value of TRUE or FALSE.

Syntax

```
<predicate> ::=
  <comparison predicate>
```

Syntax rules

None.

General rules

None.

Examples

None.

<qualified identifier>

The following shows the function, syntax, rules, and examples:

Function

Specify the qualified identifier of an entity.

Syntax

```
<qualified identifier> ::=
  <qualifier list>

<qualifier list> ::=
  [<qualifier>.]...<qualifier>

<qualifier> ::=
  <identifier> | <positional identifier>
```

Syntax rules

1. <qualifier>s must be specified in the order defined by the "containing" relationship of entities.
2. The highest level of <qualifier> of a <qualified identifier> is the <identifier> of a <DECLARE statement>.
3. A partially-qualified name with high-level or intermediate-level <qualifier>s missing can be specified if it is unambiguous within its containing module.
4. If a <qualified identifier> consists of <positional identifier>s only, then it must be fully qualified.
5. A <positional identifier> can only be used in PLAN statements, not in DECLARE statements.

General rules

1. A reference to a <qualified identifier> refers to the value associated with the <qualified identifier>.
2. An identifier can only be used to construct a <qualified identifier> if it can itself be referenced.
3. If a <qualified identifier> begins with the name of a <DECLARE statement>, only this <DECLARE statement> is searched for the <qualified identifier>.
4. The <qualified identifier> of a data declaration cannot be the same as the fully-qualified identifier of another data declaration. Therefore, the following declaration:

```
d: DECLARE BEGIN;
   SEQUENCE BEGIN;
      a: BINARY;
      d: SEQUENCE BEGIN;
         a:FLOAT;
         END;
   END;
END;
```

is not valid, since d.a can be either a fully-qualified identifier or part of the fully-qualified identifier d.d.a.

5. A reference to a data declaration inside an array declaration is only allowed from within the array. The reference always applies to the current element of the array. For example:

   ```
   CHAR CCSID( b );
   ARRAY OF
      b: BINARY;
   ```

   is not allowed.

   In contrast:

   ```
   ARRAY OF SEQUENCE
      BEGIN;
         b: BINARY;
         CHAR CCSID( b );
      END;
   ```

   is allowed and each character field of the array can have a different CCSID.

Examples

1. An <identifier> must be specified with enough high level <qualifier>s to eliminate any ambiguity. See “Identifiers” on page 7 for an example.

2. The following are examples of <qualified identifier>s:

   ```
   PersonnelRecord.name
   PersonnelRecord.children.dateOfBirth
   SoftballRecord.Team.Wins
   ```

3. Inventory and Desk are declared as follows:

   ```
   Inventory: SUBTYPE OF
              SEQUENCE BEGIN;
                ItemNumber: BINARY;
                WoodType: CHAR LENGTH(20);
              END;
   Furniture: SEQUENCE BEGIN;
                Desk: Inventory;
              END;
   ```

   The qualified names for the fields in Furniture are:

   ```
   Furniture.Desk.ItemNumber
   Furniture.Desk.WoodType
   ```

4. The following construction is not valid:

   ```
   a: b: SEQUENCE BEGIN;
      c: BINARY;
      d: CHAR CCSID(c);
      END;
   ```

   because the reference to the data type for the CCSID value is not unique—it could be a reference to “a.c” or “b.c”.

   The following construction is not valid for the same reason:

   ```
   sub:  SUBTYPE OF
      SEQUENCE BEGIN;
         c: BINARY;
         d: CHAR CCSID( c );
         END;
   a: b: sub;
   ```

<token>

The following shows the function, syntax, rules, and examples:

Function

Specify lexical units.

Syntax

```
<token> ::=
  <nondelimiter token> | <delimiter token>

<nondelimiter token> ::=
  <qualified identifier> |
  <key word> |
  <noncharacter literal>

<delimiter token> ::=
  <character literal> |
  <special symbol>

<special symbol> ::=
  ' | * | . | , | : | ; | = | + | - | ( | ) |
  > | < | <> | >= | <= | /* | */ | <arrow>

<separator> ::=
  {<comment> | <space> | <new line> | <tab>}...

<comment> ::=
  /* <source character>... */

<space> ::=
  See “Syntax rules” on page 33.

<new line> ::=
  See “Syntax rules” on page 33.

<tab> ::=
  See “Syntax rules” on page 33.
```

```
<key word> ::=
  ALGEBRAIC | AND | ARRAY | ASIS | BEGIN
   | BINARY | BIT | BITPRE | BLNENC
   | BOOLEAN | BYTRVS | CALL | CASE | CCSID
   | CHAR | CHARPRE | CHARSFX | COMPLEX | CONSTANT
   | CONSTRAINED | DECLARE | DEFAULT | DGTLSTBYT | DMNHIGH
   | DMNLOW | DMNLST | DMNMAX | DMNSIZE | END
   | ENUMERATION | ESCAPE | EXACT | FALSE | FB32
   | FB64 | FB80 | FH32 | FH64 | FH128
   | FI128 | FIT | FLOAT | FORM | FRSBYT
   | HELP | HIGH | INCLUDE | INPUT
   | JUSTIFY | LEFT | LENGTH | LOGICAL
   | LOW | LSTBIT | LSTBYT | MAXALC | MAXLEN
   | NOT | NOTE | OF | OR | OTHERWISE
   | OUTPUT | PACKED | PLAN | PREBYTRVS | PRECISION
   | PRELEN | PRESIGNED | RADIX | REJECT | RIGHT
   | ROUND | SCALE | SEQUENCE | SGNLOC | SGNMNS
   | SGNPLS | SGNUNS | SIGNED | SKIP | SUBSTR
   | SUBTYPE | THEN | TITLE | TRUE | TRUNCATE
   | UNITLEN | WHEN | ZONED | ZONENC | ZONFRSBYT
   | ZONLSTBYT

<arrow> ::=
  <-

<terminator> ::= ;
```

Syntax rules

1. The encoding of all <token>s (including <space>, <new line>, and <tab>) is determined by the <CCSID attribute> of the ADL text. The <CCSID attribute> is specified outside of the ADL text.
2. A <token>, other than a <character literal>, cannot include a <space>.
3. Any <token> can be followed by a <separator>.
4. A <nondelimiter token> can be followed by either a <delimiter token> or a <separator>. If the syntax does not allow a <nondelimiter token> to be followed by a <delimiter token>, then that <nondelimiter token> shall be followed by a <separator>.
5. The character string “/*” is not allowed within a <comment>. “*/” is only allowed as the terminator of a <comment>.

General rules

All lines of input are considered to be a continuous string.

Examples

None.

<value expression>

The following shows the function, syntax, rules, and examples:

Function

Specify an expression that can be evaluated to return a value.

Syntax

```
<value expression> ::=
  <value specification>

<value specification> ::=
  <literal> |
  <constant identifier> |
  <qualified identifier> |
  <LENGTH function>
```

Syntax rules

If a <qualified identifier> is specified, it must refer to a <field>.

General rules

1. Expressions within parentheses are evaluated first.

Examples

None.

<when clause>

The following shows the function, syntax, rules, and examples:

Function

Specify a condition that must be TRUE.

Syntax

```
<WHEN clause> ::=
  WHEN <condition>
```

Syntax rules

None.

General rules

None.

Examples

None.

Chapter 3. Statements

This chapter describes, in alphabetical order, the statements of ADL.

<assignment statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the assignment of a <value expression> to a variable. The result of evaluating the <value expression> is called the *source* of the assignment, and the variable is called the *target* of the assignment.

Syntax

```
<assignment statement> ::=
  [<identifier>:]
  <qualified identifier>
  <arrow>
  <value expression>
  <terminator>
```

Syntax rule

The <qualified identifier> at the left of the <arrow> must not be that of an <INPUT parameter>.

General rules

1. If an <identifier> is specified, it cannot be referenced.
2. If the data type and attributes of the source do not match the data type and attributes of the target, the value of the source is converted to the data type and attributes of the target.
3. Conversions are performed if, and only if, they are allowed by the matrix in Figure 6 on page 136.
4. The result of the assignment must not exceed the MAXLEN of the target or of any component of the target.

Examples

See Appendix A, "Scenarios" on page 175.

<BEGIN statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the beginning of a sequence of statements.

Syntax

```
<BEGIN statement> ::=
  [<identifier>:]
  BEGIN
  <terminator>
```

Syntax rule

For every <BEGIN statement>, an <END statement> must be specified to terminate the sequence of statements.

General rule

If an <identifier> is specified, it cannot be referenced.

Examples

None.

<CALL statement>

The following shows the function, syntax, rules, and examples:

Function

Call a named program external to the program containing the conversion plan and pass arguments to its parameters. The called program can operate on the input parameters passed to it and return values in the output parameters for further use by the conversion plan.

Syntax

```
<CALL statement> ::=
  [<identifier>:]
  CALL
  <program name>
  <argument list>
  <terminator>

<program name> ::= <character literal>

<argument list> ::=
  (<argument>{[,<argument>]...})

<argument> ::= <value expression>
```

Syntax rule

See Appendix B, "The DD&C User Exit" on page 203 for details of the exact syntax of the <character literal> allowed as the program name in the CALL statement.

General rules

1. If an <identifier> is specified, it cannot be referenced.
2. In addition to the arguments specified, an argument is passed by the <CALL statement> to provide an area for a condition token to be returned. See Appendix B, "The DD&C User Exit" on page 203
3. On return from the called program, test the returned condition token and if any exception with a severity code greater than 1 (warning) has been set by the called program, the calling plan is terminated.
4. If workspace variables or input plan parameters are used as plan parameter attribute values, they are evaluated before calling any user exit. This implies that changing the value of a workspace variable or input plan parameter with a user exit does not affect the value of the plan parameter attribute.

Examples

See Appendix A, "Scenarios" on page 175.

<CONSTANT statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the declaration of a constant.

Syntax

```
<CONSTANT statement> ::=
  <constant identifier> :
  CONSTANT
  {<literal> | <constant identifier>}
  <terminator>

<constant identifier> ::=
  <identifier> |
  ALGEBRAIC |  DGTLSTBYT | EXACT | FB32 |
  FB64 | FB80 | FH32 | FH64 | FH128 | FI128 |
  FRSBYT | LEFT | LOGICAL | LSTBIT | LSTBYT |
  RIGHT | ROUND | TRUNCATE | ZONFRSBYT | ZONLSTBYT
```

Syntax rules

None.

General rules

1. The value of the <literal> is associated with the <identifier> specified.

2. If CONSTANT(<constant identifier>) is specified, it must not result in a loop.
3. See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.
4. The following constants are defined by ADL for use as attribute values:

   ```
   LSTBIT: CONSTANT 0;     /* Value of <BLNENC attribute>  */
   ROUND: CONSTANT 0;      /* Value of <FIT attribute>     */
   TRUNCATE: CONSTANT 1;   /* Value of <FIT attribute>     */
   EXACT: CONSTANT 2;      /* Value of <FIT attribute>     */
   FB32: CONSTANT 0;       /* Value of <FORM attribute>    */
   FB64: CONSTANT 1;       /* Value of <FORM attribute>    */
   FB80: CONSTANT 2;       /* Value of <FORM attribute>    */
   FH32: CONSTANT 3;       /* Value of <FORM attribute>    */
   FH64: CONSTANT 4;       /* Value of <FORM attribute>    */
   FH128: CONSTANT 5;      /* Value of <FORM attribute>    */
   FI128: CONSTANT 6;      /* Value of <FORM attribute>    */
   LEFT: CONSTANT 0;       /* Value of <JUSTIFY attribute> */
   RIGHT: CONSTANT 1;      /* Value of <JUSTIFY attribute> */
   DGTLSTBYT: CONSTANT 0;  /* Value of <SGNLOC attribute>  */
   ZONFRSBYT: CONSTANT 1;  /* Value of <SGNLOC attribute>  */
   ZONLSTBYT: CONSTANT 2;  /* Value of <SGNLOC attribute>  */
   FRSBYT: CONSTANT 3;     /* Value of <SGNLOC attribute>  */
   LSTBYT: CONSTANT 4;     /* Value of <SGNLOC attribute>  */
   ALGEBRAIC: CONSTANT 0;  /* Value of <SGNCNV attribute>  */
   LOGICAL: CONSTANT 1;    /* Value of <SGNCNV attribute>  */
   ```

   ADL-defined <constant identifier>s having the same value (such as ALGEBRAIC and LEFT) are interchangeable when used to represent attribute values. It is strongly recommended that only those <constant identifier>s described in the “Syntax” section of each attribute specification are used.

Examples

1. Declare `speedlimit` to be a constant with the value of 55.

   ```
   speedlimit: CONSTANT 55;
   ```

2. Constants can be declared using other constants. For example:

   ```
   a: CONSTANT b;
   b: CONSTANT 10;
   ```

   If a constant is referenced from within a plan statement, the constant is ambiguous if more than one occurrence of the constant's declaration is found. If, for example, a second declaration statement contains:

   ```
   b: CONSTANT 'aa';
   ```

   then a reference to `b` from within a plan is ambiguous. A reference to `a` from within a plan is valid, however, because `a` is only declared once and can be resolved within the same declaration.

<data declaration statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the declaration for one or more instances of data.

Syntax

```
<data declaration statement> ::=
  [<identifier>:]... <data>

<data> ::=
  <constructor> |
  {<subtype instance> <terminator>} |
  {<field> <terminator>}

<constructor> ::=
  <ARRAY> |
  <CASE> |
  <SEQUENCE>

<field> ::=
  <ASIS> |
  <BINARY> |
  <BIT> |
  <BITPRE> |
  <BOOLEAN> |
  <CHAR> |
  <CHARPRE> |
  <CHARSFX> |
  <ENUMERATION> |
  <FLOAT> |
  <PACKED> |
  <ZONED>
```

Syntax rules

None.

General rules

1. If a <data declaration statement> specifies an attribute or attributes which are mutually exclusive with the attributes specified in the <DEFAULT statement> for the specified data type, the attribute or attributes on the <data declaration statement> take precedence over the default ones.
2. If multiple <identifier>s are specified, the <data> declaration applies to each of them separately. If specified in the declaration of the elements of a <SEQUENCE>, the <data> declarations apply only to the elements in the order in which the <identifier>s are specified.
3. <data declaration statement>s that are contained within a constructor or SUBTYPE statement can only contain one variable-length field, which must be positioned at the end of the constructor or <SUBTYPE statement>. Therefore, MAXALC(FALSE)

is allowed for this last field and all constructors that include this field. For all other fields, MAXALC(TRUE) is required.

Examples

1. Declare a character string of length 5:

   ```
   x: CHAR LENGTH(5);
   ```

2. Declare a sequence of unnamed fields:

   ```
   s: SEQUENCE BEGIN;
        CHAR LENGTH(5);
        BINARY PRECISION(15);
      END;
   ```

   In this case, the CHAR field is assigned a <positional identifier> of

   ```
   "1" within SEQUENCE s
   ```

   and the BINARY field is assigned a <positional identifier> of

   ```
   "2" within SEQUENCE s.
   ```

<DECLARE statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the descriptions of a set of data values.

Syntax

```
<DECLARE statement> ::=
  [<identifier>:]
  DECLARE
  <DECLARE attributes list>
  <BEGIN statement>
    {
      [<DEFAULT statement>] |
      [<CONSTANT statement>] |
      [<SUBTYPE statement>] |
      <data declaration statement>
    }...
  <END statement>

<DECLARE attributes list> ::=
  {
    [<NOTE attribute>]
    [<HELP attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None.

General rule

1. Within a <DECLARE statement>, only data declarations that lie with the scope of the current <DECLARE statement> can be referenced.
2. See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.

Examples

See Appendix A, “Scenarios” on page 175.

<DEFAULT statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the default values of attributes of a particular data type. These attribute values are to be used if attributes are not otherwise specified in a <data declaration statement> of a data type.

Syntax

```
<DEFAULT statement> ::=
  [<identifier>:]
  DEFAULT
    {
      {ARRAY <ARRAY defaulted attributes list>} |
      {ASIS <ASIS defaulted attributes list>} |
      {BINARY <BINARY defaulted attributes list>} |
      {BIT <BIT defaulted attributes list>} |
      {BITPRE <BITPRE defaulted attributes list>} |
      {BOOLEAN <BOOLEAN defaulted attributes list>} |
      {CASE <CASE defaulted attributes list>} |
      {CHAR <CHAR defaulted attributes list>} |
      {CHARPRE <CHARPRE defaulted attributes list>} |
      {CHARSFX <CHARSFX defaulted attributes list>} |
      {ENUMERATION <ENUMERATION defaulted attributes list>} | .
      {FLOAT <FLOAT defaulted attributes list>} |
      {PACKED <PACKED defaulted attributes list>} |
      {ZONED <ZONED defaulted attributes list>}
    }
  <terminator>
```

Syntax rules

The values of all attributes in a <DEFAULT statement> must be specified by a <literal> or <constant identifier>.

General rules

1. If an <identifier> is specified, it cannot be referenced.
2. The scope of a <DEFAULT statement> for a data type is the entire <DECLARE statement>, regardless of its position within the <DECLARE statement>.
3. Only one <DEFAULT statement> is allowed for each data type.
4. If a <DEFAULT statement> is not specified for an ADL data type, the following <DEFAULT statement> for that data type is used:

```
DEFAULT ARRAY   DMNLOW(1) MAXALC(TRUE) SKIP(0);
DEFAULT ASIS    LENGTH(8) UNITLEN(1);
DEFAULT BINARY  BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
                FIT(ROUND) PRECISION(31) RADIX(2) SCALE(0)
                SGNCNV(LOGICAL) SIGNED(TRUE);
DEFAULT BIT     LENGTH(1);
DEFAULT BITPRE  MAXALC(TRUE) MAXLEN(8) PREBYTRVS(FALSE) PRELEN(16)
                PRESIGNED(TRUE);
DEFAULT BOOLEAN BLNENC(LSTBIT) BYTRVS(FALSE) LENGTH(8);
DEFAULT CASE    MAXALC(TRUE);
DEFAULT CHAR    CCSID(0) JUSTIFY(LEFT) LENGTH(1) UNITLEN(8);
DEFAULT CHARPRE CCSID(0) MAXALC(TRUE) MAXLEN(1) PREBYTRVS(FALSE)
                PRELEN(16) PRESIGNED(TRUE) UNITLEN(8);
DEFAULT CHARSFX CCSID(0) MAXALC(TRUE) MAXLEN(1) UNITLEN(8);
DEFAULT ENUMERATION BYTRVS(FALSE) LENGTH(8)
                SGNCNV(LOGICAL) SIGNED(FALSE);
DEFAULT FLOAT   BYTRVS(FALSE) COMPLEX(FALSE) FIT(ROUND) FORM(FB32)
                RADIX(2);
DEFAULT PACKED  COMPLEX(FALSE) CONSTRAINED(FALSE) FIT(ROUND)
                PRECISION(15) SCALE(0) SGNLOC(DGTLSTBYT) SGNMNS(x'D')
                SGNPLS(x'C') SIGNED(TRUE);
DEFAULT ZONED   COMPLEX(FALSE) CONSTRAINED(FALSE) FIT(ROUND)
                PRECISION(15) SCALE(0) SGNLOC(ZONLSTBYT)
                SGNMNS(x'D') SGNPLS(x'C') SIGNED(TRUE) ZONENC(x'F') ;
```

Examples

1. In the following example, c takes default attributes from the <DEFAULT statement> for BINARY.

   ```
   DEFAULT BINARY BYTRVS(TRUE) PRECISION(31)
                  SGNCNV(ALGEBRAIC) FIT(ROUND);
   c: BINARY PRECISION(15);
   ```

   Thus the complete declaration of c is:

   ```
   BINARY BYTRVS(TRUE) COMPLEX(FALSE) CONSTRAINED(FALSE)
          RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(15) SGNCNV(ALGEBRAIC)
          FIT(ROUND);
   ```

   Note: The value of the PRECISION attribute specified in the <data declaration statement> for c overrides the value specified in the <DEFAULT statement>.

2. For ZONED fields, the SGNLOC(ZONLSTBYT | ZONFRSBYT), SGNMNS, and SGNPLS attributes are mutually exclusive with the SGNLOC(FRSBYT | LSTBYT) and CCSID attributes.

   ```
   DEFAULT ZONED COMPLEX(FALSE) ZONENC(x'F') FIT(ROUND)
                 CONSTRAINED(FALSE) SCALE(0) SGNLOC(ZONLSTBYT)
                 SGNMNS(x'D') SGNPLS(x'C') PRECISION(15) SIGNED(TRUE);
   s: ZONED SGNLOC(FRSBYT) CCSID(0);
   ```

   Thus the complete declaration of s is:

```
ZONED COMPLEX(FALSE) ZONENC(x'F') FIT(ROUND)
      CONSTRAINED(FALSE) SCALE(0) SGNLOC(FRSBYT)
      CCSID(0) PRECISION(15) SIGNED(TRUE);
```

Note: s takes the SGNLOC(FRSBYT) and CCSID(0) attribute values, while the SGNLOC(ZONLSTBYT), SGNMNS(X'D'), and SGNPLS(X'C') attributes and values from the <DEFAULT statement> for <ZONED> are ignored.

<END statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the end of a sequence of statements.

Syntax

```
<END statement> ::=
  [<identifier>: ]
  END
  <terminator>
```

Syntax rule

An <END statement> can be specified only at the termination of a sequence of statements begun by a <BEGIN statement>.

General rule

If an <identifier> is specified, it cannot be referenced.

Examples

None.

<INCLUDE statement>

The following shows the function, syntax, rules, and examples:

Function

Include the ADL text contained in a named file.

Syntax

```
<INCLUDE statement> ::=
  INCLUDE
  <file name>
  <terminator>

<file name> ::=
  <character literal>
```

Syntax rules

1. The named file must contain only ADL text.
2. The included text may contain <INCLUDE statement>s. The maximum number of nested <INCLUDE statements> allowed in DD&C is 32.
3. The <file name> specified must be the local identifier for that file.
4. The <INCLUDE statement> can appear wherever it is valid for any other ADL statements to appear.
5. Included text must not result in a loop of included texts.

General rules

1. The included text replaces the <INCLUDE statement> in the source text of the <parse unit>.
2. A <comment> is not allowed within an <INCLUDE statement>.
3. The current directory is always searched first for the <file name> to be included. Thereafter, the environment variable ADLINC determines the directories to search for the <file name> to be included. If the search path contains network drives and a network error occurs during the search, the search continues without reporting the error.

Example

Include ADL text describing `PersonnelRecord`:

```
INCLUDE 'PERSONNEL.DCL.RECORD';
```

<OTHERWISE statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the declaration to be used or the action to be performed if no <WHEN statement> of a CASE evaluates to TRUE.

Syntax

```
<OTHERWISE statement> ::=
  [<identifier>: ]
  <OTHERWISE clause>

<OTHERWISE clause> ::=
  OTHERWISE
    {
      <data declaration statement> |
      <REJECT statement> |
      <SKIP statement> |
      <terminator>
    }
```

Syntax rules

If a <data declaration statement> is included in an <OTHERWISE statement>, only one optional identifier is allowed for the <data declaration statement>.

General rules

1. If an <identifier> is specified, it is ignored.
2. If OTHERWISE <terminator> is specified, no data exists.
3. If a <SKIP statement> is specified in the <OTHERWISE clause>, no data exists. However, if MAXALC(TRUE) is specified, space exists.
4. If the <REJECT statement> is specified, the ***20—CASE rejected*** exception occurs.

Examples

See "Examples" on page 77 for an example of the <OTHERWISE statement> used in the CASE declaration.

<PLAN statement>

The following shows the function, syntax, rules, and examples:

Function

Specify a plan for converting a set of data values from one representation to another.

Syntax

```
<PLAN statement> ::=
  <identifier>:
  PLAN
  <parameter list>
  <BEGIN statement>
    {<assignment statement> | <CALL statement>}...
  <END statement>

<parameter list> ::=
  (<parameter>[,<parameter>]...)

<parameter> ::=
  {<INPUT parameter> | <OUTPUT parameter>}

<INPUT parameter> ::=
  <qualified identifier>
    [: INPUT
      {
        [<LENGTH attribute>]
        [<CCSID attribute>]
      }!
    ]
```

```
<OUTPUT parameter> ::=
  <qualified identifier>
  : OUTPUT
    {
      [<MAXLEN attribute>]
      [<CCSID attribute>]
    }!
```

Syntax rules

1. A <qualified identifier> specified in a <parameter> must be that of a <data declaration statement>.
2. The value of <MAXLEN attribute> and <LENGTH attribute> must be greater than zero. These attributes specify the actual buffer size in bytes passed to the Conversion Plan Executor component of DD&C.
3. Constants cannot be used as plan parameters.

General rules

1. The statements contained within a PLAN are executed in the order specified by the <PLAN statement>.
2. If neither INPUT nor OUTPUT is specified for a <parameter>, INPUT is assumed.
3. Multiple assignments to the same <qualified identifier> or its components result in only the final assignment being effective.
4. If the <LENGTH attribute> is specified on an <INPUT parameter>, its value is equal to the declared length of the data in bytes. Otherwise,
   - If the value is less than the declared length of the data, the ***16—Input area too short*** exception occurs.
   - If the value is greater than the declared length of the data, the excess data is ignored.
5. If the <MAXLEN attribute> is specified on an <OUTPUT parameter>, its value shall be greater than or equal to the declared length of the data in bytes. Otherwise, the ***17—Output area too short*** exception occurs.
6. For every <qualified identifier> specified in the <parameter list> of a <PLAN statement>, the caller of the plan must specify the address of an area at which a value is located or at which a value can be returned. The <data declaration statement> associated with the <qualified identifier> provides a template that determines how this area is to be interpreted.
7. If the <qualified identifier> specified in the <parameter list> is that of a <constructor>, the value specified must be a composite value whose components are declared by the components of the <constructor>.
8. In addition to the parameters specified in the <parameter list>, the caller of the plan must specify the address in which the condition token is to be returned. The ADL communications area (FMTADLCA) of the condition token is described in SMARTdata UTILITIES for AIX: Data Description and Conversion.

9. Variable plan-parameter attributes can only be input parameters or workspace variables, not output parameters. They are evaluated once only, when execution of the plan begins. If, for example, the attribute value is a parameter in a CALL statement and a user-written program changes the value, this has the following consequences:

   - The parameter value changes for subsequent calls or assignment statements.
   - The plan parameter value remains unchanged.

10. See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.
11. The maximum allowed number of input and output plan parameters is <max8>.

Examples

See Appendix A, “Scenarios” on page 175.

<REJECT statement>

The following shows the function, syntax, rules, and examples:

Function

Specify that the ***20—CASE rejected*** exception occurs when specified on a <WHEN statement> or <OTHERWISE statement>.

Syntax

```
<REJECT statement> ::=
  [<identifier>:]
  REJECT <terminator>
```

Syntax rules

None.

General rules

If an <identifier> is specified, it is ignored.

Examples

1. Declare a case where if the C1 is chosen, the plan is terminated due to bad data:

```
s: SEQUENCE
   BEGIN;
   a: CHAR;
   CASE BEGIN;
     c1:  WHEN a = '1'
          THEN REJECT;
     c2:  WHEN a = '2'
          THEN x: BINARY SCALE(5) PRECISION(15);
          OTHERWISE y: FLOAT;
        END;
   END;
```

2. Declare a case where if the <OTHERWISE statement> is chosen, the plan is terminated:

   ```
   s: SEQUENCE
      BEGIN;
      a: CHAR;
      CASE BEGIN;
        c1:  WHEN a = '1'
             THEN y: FLOAT;
        c2:  WHEN a = '2'
             THEN x: BINARY SCALE(5) PRECISION(15);
             OTHERWISE REJECT;
           END;
      END;
   ```

<SKIP statement>

The following shows the function, syntax, rules, and examples:

Function

Specify the number of bits to be skipped in the data of a constructor.

Syntax

```
<SKIP statement> ::=
  [<identifier>:]
  SKIP({<positive integer> | <constant identifier>})
  <terminator>
```

Syntax rules

None.

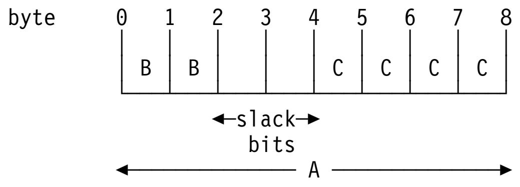
General rules

1. If a <constant identifier> is specified, it must be one whose value is a <positive integer>.
2. If an <identifier> is specified, it cannot be referenced.
3. If you specify the AUTOSKIP compiler option, <SKIP statement>s are generated automatically according to the ADL alignment rules. When this option is not specified, <SKIP statement>s must be inserted in the appropriate places in the ADL source code. The ADL declaration translator component of DD&C checks that this has been done and returns an error if the alignment of a field is incorrect.

Examples

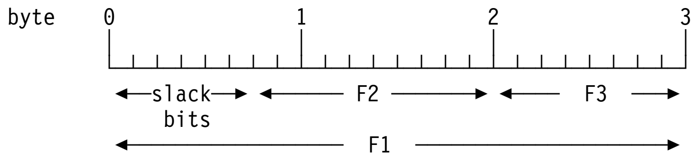
1. Declare a sequence with 16 slack bits between `B` and `C` :

   ```
   A  : SEQUENCE BEGIN;
        B  : CHAR LENGTH(2);
             SKIP(16);
        C  : BINARY PRECISION(31);
        END;
   ```



2. Declare a sequence, F1, with 6 slack bits at the beginning of the sequence:

```
F1 : SEQUENCE BEGIN;
          SKIP(6);
     F2 : BIT LENGTH(10);
     F3 : CHAR;
     END;
```



<SUBTYPE statement>

The following shows the function, syntax, rules, and examples:

Function

Declare a subtype of an ADL type or of another subtype. A subtype specifies the attributes and components of all instances of the subtype. A subtype inherits the components and attributes of the type that declares it, but it can override those attributes.

Syntax

```
<SUBTYPE statement> ::=
  <subtype identifier>:
  SUBTYPE
  OF
    {
      <constructor> |
      {<field> <terminator>} |
      {<subtype instance> <terminator>}
    }

<subtype identifier> ::= <identifier>
```

Syntax rule

If a <subtype instance> is specified, the referenced subtype must not result in a loop of subtype references.

General rule

See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.

Examples

1. Specify `name` as a subtype of <SEQUENCE>:

   ```
     name:
        SUBTYPE OF SEQUENCE
        BEGIN;
          lastname: CHAR LENGTH(12);
          firstname: CHAR LENGTH(12);
          initial: CHAR;
        END;
   ```

2. Specify `address` as a subtype of <SEQUENCE>:

   ```
   address: SUBTYPE OF SEQUENCE
            BEGIN;
               street: CHAR LENGTH(30);
               city: CHAR LENGTH(15);
               state: CHAR LENGTH(2);
               zip: CHAR LENGTH(10);
            END;
   ```

3. Declare `card` to be a sequence containing instances of `name` and `address`. If <identifier>s are specified for the `name` and `address` components of `card`, the <identifier>s `alias` and `drop` replace `name` and `address` in the fully-qualified names of the components of `name` and `address`.

   ```
   card: SEQUENCE
         BEGIN;
           alias: name;
           drop: address;
         END;
   ```

   The qualified names of the fields in `card` are:

   ```
   card.alias.lastname
   card.alias.firstname
   card.alias.initial
   card.drop.street
   card.drop.city
   card.drop.state
   card.drop.zip
   ```

4. Declare a subtype of a subtype:

   ```
   fixnum: SUBTYPE OF BINARY;
   partscount: SUBTYPE OF fixnum PRECISION(15);
   ```

5. Declare a subtype of a case:

```
answer: SUBTYPE OF CASE
                   BEGIN;
                     WHEN a = '1'
                     THEN b: CHAR LENGTH(4);
                     WHEN a = '2'
                     THEN c: BINARY;
                     OTHERWISE d: FLOAT;
                   END;
x: SEQUENCE
   BEGIN;
     a: CHAR;
        answer;
   END;
```

<WHEN statement>

The following shows the function, syntax, rules, and examples:

Function

Specify in a CASE declaration the conditions under which a <THEN clause> is to be selected.

Syntax

```
<WHEN statement> ::=
  [<identifier>:]
  <WHEN clause>
  <THEN clause>

<THEN clause> ::=
  THEN
    {
      <data declaration statement> |
      <REJECT statement> |
      <SKIP statement> |
      <terminator>
    }
```

Syntax rule

If a <data declaration statement> is included in an <OTHERWISE statement>, only one optional identifier is allowed for the <data declaration statement>.

General rules

1. If THEN <terminator> is specified in the <THEN clause>, no data exists.
2. If a <SKIP statement> is specified in the <THEN clause>, no data exists. However, if MAXALC(TRUE) is specified, space must exist.
3. If the <REJECT statement> is specified, the ***20—CASE rejected*** exception occurs.

Examples

See “Examples” on page 77 for an example of the <WHEN statement> used in the CASE declaration.

Chapter 4. Data Declarations

This chapter describes, in alphabetical order, the data types that can be declared in ADL.

<ARRAY>

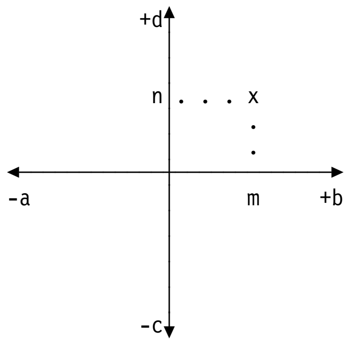
The following shows the function, syntax, rules, and examples:

Function

An array is a collection whose elements are uniquely associated with the integer values of one or more attributes. By geometric analogy, these attributes are called *dimensions*, but their actual meaning is determined by the application.

Each element of an array has its own value, and is associated with a unique combination of dimension, or attribute, values. The array as a whole allows elements to exist at all possible combinations of values within the defined range of each dimension.

The elements of an array are addressable through integer values specified for each dimension. As an example, consider an array of two dimensions, where one dimension has integer values from -a to +b and the other dimension has integer values from -c to +d. A given element of the array, such as x in the following figure, is addressed by the integer values m and n, where m is in the range -a to +b and n is in the range -c to +d. Higher-order arrays have more dimensions and their elements are addressed by more integer values.



The dimensions of an array are defined and addressed as an ordered list.

The range of a dimension of an array can be described in terms of either its <DMNLOW attribute> and <DMNHIGH attribute> or its <DMNLOW attribute> and <DMNSIZE attribute>.

The total number of elements E in an array is the product of the e_i of its dimensions:

$$E = PRODUCT(e_i)$$

Where:

i is from 1 to n
e_i is the extent of the i^{th} dimension.

When an array is mapped onto a linear address space, the dimensions of the array must also be linear. For a one-dimensional array X with indexes in the range -1 to 2, this is straightforward. The memory mapping is X(-1) X(0) X(+1) X(+2).

Multidimensional arrays are mapped onto a linear address space by iterating the indexes of each low order dimension through their ranges for each higher order dimension.

The elements of an array are mapped to memory as contiguous fields. They are mapped such that the last dimension varies the fastest.

- A 3-element by 5-element two-dimensional array named A is stored linearly in memory as follows:

a(1,1)	a(1,2)	a(1,3)	a(1,4)	a(1,5)	a(2,1)	a(2,2)	a(2,3)	a(2,4)

a(2,5)	a(3,1)	a(3,2)	a(3,3)	a(3,4)	a(3,5)

- A 2-element one-dimensional array, X, consisting of the sequence A,B, is stored linearly in memory as follows:

x(1).A	x(1).B	x(2).A	x(2).B

The <DMNLOW attribute> and <DMNHIGH attribute> or <DMNLOW attribute> and <DMNSIZE attribute> of each dimension of an array can be specified by a signed integer or by reference to a field containing a signed integer value.

When an array is defined without reference fields, each program that exports or imports the array must know the values of the <DMNLOW attribute>, <DMNHIGH attribute>, and <DMNSIZE attribute> of the array.

When an array is defined with reference fields, a combination of the <DMNLOW attribute>, <DMNHIGH attribute>, and <DMNSIZE attribute> values for each variable dimension can also be included in the data. Then, when the data is imported by another program, the locations of each element can be determined again.

Thus, when describing the dimensions of an array, the following must be specified for each dimension:

```
►►──┬────────────────┬──┬─<DMNHIGH attr.>─┬──┬───────────────┬──►◄
    └─<DMNLOW attr.>─┘  └─<DMNSIZE attr.>─┘  └─<DMNMAX attr.>─┘
```

Where:

- If not specified, the single <DMNLOW attribute> value specified with the ARRAY, SUBTYPE, or DEFAULT statement is used, following the normal hierarchy of attributes.
- Either the <DMNHIGH attribute> or <DMNSIZE attribute> is required
- Either an integer value or a field reference must be specified for the <DMNLOW attribute>, <DMNHIGH attribute>, or <DMNSIZE attribute>
- The <DMNMAX attribute> must be specified if either the <DMNSIZE attribute> is specified and is variable, or both the <DMNLOW attribute> and <DMNHIGH attribute> are specified and at least one of them is variable.

A further consideration for arrays defined with reference fields is whether space is allocated for the maximum number of elements in each dimension, MAXALC(TRUE), or for only the actual number of elements in each dimension, MAXALC(FALSE).

- A two-dimensional, MAXALC(TRUE) array specified with reference fields is mapped into memory as follows:

```
DEFAULT ARRAY DMNLOW(1) MAXALC(TRUE) SKIP(0);
xmp1: SEQUENCE
      BEGIN;
        s: BINARY;
        t: BINARY;
        z: ARRAY  DMNLST(DMNHIGH(s) DMNMAX(4),
                         DMNHIGH(t) DMNMAX(4))
           OF BINARY;
      END;
```

		active	active	inactive	inactive	active	active	inactive	inactive
s = 2	t = 2	z(1,1)	z(1,2)	z(1,3)	z(1,4)	z(2,1)	z(2,2)	z(2,3)	z(2,4)

inactive	inactive	inactive	inactive	active	active	active	active
z(3,1)	z(3,2)	z(3,3)	z(3,4)	z(4,1)	z(4,2)	z(4,3)	z(4,4)

Note: z(1,1) z(1,2) z(2,1) z(2,2) contain the active elements for this array as defined by the <DMNHIGH attribute>s. z(1,3) z(1,4) z(2,3) z(2,4) z(3,1) z(3,2) z(3,3) z(3,4) z(4,1) z(4,2) z(4,3) z(4,4) are inactive elements which exist as defined by the <DMNMAX attribute>s and MAXALC(TRUE) specified for the array.

Contrary to the above example, if MAXALC(FALSE) is specified, only the required space for a dimension of an array defined with reference fields is allocated.

- A two-dimensional, MAXALC(FALSE) array specified with reference fields is mapped into memory as follows:

```
DEFAULT ARRAY DMNLOW(1) MAXALC(TRUE) SKIP(0);
xmp2: SEQUENCE
      BEGIN;
        s: BINARY;
        t: BINARY;
        z: ARRAY MAXALC(FALSE) DMNLST(DMNHIGH(s) DMNMAX(4),
                                      DMNHIGH(t) DMNMAX(4))
           OF BINARY;
      END;
               active active active active
 |      |      |      |      |      |      |
 |s = 2 |t = 2 |z(1,1)|z(1,2)|z(2,1)|z(2,2)|
 +------+------+------+------+------+------+
```

Note: z(1,1) z(1,2) z(2,1) z(2,2) contain active elements as defined by the <DMNHIGH attribute>s and MAXALC(FALSE) specified for the array.

An algorithm is available for addressing individual elements within an array. For an array (A) with (i) dimensions (d)

$A(d_1, d_2, \ldots d_i)$

the algorithm for addressing individual elements within an array is:

$((..((d_1e_2 + d_2)e_3 + d_3)$
$e_4 + \ldots + d_{i-1})e_i + d_i) * (s + SKIP)$
$+ base - (..((DMNLOW_1e_2 + DMNLOW_2)$
$e_3 + DMNLOW_3)e_4 + \ldots + DMNLOW_{i-1})$
$e_i + DMNLOW_i) * (s + SKIP)$

where:

d	= dimension
s	= size of element
e	= extent
base	= relative offset of start of array from start of data
DMNLOW	= low bound of subscript (obtained from DMNLOW attribute)
SKIP	= bits skipped before each element except the first element (obtained from the SKIP attribute)

The value for e_n is determined according to the array type.

- For a MAXALC(TRUE) array,

  e_n = <DMNMAX attribute>

- For a MAXALC(FALSE) or an array defined without reference fields,
  - if the high bound of the array is given,

    e_n = <DMNHIGH attribute> - <DMNLOW attribute> + 1

  - if the size of the array is given,

    e_n = <DMNSIZE attribute>

where

e_n	= extent of the n^{th} dimension
n	= dimension number

If, for all the dimensions, the values for the <DMNHIGH attribute>, <DMNLOW attribute>, or <DMNSIZE attribute> are signed integer or constants, the second part of the algorithm can be calculated at conversion plan build time. However, if an attribute contains a reference, the referenced value is obtained at conversion time.

If DMNLOW is equal to zero, the array element address reference simplifies to:

$((..((d_1e_2 + d_2)e_3 + d_3)$
$e_4 + ... + d_{n-1})e_n + d_n) * (s + SKIP) + base$

Syntax

```
<ARRAY> ::=
  ARRAY
  <ARRAY attributes list>
  OF
    {
      {<field><terminator>} |
      <CASE> |
      <SEQUENCE> |
      {<subtype instance><terminator>}
    }

<ARRAY attributes list> ::=
  {
    <DMNLST attribute>
    <ARRAY defaulted attributes list>
    <ARRAY optional attributes list>
  }!

<ARRAY defaulted attributes list> ::=
  {
    [<DMNLOW attribute>]
    [<MAXALC attribute>]
    [<SKIP attribute>]
  }!

<ARRAY optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

1. The attributes of the <ARRAY defaulted attributes list> are specified in the ARRAY <DEFAULT statement> or the <ARRAY> <data declaration statement>.
2. An array of <subtype instance> is not specified if the <subtype instance> is a subtype of ARRAY.
3. All elements of an ARRAY have the same length.

General rules

1. The length of an array in bits is the product of the number of elements in the array (E) times the length in bits of each element, plus (E-1) times the <SKIP attribute> of the array.
2. The minimum alignment of ARRAY elements is byte alignment.
3. If MAXALC(TRUE) is specified and any attribute of the <DMNLST attribute>s has a <qualified identifier> for a value, space is allocated for the maximum number of elements (as defined by the <DMNMAX attribute>) in each dimension. The actual number of active elements in each dimension is specified by the reference field.
4. If MAXALC(FALSE) is specified and any attribute of the <DMNLST attribute>s has a <qualified identifier> for a value, space is allocated as defined by the attributes of each dimension in the <DMNLST attribute>s.

Examples

1. Declare an array of twelve numbers for the number of days in each month in a year.

   ```
   MonthsInYr: ARRAY DMNLST(DMNSIZE(12))
               OF BINARY PRECISION(15);
   ```

2. Declare an array to specify which seats of an airplane have been reserved. The number of rows and the number of seats per row are specified as reference fields to account for different airplane configurations. The maximum number of rows in an airplane is 50 and the maximum number of seats in a row is 10.

   ```
   PlaneReservations:
       SEQUENCE BEGIN;
         rows: BINARY PRECISION(15);
         seats: BINARY PRECISION(15);
         ReservedSeats: ARRAY DMNLST(DMNMAX(50) DMNSIZE(rows),
                                     DMNMAX(10) DMNSIZE(seats))
                          OF BOOLEAN;
       END;
   ```

<ASIS>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the ASIS type. The actual type of this data is unknown.

The length of an ASIS field in bits is specified with the LENGTH attribute multiplied by UNITLEN. The value of the LENGTH attribute can be an integer literal or a reference to another field. If specified by an integer literal, all instances of the variable are of the same length. But if specified by reference to another field, then each instance can be of a different length. In this case, however, the maximum length of the string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length.

Syntax

```
<ASIS> ::=
  ASIS
  <ASIS attributes list>

<ASIS attributes list> ::=
  {
    <ASIS defaulted attributes list>
    <ASIS optional attributes list>
  }!

<ASIS defaulted attributes list> ::=
  [
    LENGTH({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})
  ]! |
  [
    LENGTH(*)
    MAXLEN({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})]
  ]! |
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})]
  ]!

<ASIS optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<max ASIS> ::=
  See Syntax Rule 3.
```

Syntax rules

1. A field is declared an ASIS field if its type cannot be expressed in ADL.
2. A field is declared an ASIS field if its data value is not to be converted in <assignment statement>s.
3. The LENGTH or MAXLEN of an ASIS field is specified in terms of units whose length in bits is specified by UNITLEN. The length of an ASIS field in bits is determined by (LENGTH * UNITLEN), and the greatest value that can be specified for LENGTH is determined by (<max31> / UNITLEN). The maximum length of an ASIS field in bits is determined by (MAXLEN * UNITLEN), and the greatest value that can be specified for MAXLEN is determined by (<max31> / UNITLEN).
4. If LENGTH(<qualified identifier>) is specified, the referenced field must be BINARY, PACKED, or ZONED with SCALE(0) and COMPLEX(FALSE) and a maximum value of <max ASIS>.

5. LENGTH(*) can be specified only for the last element of all containing <constructor>s.
6. LENGTH(*) cannot be specified for any element of an ARRAY.

General rules

1. If LENGTH(*) is specified, the length of the ASIS field is the number of bits from the position of the ASIS field in the data to the end of the data. The length in bits of the ASIS field is the "actual field length", as defined in "Actual Field Length" on page 137. In particular, for LENGTH(*) this is the largest multiple of the UNITLEN value that fits within the data.
2. The minimum alignment of an <ASIS> field is bit alignment.

Examples

1. Declare a program pointer field `arrptr`:

   Note: Since pointers cannot be declared in ADL, `arrptr` is defined as an <ASIS> data type.

   ```
   DEFAULT ASIS LENGTH(8) UNITLEN(1);
   arrptr: ASIS LENGTH(32);
   ```

2. Declare `dontcare` to be a 50-bit field in which no conversions are to take place:

   ```
   DEFAULT ASIS LENGTH(8) UNITLEN(1);
   dontcare: ASIS LENGTH(50);
   ```

<BINARY>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the BINARY type—a fixed-point, binary-encoded numeric field.

1. A fixed-point, binary-encoded number is represented as a bit string, as shown in Figure 3 on page 65.
   - For an unsigned number (SIGNED(FALSE)), all bits are used to express the binary encoding of the absolute value of the number.
   - For a signed number (SIGNED(TRUE)), the first bit represents the sign, and the remaining bits represent the binary encoding of the number. Positive numbers are represented in true binary notation with the sign bit set to zero. Negative numbers are represented in two's complement notation with a 1 in the sign bit position. The two's complement of a number is obtained by inverting each bit and adding a 1 in the low order bit position.
2. The length of the bit string is specified by the <LENGTH attribute>.
3. For RADIX(2):

a. The maximum number of significant digits in a BINARY field is specified by the <PRECISION attribute> for CONSTRAINED(TRUE); otherwise, it is implied by the <LENGTH attribute>. Binary precision is indicated by RADIX(2).

b. Binary scaling is indicated by RADIX(2) and the scaling factor is specified as a power of 2. The actual value of a fixed-point number is given by the formula:

```
Actual_Value = Stored_Value * 2**(-SCALE)
```

If the scaling factor is negative, then the actual value is greater than the stored value.

c. Stored_value

- If CONSTRAINED(TRUE) is specified:
  - For signed numbers, the range is -(2**PRECISION) to (2**PRECISION)-1.
  - For unsigned numbers, the range is 0 to (2**PRECISION)-1.
- If CONSTRAINED(FALSE) is specified:
  - For signed numbers, the range is -(2**(LENGTH-1)) to (2**(LENGTH-1))-1.
  - For unsigned numbers, the range is 0 to (2**LENGTH)-1.

d. Actual_value

- If CONSTRAINED(TRUE) is specified:
  - For signed numbers, the range is -(2**PRECISION)*(2**(-SCALE)) to ((2**PRECISION)-1)*(2**(-SCALE)).
  - For unsigned numbers, the range is 0 to ((2**PRECISION)-1)*(2**(-SCALE)).
- If CONSTRAINED(FALSE) is specified:
  - For signed numbers, the range is -(2**(LENGTH-1))*(2**(-SCALE)) to ((2**(LENGTH-1))-1)*(2**(-SCALE)).
  - For unsigned numbers, the range is 0 to ((2**LENGTH)-1)*(2**(-SCALE)).

e. A decimal fraction can be represented exactly in a RADIX(2) BINARY field only if it is a sum of powers of two and all its significant bits can be encoded in the stored value.

4. For RADIX(10):

a. The maximum number of decimal digits that can be encoded in a BINARY field is specified by the <PRECISION attribute> for CONSTRAINED(TRUE); otherwise, it is implied by the <LENGTH attribute>.

b. Decimal scaling is indicated by RADIX(10) and the scaling factor is specified as a power of 10. The actual value of a fixed-point number is given by the formula:

```
Actual_Value = Stored_Value * 10**(-SCALE)
```

If the scaling factor is negative, then the actual value is greater than the stored value. For example, if decimal 123 is stored in RADIX(10) and it has a scaling factor of -3, then the actual value is 123000. If the scaling factor is positive, then the actual value is equal or less than the stored value. For example, if decimal 123 is stored and it has a scaling factor of 3, then .123 is the actual value.

c. Stored_value

  - If CONSTRAINED(TRUE) is specified:
    - For signed numbers, the range is -((10**PRECISION)-1) to (10**PRECISION)-1.
    - For unsigned numbers, the range is 0 to (10**PRECISION)-1.
  - If CONSTRAINED(FALSE) is specified:
    - For signed numbers, the range is -(2**(LENGTH-1)) to (2**(LENGTH-1))-1.
    - For unsigned numbers, the range is 0 to (2**LENGTH)-1.

d. Actual_value

  - If CONSTRAINED(TRUE) is specified:
    - For signed numbers, the range is -((10**PRECISION)-1)*(10**(-SCALE)) to ((10**PRECISION)-1)*(10**-SCALE).
    - For unsigned numbers, the range is 0 to ((10**PRECISION)-1)*(10**(-SCALE)).
  - If CONSTRAINED(FALSE) is specified:
    - For signed numbers, the range is -(2**(LENGTH-1))*(10**(-SCALE)) to ((2**(LENGTH-1))-1)*(10**(-SCALE)).
    - For unsigned numbers, the range is 0 to ((2**LENGTH)-1)*(10**(-SCALE)).

e. A decimal fraction can be stored exactly for RADIX(10) fields only if it is a sum of the powers of 10, and all its significant bits can be encoded in the stored value.

```
         DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
                        RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
                        SGNCNV(ALGEBRAIC) FIT(ROUND);
         X: BINARY RADIX(10) PRECISION(7) SCALE(3) CONSTRAINED(TRUE);
Is represented by the 32-bit, signed, <BINARY> field:
```

s	o	o	o	o	o	o	o	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p	p

Where the bits are labeled as follows:

s Sign bit (1 bit)

o Overflow bits.
7 additional bits that can be used to increase the range of
specifiable integers if CONSTRAINED(FALSE) is specified.

p Bits in which the significant digits of the value are stored
as an integer.
24 bits required to represent a value up to 9999.999

Figure 3. Layout of a signed BINARY field

Syntax

```
<BINARY> ::=
  BINARY
  <BINARY attributes list>

<BINARY attributes list> ::=
  {
    <BINARY defaulted attributes list>
    <BINARY optional attributes list>
  }!

<BINARY defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [<PRECISION attribute>]
    [<RADIX attribute>]
    [<SCALE attribute>]
    [<SGNCNV attribute>]
    [<SIGNED attribute>]
  }!
```

```
<BINARY optional attributes list> ::=
  {
    [<HELP attribute>]
    [LENGTH({1..32 | <constant identifier>})]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

1. For RADIX(2) fields:
   - SCALE multiplied by -1 specifies the scaling factor of the field as a power of 2.
   - If SIGNED(TRUE) is specified, LENGTH is greater than or equal to PRECISION+1.
   - If SIGNED(FALSE) is specified, LENGTH is greater than or equal to PRECISION and PRECISION is greater than or equal to 1.
2. For RADIX(10) fields:
   - SCALE multiplied by -1 specifies the scaling factor of the field as a power of 10.
   - If SIGNED(TRUE) is specified, LENGTH is greater than or equal to CEIL(PRECISION * 3.32) + 1.
   - If SIGNED(FALSE) is specified, LENGTH is greater than or equal to CEIL(PRECISION * 3.32).
   - PRECISION is greater than or equal to 1.

General rules

1. If the <LENGTH attribute> is not specified, it is determined from the specified PRECISION:

```
                    Unsigned BINARY Numbers
RADIX(10) PRECISION   RADIX(2) PRECISION        LENGTH
-------------------   ------------------      -----------
      1 - 4                 1 - 16                16
      5 - 9                17 - 32                32
                     Signed BINARY Numbers
RADIX(10) PRECISION   RADIX(2) PRECISION        LENGTH
-------------------   ------------------      -----------
      1 - 4                 0 - 15                16
      5 - 9                16 - 31                32
```

2. For RADIX(10), the number of bits required to encode the specified number of significant digits is determined by the CEIL(PRECISION * 3.32).
3. The minimum alignment of an BINARY field is byte alignment.
4. If COMPLEX(TRUE) is specified, the total length of the field is the sum of the lengths of the two parts of the complex number (Real and Imaginary), plus any skip space necessary to ensure that the second part of the number is byte-aligned.

Examples

In the following examples, the bits of a BINARY field are defined by:

s = sign bit
o = overflow bit
p = precision bit

RADIX(2) examples

1. Signed RADIX(2) BINARY field:

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
x: BINARY PRECISION(15);

bit |s|p|p|p|p|p|p|p|p|p|p|p|p|p|p|p|
    |1|1|1|1|1|1|1|1|1|1|1|1|0|0|0|0|
    └───────────────────────────────┘

Field low bound  = -32768
Field high bound = 32767
Scale factor     = 2**(-0) = 1
Stored value     = X'FFF0' = -16
Actual value     = -16 * 1 = -16
```

2. Unsigned RADIX(2) BINARY field:

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
x: BINARY PRECISION(16) SIGNED(FALSE);

bit |p|p|p|p|p|p|p|p|p|p|p|p|p|p|p|p|
    |1|1|1|1|1|1|1|1|1|1|1|1|0|0|0|0|
    └───────────────────────────────┘

Field low bound  = 0
Field high bound = 65535
Scale factor     = 2**(-0) = 1
Stored Value     = X'FFF0' = 65,520
Actual Value     = 65,520 * 1 = 65,520
```

3. Signed RADIX(2) BINARY field with precision less than length. The overflow bits in this field cannot be used to store significant digits.

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
x: BINARY PRECISION(11) CONSTRAINED(TRUE);

bit |s|o|o|o|o|p|p|p|p|p|p|p|p|p|p|p|
    |0|0|0|0|0|1|1|1|1|1|0|0|0|1|1|0|

Field low bound  = -2048
Field high bound = 2047
Scale factor     = 2**(-0) = 1
Stored value     = X'07C6' = 1,990
Actual value     = 1,990 * 1 = 1,990
```

4. Unsigned RADIX(2) BINARY field with precision less than length. The overflow bits in this field can be used to store significant digits.

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
X: BINARY PRECISION(11) SCALE(5) SIGNED(FALSE);

bit |o|o|o|o|o|p|p|p|p|p|p|p|p|p|p|p|
    |0|0|0|0|0|0|0|1|0|1|1|0|0|0|0|0|

Field low bound  = 0
Field high bound = 2047.96875
Scale factor     = 2**(-5) = 0.03125
Stored value     = X'0B0' = 176
Actual value     = 176 * 0.03125 = 5.50000
```

5. Unsigned RADIX(2) BINARY number with precision less than length and scale greater than precision. The overflow bits in this field can be used to store significant digits.

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
X: BINARY PRECISION(6) SCALE(8);

bit |s|o|o|o|o|o|o|o|o|o|p|p|p|p|p|p|
    |1|1|1|0|1|0|0|0|1|0|1|1|0|0|0|0|

Field low bound  = -128.00000000
Field high bound = 127.99609375
Scale factor     = 2**(-8) = 0.00390625
Stored value     = X'E8B0' = -5968
Actual value     = -(5968 * 0.00390625) = -23.31250000
```

RADIX(10) examples

1. Signed RADIX(10) BINARY field with a scale less than zero.

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
X: BINARY PRECISION(3) LENGTH(16) SCALE(-4)
          RADIX(10) CONSTRAINED(TRUE);

bit |s|o|o|o|o|o|p|p|p|p|p|p|p|p|p|p|
    |0|0|0|0|0|0|0|0|0|1|0|0|0|0|0|0|
    └─────────────────────────────────┘

Field low bound  = -9990000
Field high bound = 9990000
Scale factor     = 10**(-(-4)) = 10000
If x is assigned the actual value 640000 decimal, then
stored_integer_value = actual_value * RADIX**SCALE
stored_integer_value = 640000 * 10-4 = 64
stored_integer_value = 64 = X'0040'
Actual value     = stored_integer_value * scale_factor
Actual value     = 64 * 10000 = 640000
```

2. Signed RADIX(10) BINARY field with scale greater than zero.

```
DEFAULT BINARY BYTRVS(FALSE) COMPLEX(FALSE) CONSTRAINED(FALSE)
               RADIX(2) SCALE(0) SIGNED(TRUE) PRECISION(31)
               SGNCNV(ALGEBRAIC) FIT(ROUND);
X: BINARY LENGTH(16) PRECISION(4) SCALE(2)
          RADIX(10) CONSTRAINED(TRUE);

bit |s|o|p|p|p|p|p|p|p|p|p|p|p|p|p|p|
    |0|0|0|0|0|0|0|0|0|1|1|0|0|0|1|1|
    └─────────────────────────────────┘

Field Low Bound  = -99.99
Field High Bound = 99.99
Scale Factor     = 10**(-(2)) = 0.01
If x is assigned the actual value 0.99 decimal, then
stored_integer_value = actual_value * RADIX**SCALE
stored_integer_value = 0.99 * 10**2 = 99
stored_integer_value = 99 = X'0063'
Actual Value     = stored_integer_value * scale_factor
Actual Value     = 99 * 0.01 = 0.99
```

<BIT>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the BIT string type.

The length of a BIT string in bits is specified by the LENGTH attribute. The first bit is at ordinal position 1, and the last bit is at the ordinal position specified by the LENGTH attribute. The value of the LENGTH attribute can be specified by an integer literal or by reference to another field. If specified by an integer literal, all instances of the variable are of the same length. But if specified by reference to another field, then each instance can be of a different length. In this case, however, the maximum length of the string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length.

Syntax

```
<BIT> ::=
  BIT
  <BIT attributes list>

<BIT attributes list> ::=
  {
    <BIT defaulted attributes list>
    <BIT optional attributes list>
  }!

<BIT defaulted attributes list> ::=
  [LENGTH({1..<max31> | <constant identifier>})] |
  [
    LENGTH(*)
    MAXLEN({1..<max31> | <constant identifier>})
  ]! |
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max31> | <constant identifier>})
  ]!

<BIT optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

1. If LENGTH(<qualified identifier>) is specified, the referenced field is BINARY, PACKED, or ZONED with SCALE(0) and COMPLEX(FALSE) and a maximum value of <max31>.

2. LENGTH(*) can be specified only for the last element of all containing <constructor>s.

3. LENGTH(*) cannot be specified for any element of an ARRAY.

General rules

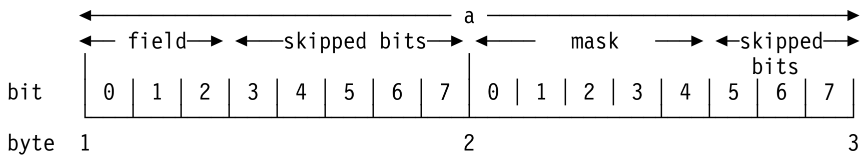
1. The field length is the actual field length, as described in “Actual Field Length” on page 137.

2. The minimum alignment of a BIT field is bit alignment.

Examples

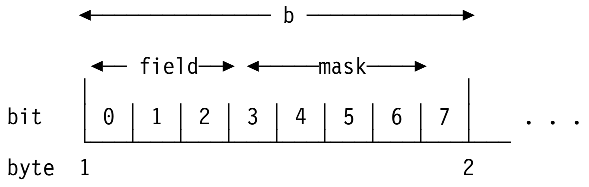
1. Declare a `mask` of 5 bits that is located in the next byte:

```
a: SEQUENCE;
   BEGIN;
     field: BIT LENGTH(3);
     SKIP(5);
     mask: BIT LENGTH(5);
     SKIP(3);
   END;
```



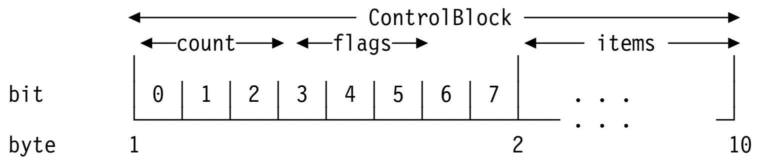
2. Declare a `mask` of 4 bits that is located following the last bit of the preceding field:

```
b: SEQUENCE;
   BEGIN;
     field: BIT LENGTH(3);
     mask: BIT LENGTH(4);
     SKIP(1);
   END;
```



3. Declare `flags` of three bits that is located following the last bit of the preceding field. Note that two slack bits follow `flags` since the `items` field consists of characters represented by bytes.

```
ControlBlock: SEQUENCE
   TITLE('This is a dummy control block')
   BEGIN;
      count: BINARY PRECISION(3) LENGTH(3) SIGNED(FALSE);
      flags: BIT LENGTH(3);
      SKIP(2);
      items: CHAR LENGTH(8);
   END;
```



<BITPRE>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the BITPRE type, a variable-length bit string with a BINARY prefix.

Since each instance can be a different length, the maximum length of the BITPRE string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length.

Syntax

```
<BITPRE> ::=
  BITPRE
  <BITPRE attributes list>

<BITPRE attributes list> ::=
  {
    <BITPRE defaulted attributes list>
    <BITPRE optional attributes list>
  }!
```

```
<BITPRE defaulted attributes list> ::=
  {
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max BITPRE> |
          <constant identifier>
        })
    ]
    [<PREBYTRVS attribute>]
    [<PRELEN attribute>]
    [<PRESIGNED attribute>]
  }!
<BITPRE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rule

The <MAXLEN attribute> does not include the length of the prefix.

General rules

1. The maximum length of a BITPRE field is determined by MAXLEN. The greatest value (<max BITPRE>) that can be specified for MAXLEN is determined by (2**(PRELEN-1)-1).
2. The prefix is encoded as a BINARY number with the attributes COMPLEX(FALSE) CONSTRAINED(FALSE) RADIX(2) SCALE(0) SGNCNV(LOGICAL) FIT(ROUND) and the attributes PREBYTRVS, PRESIGNED, and PRELEN specified for the BITPRE data type. PRECISION is the maximum value allowed for the specified LENGTH and SIGNED values.
3. The actual length in bits of the variable-length bit string is specified by the length prefix integer that immediately precedes the bit string.
4. The amount of space for a BITPRE field with MAXALC(TRUE) is MAXLEN + PRELEN.
5. The amount of space for a BITPRE field with MAXALC(FALSE) is PRELEN plus the actual length of the bit string.
6. The minimum alignment of an BITPRE field is byte alignment.
7. The value of the prefix does not include the length of the prefix.

Examples

Declare `X` to be a variable-length bit string with a maximum length of 128 bits in which space will only be allocated for the current instance.

```
X: BITPRE MAXLEN(128) MAXALC(FALSE);
```

<BOOLEAN>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the BOOLEAN type.

Syntax

```
<BOOLEAN> ::=
  BOOLEAN
  <BOOLEAN attributes list>

<BOOLEAN attributes list> ::=
  {
    <BOOLEAN defaulted attributes list>
    <BOOLEAN optional attributes list>
  }!
```

```
<BOOLEAN defaulted attributes list> ::=
  {
    [<BLNENC attribute>]
    [<BYTRVS attribute>]
    [LENGTH({1..64 | <constant identifier>})]
  }!
```

```
<BOOLEAN optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None

General rules

- The minimum alignment of a BOOLEAN field is byte alignment.
- See "<BLNENC attribute>" on page 103 for details of how to specify the attributes of a BOOLEAN value.

Examples

Declare `living` to be a BOOLEAN value where TRUE is encoded in a 32-bit string:

```
DEFAULT BOOLEAN BLNENC(LSTBIT) BYTRVS(FALSE) LENGTH(8);
living: BOOLEAN LENGTH(32);
```

<CASE>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the CASE type, an ordered collection of selections for the declaration of data.

Syntax

```
<CASE> ::=
  CASE
  <CASE attributes list>
  <BEGIN statement>
    <WHEN statement>...
    [<OTHERWISE statement>]
  <END statement>
```

```
<CASE attributes list> ::=
  {
    <CASE defaulted attributes list>
    <CASE optional attributes list>
  }!
```

```
<CASE defaulted attributes list> ::=
  [<MAXALC attribute>]

<CASE optional attributes list> ::=
  {
    [<LENGTH attribute>]
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

1. The <LENGTH attribute> must be specified if the space occupied by the <CASE> exceeds the space occupied by any of the alternatives of the <CASE>.
2. You must ensure the validity of an <identifier> referenced in a <CASE statement>. For example:

   ```
   x: BINARY
   a: CASE BEGIN;
      b: WHEN x = 1 THEN d: CHAR;
      c: WHEN x = 2 THEN d: FLOAT;
      END;
   ```

   A reference to d or a.d is ambiguous—at least b.d or c.d must be used. When referring to b.d or c.d, it is the responsibility of the user and the application program calling DD&C to ensure that only valid data is referenced.

General rules

1. The <condition>s of the <WHEN statement>s of the CASE are evaluated in the order in which the <WHEN statement>s are specified within the <CASE>. When the evaluation of one of them returns TRUE, the <THEN clause> of that <WHEN statement> is selected. All subsequent <WHEN statement>s and the <OTHERWISE statement> in the CASE declaration are ignored.
2. If no <WHEN clause> evaluates to TRUE, the <OTHERWISE statement> is selected.
3. A null data declaration exists if there is no data to describe. A CASE results in a null data declaration if one of the following is selected:
   a. THEN <terminator>.
   b. OTHERWISE <terminator>.
   c. OTHERWISE <SKIP statement>.
   d. All <WHEN statement>s evaluating to FALSE and no <OTHERWISE statement>.
4. If MAXALC(TRUE) is specified, space is allocated to hold the largest variant.
5. If MAXALC(FALSE) is specified, space is allocated to hold the selected variant.
6. If MAXALC(TRUE) is specified, space for the largest variant is allocated which can be the number of bits specified in the <SKIP statement>.
7. The minimum alignment of an CASE field is byte alignment.

8. See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.

Examples

1. Declare a sequence that has a discriminant field named a and variant fields whose declaration depends on the value of a.

```
x: SEQUENCE
   BEGIN;
   a: CHAR;
   CASE BEGIN;
     c1:  WHEN a = '1'
          THEN w: CHAR LENGTH(4);
     c2:  WHEN a = '2'
          THEN z: BINARY SCALE(5) PRECISION(15);
          OTHERWISE y: FLOAT;
        END;
   END;
```

2. Declare a sequence that has two discriminant fields (a and b).

```
Y: SEQUENCE
   BEGIN;
   a: CHAR;
   b: BOOLEAN;
   CASE BEGIN;
      c1: WHEN a = '1'
          THEN c: CHAR LENGTH(4);
      c2: WHEN b = TRUE
          THEN d: BINARY SCALE(5) PRECISION(15);
          OTHERWISE e: FLOAT;
        END;
   END;
```

3. Declare a sequence that has a discriminant field named a.

   Note: There is no declaration for the first <WHEN statement>. If this first <WHEN statement> evaluates to true, c1 results in a null declaration.

```
Y: SEQUENCE
   BEGIN;
   a: CHAR;
   CASE BEGIN;
      c1: WHEN a = '1'
          THEN ;
      c2: WHEN a = '4'
          THEN d: BINARY SCALE(5) PRECISION(15);
          OTHERWISE e: FLOAT;
        END;
   END;
```

4. Declare a case with a discriminant field that is dependent on itself:

```
ValidCase: CASE
           BEGIN;
           c1: WHEN x.a = '1'
               THEN x: SEQUENCE
                       BEGIN;
                         a: CHAR;
                         b: BINARY PRECISION(15);
                       END;
           c2: WHEN y.a = '2'
               THEN y: SEQUENCE
                       BEGIN;
                         a: CHAR;
                         b: CHAR LENGTH(2);
                       END;
           END;
```

<CHAR>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the CHAR type, a string of characters.

The length of a CHAR field can be specified directly by the <LENGTH attribute> or indirectly by the <LOW attribute> and the <HIGH attribute>. The unit of measure for a CHAR field is characters; the UNITLEN attribute specifies the number of bits per character (8 or 16).

In either case, the attribute values can be specified by an integer literal or by reference to another field. If specified by an integer literal, all instances of the variable are of the same length. But if specified by reference to another field, then each instance can be of a different length. In this case, however, the maximum length of the string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length.

Characters are encoded in the string as specified by the coded character set identifier (CCSID) attribute. For each CCSID, the IBM Character Data Representation Architecture (CDRA) defines what characters can be encoded and how they are encoded. A variety of single-byte, double-byte, and mixed character sets are accommodated by CDRA.

Syntax

```
<CHAR> ::=
  CHAR
  <CHAR attributes list>
```

```
<CHAR attributes list> ::=
  {
    <CHAR defaulted attributes list>
    <CHAR optional attributes list>
  }!

<CHAR defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<JUSTIFY attribute>]
    <CHAR length attributes list>
  }!

<CHAR length attributes list> ::=
  [
    <CHAR LENGTH asterisk attributes list> |
    <CHAR LENGTH fixed attributes list> |
    <CHAR LENGTH identifier attributes list> |
    <CHAR HIGHLOW1 attributes list> |
    <CHAR HIGHLOW2 attributes list> |
    <CHAR HIGHLOW3 attributes list> |
    <CHAR HIGHLOW4 attributes list>
  ]

<CHAR LENGTH asterisk attributes list> ::=
  [
    LENGTH(*)
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR length fixed attributes list> ::=
  [
    LENGTH({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR LENGTH identifier attributes list> ::=
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR HIGHLOW1 attributes list> ::=
  [
    HIGH({1..<max CHAR> | <constant identifier>})
    LOW({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!
```

```
<CHAR HIGHLOW2 attributes list> ::=
  [
    HIGH({1..<max CHAR> | <constant identifier>})
    LOW(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR HIGHLOW3 attributes list> ::=
  [
    HIGH(<qualified identifier>)
    LOW({1..<max CHAR> | <constant identifier>})
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR HIGHLOW4 attributes list> ::=
  [
    HIGH(<qualified identifier>)
    LOW(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<max CHAR> ::=
  See Syntax Rule 2.
```

Syntax rules

1. HIGH is greater than or equal to LOW-1.
2. The LENGTH or MAXLEN of a CHAR field is specified in terms of units whose length in bits is specified by UNITLEN. The length of a CHAR field in bits is determined by (LENGTH * UNITLEN), and the greatest value that can be specified for LENGTH is determined by (<max31> / UNITLEN). The maximum length of a CHAR field in bits is determined by (MAXLEN * UNITLEN), and the greatest value that can be specified for MAXLEN is determined by (<max31> / UNITLEN).
3. If a <qualified identifier> is specified for the <LENGTH attribute>, <HIGH attribute>, or <LOW attribute>, the referenced field is BINARY, PACKED, or ZONED with SCALE(0) and COMPLEX(FALSE) and with a maximum value of <max CHAR>.
4. LENGTH(*) can be specified only for the last element of all containing <constructor>s.
5. LENGTH(*) cannot be specified for any element of an ARRAY.

General rules

1. If the <CCSID attribute> is specified, it applies only to the field being described and not to the attributes of the field being declared. The length in bits of the CHAR field is the actual field length, as defined in "Actual Field Length" on page 137. In particular, for LENGTH(*), this is the largest multiple of the UNITLEN value that fits within the data.
2. The minimum alignment of a CHAR field is byte alignment.
3. If the <LOW attribute> and <HIGH attribute> are specified, the length in bits of the CHAR field equals

   ```
   ((HIGH - LOW + 1) * UNITLEN).
   ```

Examples

1. Declare `lastname` to be a character string of length (20) that is encoded as defined by CCSID 500.

   ```
   lastname: CHAR LENGTH(20) CCSID(500);
   ```

2. Declare `stretch` to be a character string that begins at position 15 and ends at position 300.

   ```
   stretch: CHAR HIGH(300) LOW(15);
   ```

3. Declare `a` to be a character string that begins and ends at positions defined with reference fields.

   ```
   a: SEQUENCE
      BEGIN;
        b: BINARY;
        c: PACKED;
        d: CHAR HIGH(b) LOW(c) MAXLEN(100) MAXALC(FALSE);
      END;
   ```

4. The following ADL declarations are equivalent:

   ```
   CHAR LENGTH(10) UNITLEN(8);
   CHAR LOW(1) HIGH(10) UNITLEN(8);
   CHAR LOW(11) HIGH(15) UNITLEN(16);
   ```

<CHARPRE>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the CHARPRE type, a variable-length string of characters with a BINARY length prefix.

Figure 4 on page 82 shows the possible representations of CHARPRE strings.

Since each instance can be a different length, the maximum length of the CHARPRE string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length. The characters of the string are encoded as specified by the CCSID attribute.

```
Variable-length with prefix length and MAXALC(FALSE)
   previous field |lpcccccccccccccccccccccccccccc|

Variable-length with prefix length and MAXALC(TRUE)
   previous field |lpcccccccccccccccccccccccccccccnnnnnnnn| next field

LEGEND:
   ccc... specifies a string of characters
   lp     specifies a length prefix
   nnn... specifies a string of unused bytes
```

Figure 4. Variable-length character strings with length prefixes

Syntax

```
<CHARPRE> ::=
  CHARPRE
  <CHARPRE attributes list>

<CHARPRE attributes list> ::=
  {
    <CHARPRE defaulted attributes list>
    <CHARPRE optional attributes list>
  }!

<CHARPRE defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max CHARPRE> |
          <constant identifier>
        })
    ]
    [<PREBYTRVS attribute>]
    [<PRELEN attribute>]
    [<PRESIGNED attribute>]
    [UNITLEN({8 | 16 | <constant identifier>})]
  }!

<CHARPRE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

```
<max CHARPRE> ::=
  See Syntax Rule 1 on page 83.
```

Syntax rules

The <MAXLEN attribute> does not include the length of the prefix.

General rules

1. The MAXLEN of a CHARPRE field is specified in terms of units whose length in bits is specified by UNITLEN. The maximum length of a CHARPRE field in bits is determined by (MAXLEN * UNITLEN), and the greatest value that can be specified for MAXLEN is determined by (2**(PRELEN-1)-1).
2. The prefix is encoded as a BINARY number with the attributes COMPLEX(FALSE) CONSTRAINED(FALSE) RADIX(2) SCALE(0) SGNCNV(LOGICAL) FIT(ROUND) and the attributes PREBYTRVS, PRESIGNED, and PRELEN specified for the BITPRE data type. PRECISION is the maximum value allowed for the specified LENGTH and SIGNED values.
3. If the <CCSID attribute> is specified, it applies only to the value of the field being described and not to the attributes of the field being declared.
4. The value of the prefix does not include the length of the prefix.
5. If MAXALC(TRUE) is specified, the actual length in bits of the data portion of a CHARPRE field is determined by (MAXLEN * UNITLEN). If the prefix value is less than MAXLEN, trailing bytes exist after the active data.
6. If MAXALC(FALSE) is specified, the actual length in bits of the data portion of a CHARPRE field is determined by the value in the prefix times UNITLEN.
7. The minimum alignment of a CHARPRE field is byte alignment.

Examples

Declare `TextField` to be a variable-length character string with a maximum length of 32767 characters. The character string is prefixed by a length field that specifies the actual length of the character string. Only space for the actual number of characters is allocated.

```
TextField: CHARPRE MAXLEN(32767) MAXALC(FALSE);
```

<CHARSFX>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the CHARSFX type, a variable-length character string that is terminated by a suffix.

Since each instance can be a different length, the maximum length of the CHARSFX string must also be specified, along with an indication of whether storage has been allocated for the maximum length or only for the current length. The characters of the string are encoded as specified by the CCSID attribute. The suffix value depends on the CCSID attribute specified. For a single-byte or mixed-byte character string, the suffix is X'00'. For a double-byte character string, it is X'0000'.

Figure 5 shows the possible representations of CHARSFX strings.

```
Variable-length with suffix and MAXALC(FALSE)
   previous field |cccccccccccccccccccccccccccs|
   ____________________________________________|


Variable-length with suffix and MAXALC(TRUE)
   previous field |cccccccccccccccccccccccccccsnnnnnnn next field
   _______________|________________________________|______________

LEGEND:
   ccc... specifies a string of characters
   s      specifies a suffix
   nnn... specifies a string of unused bytes
```

Figure 5. Variable-length character strings with a suffix

Syntax

```
<CHARSFX> ::=
  CHARSFX
  <CHARSFX attributes list>

<CHARSFX attributes list> ::=
  {
    <CHARSFX defaulted attributes list>
    <CHARSFX optional attributes list>
  }!

<CHARSFX defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max CHARSFX> |
          <constant identifier>
        })
    ]
    [UNITLEN({8 | 16 | <constant identifier>})]
  }!

<CHARSFX optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<max CHARSFX> ::=
  See Syntax Rule 2 on page 85.
```

Syntax rules

1. The <MAXLEN attribute> specified includes the length of the suffix.
2. The MAXLEN of a CHARSFX field is specified in terms of units whose length in bits is specified by UNITLEN. The maximum length of a CHARSFX field in bits is determined by (MAXLEN * UNITLEN), and the greatest value that can be specified for MAXLEN is determined by (<max31> / UNITLEN).

General rules

1. If the <CCSID attribute> is specified, it applies only to the value of the field being described and not to the attributes of the field being declared.
2. The actual length in bits of the data portion of a CHARSFX field is determined by the number of bytes encountered prior to the suffix times 8.
3. The minimum alignment of a CHARSFX field is byte alignment.

Examples

Declare `DataLine` to be a variable-length character string with a maximum length of 100 characters.

Note: The storage allocation for this string is 101 bytes.

```
DataLine: CHARSFX MAXLEN(101);
```

<ENUMERATION>

The following shows the function, syntax, rules, and examples:

Function

Declare a set of identifiers to be associated with constant integers.

Syntax

```
<ENUMERATION> ::=
  ENUMERATION
  <ENUMERATION attributes list>
  <enumeration list>

<ENUMERATION attributes list> ::=
  {
    <ENUMERATION defaulted attributes list>
    <ENUMERATION optional attributes list>
  }!

<ENUMERATION defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [LENGTH({8 | 16 | 32 | <constant identifier>})]
    [<SGNCNV attribute>]
    [<SIGNED attribute>]
  }!
```

```
<ENUMERATION optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<enumeration list> ::=
  (<enumeration value>[, <enumeration value>]...)

<enumeration value> ::=
  <enumeration identifier>[:<signed integer>]

<enumeration identifier> ::=
  <identifier>
```

Syntax rules

1. If a <signed integer> is not specified for the first <enumeration value>, a value of 0 is assumed.
2. The enumeration values of <ENUMERATION statements> must be mutually exclusive.
3. If a <signed integer> is not specified for a subsequent <enumeration value>, a value of 1 plus the value of the previous <enumeration value> is assumed.
4. The range of enumerated values is:

Table 5. Range of enumerated values

LENGTH	SIGNED	PRECISION	MINIMUM	MAXIMUM
8	TRUE	7	-128	+127
8	FALSE	8	0	255
16	TRUE	15	-32768	+32767
16	FALSE	16	0	65535
32	TRUE	31	-2,147,483,648	+2,147,483,647
32	FALSE	32	0	2,147,483,647

General rules

1. The value of each <enumeration identifier> in the declaration is stored as an <integer literal>.
2. The ENUMERATION encoding is of the BINARY type, with COMPLEX(FALSE), CONSTRAINED(FALSE), FIT(ROUND), RADIX(2), SCALE(0), and with the attributes specified for the ENUMERATION data type. The PRECISION is the maximum value for the given LENGTH and SIGNED values.
3. The minimum alignment of an ENUMERATION field is byte alignment.

Examples

1. Declare `color` to be an enumeration consisting of red, green, blue, and yellow:

```
color: ENUMERATION(red, green, blue, yellow);

The values associated with each are:
red = 0, green = 1, blue = 2, yellow = 3
```

2. Declare `animals` to be an enumeration consisting of cat, dog, monkey, elephant where monkey is to take on the value of 4:

   ```
   animal: ENUMERATION(cat, dog, monkey:4, elephant);
   ```

   The values associated with each are: cat = 0, dog = 1, monkey = 4, elephant = 5

3. Declare `family` to be an enumeration consisting of grandmother, mother, me, daughter, and granddaughter, zeroed on me:

   ```
   family: ENUMERATION(grandmother:-2, mother, me, daughter, granddaughter);
   ```

   The values associated with each are: grandmother = -2, mother = -1, me = 0, daughter = 1, granddaughter = 2

<FLOAT>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the FLOAT type, a floating-point numeric field.

A **floating-point number** is a bit string characterized by three components: a sign, a signed exponent, and a significand. A floating-point number is represented in storage in one of the formats specified by the <FORM attribute>. Its numerical value, V, may be derived from its stored representation as follows:

```
e = C-b
V = (S*(B**e))*((-1)**s)
```

where the terms of these expressions are defined as follows:

Term	**Definition**
Sign (s)	The high-order bit in the stored representation of the number. The value of the number is considered to be positive or negative depending on whether the sign is zero or one, respectively.
Exponent (e)	The component of a floating-point number that normally signifies the integer power to which the **base** is raised in determining the value of the represented number. The exponent is not stored directly, but is first converted to a **characteristic**.
Base (B)	The number to which the exponent is applied when determining the numerical value of a floating-point number. The base used depends on the format.
Characteristic (C)	The sum of the exponent and a constant (**bias**) chosen to make the range of the stored representation of the exponent non-negative. The characteristic is stored in the bits imme-

diately following the sign. The length of the characteristic depends on the format.

Bias (b)	A constant that is added to the exponent in order to create the unsigned characteristic which is stored to represent the exponent. The bias used depends on the format.
Significand (S)	The component of a floating-point number that specifies the value to be multiplied by the base raised to the power of the exponent. The length and interpretation of the significand depends on the format.

Different formats of the FLOAT data type accommodate either binary or hexadecimal representations of floating-point numbers.

HEXADECIMAL Formats: In hexadecimal floating-point numbers, the significand consists of an implicit leading zero bit to the left of its implied binary point and a fraction field to the right. The significand is stored following the characteristic in the representation of the number.

A value that is stored in the significand can be normalized to represent it with the greatest precision possible for a given format. Normalization is done by taking the value in hexadecimal form and shifting left or right until the first digit to the right of the hexadecimal point is nonzero and all digits to the left of the hexadecimal point are zero. The exponent is reduced by the number of hexadecimal digits that were shifted left or increased by the number of hexadecimal digits that were shifted right. The result is stored in the significand.

Up to three leftmost bits of the significand of a normalized hexadecimal floating-point number may be zeros, since the nonzero test applies to the entire leftmost hexadecimal digit. Thus, the guaranteed binary precision is three less than the maximum binary precision.

There are two values that represent zero: +0 and -0. A true zero is a floating-point number with a zero sign, characteristic, and significand.

There are three formats of hexadecimal floating-point numbers:

Table 6. Hexadecimal floating-point formats

FORMAT	FORM	SIGN	CHARACTERISTIC	BIAS	SIGNIFICAND	LENGTH
single	FH32	1 bit	7 bits	64	6 hex digits	32 bits
double	FH64	1 bit	7 bits	64	14 hex digits	64 bits
extended	FH128	1 bit	7 bits	64	28 hex digits	128 bits

Single-precision hexadecimal floating-point numbers, FORM(FH32), are represented as follows:

```
┌─────┬───────────────┬────────────┐
│sign │characteristic │significand │
└─────┴───────────────┴────────────┘
0     1               8           31
```

Double-precision hexadecimal floating-point numbers, FORM(FH64), are represented as follows:

```
┌─────┬───────────────┬────────────┐
│sign │characteristic │significand │
└─────┴───────────────┴────────────┘
0     1               8           63
```

Extended-precision hexadecimal floating-point numbers, FORM(FH128), are represented as follows:

```
┌─────┬───────────────┬──────────────────────────┐
│     │high–order     │leftmost 14 hex digits of │
│sign │characteristic │28 hex digit significand  │
└─────┴───────────────┴──────────────────────────┘
0     1               8                         63
┌─────┬───────────────┬──────────────────────────┐
│     │low–order      │rightmost 14 hex digits of│
│sign │characteristic │28 hex digit significand  │
└─────┴───────────────┴──────────────────────────┘
64    65             72                        127
```

The characteristic and sign of the high-order part are the characteristic and sign of the extended floating-point number. If the high-order part is normalized, the extended number is considered normalized. When an extended floating-point number is operated on, the sign of the low-order part is set to the same as that of the high-order part, and, unless the result is a true 0, the characteristic of the low-order part is made 14 less than that of the high-order part. If the subtraction of 14 from the high-order part is less than zero, the low-order characteristic is made 128 larger than the correct value. When an extended floating-point field is initialized, the low-order part may be set to a true zero if the low-order significand is zero. The preceding guarantees that both parts of the extended floating-point field are valid long floating-point numbers and can each be used as a long floating-point field.

BINARY Formats: In binary floating-point numbers, the significand consists of an explicit or implicit integer bit to the left of its implied binary point and fraction bits to the right. The significand is stored following the characteristic in the representation of the number.

A value is *normalized* in order to represent it with the greatest precision possible for a given format. Normalization is done by taking the value in binary form and shifting left or right until a single binary one is to the left of the binary point. The exponent is reduced by the number of bits which were shifted left or increased by the number of bits which were shifted right. The resulting normalized significand is stored according to the format.

A *denormalized* value occurs when a normalized value would require an exponent value smaller than the minimum exponent for the format. In this case, the value is shifted left until the exponent equals the minimum exponent for the format. The resulting denormalized significand is stored according to the format of the number. The integer bit is zero and the stored significand may have leading zeros. The characteristic is set to zero to signal that this number is denormalized.

There are two values which represent zero: +0 and -0.

The following special values can also be expressed:

Infinity Infinity is indicated by a reserved characteristic of (2**b)-1 and the fraction bits of the significand equal to zero.

Not a Number (NaN) The value of the floating-point is indicated to be a NaN by a reserved characteristic of (2**b)-1 and the fraction bits of the significand not equal to zero.

There are four formats for binary floating-point numbers:

Table 7. Binary floating-point formats

FORMAT	FORM	SIGN	CHARACTERISTIC	BIAS	SIGNIFICAND	LENGTH	PRECISION
single	FB32	1 bit	8 bits	127	23 bits	32 bits	24 bits
double	FB64	1 bit	11 bits	1023	52 bits	64 bits	53 bits
extended	FB80	1 bit	15 bits	16383	64 bits	80 bits	64 bits
OS/2 extended	FI128	1 bit	15 bits	16383	64 bits	128 bits	64 bits

Single-precision binary floating-point numbers, FORM(FB32), are represented as follows:

```
 ┌─────┬──────────────┬────────────┐
 │sign │characteristic│significand │
 └─────┴──────────────┴────────────┘
 0     1              9           31
```

In the single format, the integer bit of the significand is implicit and not stored. The implied binary point is to the left of the first bit of the stored significand.

Double-precision binary floating-point numbers, FORM(FB64), are represented as follows:

```
 ┌─────┬──────────────┬────────────┐
 │sign │characteristic│significand │
 └─────┴──────────────┴────────────┘
 0     1             12           63
```

In the double format, the integer bit of the significand is implicit and not stored. The implied binary point is to the left of the first bit of the stored significand.

Extended-precision binary floating-point numbers, FORM(FB80), are represented as follows:

```
┌────┬──────────────┬───────────┐
│sign│characteristic│significand│
└────┴──────────────┴───────────┘
0    1              16          79
```

In the extended format, the integer bit of the significand is explicit and is the first bit in the stored significand. If the integer bit is one, the number is normalized. If the integer bit is zero, the number is denormalized. The implied binary point is to the left of the second bit of the stored significand.

OS/2 extended-precision binary floating-point numbers, FORM(FI128), are represented as follows:

```
┌────┬──────────────┬───────────┬──────┐
│sign│characteristic│significand│unused│
└────┴──────────────┴───────────┴──────┘
0    1              16          79     127
```

In the OS/2 extended format, the integer bit of the significand is explicit and is the first bit in the stored significand. If the integer bit is one, the number is normalized. If the integer bit is zero, the number is denormalized. The implied binary point is to the left of the second bit of the stored significand.

The <PRECISION attribute> specifies the maximum number of significant digits of interest in the field. When PRECISION is less than the implied length, the entire significand is still used. Thus, PRECISION does not affect the format or value of the stored floating-point field.

Syntax

```
<FLOAT> ::=
  FLOAT
  <FLOAT attributes list>

<FLOAT attributes list> ::=
  {
    <FLOAT defaulted attributes list>
    <FLOAT optional attributes list>
  }!

<FLOAT defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [<COMPLEX attribute>]
    [<FIT attribute>]
    [<FORM attribute>]
    [<RADIX attribute>]
  }!
```

```
<FLOAT optional attributes list> ::=
  {
    [<HELP attribute>]
    [<PRECISION attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None.

General rules

1. The default and maximum precision that can be specified is determined by the following table:

Table 8. <FLOAT> maximum precision

FORM	RADIX(2)	RADIX(10)
FH32	24 binary digits	6 decimal digits
FH64	56 binary digits	16 decimal digits
FH128	112 binary digits	33 decimal digits
FB32	24 binary digits	7 decimal digits
FB64	53 binary digits	15 decimal digits
FB80	64 binary digits	19 decimal digits
FI128	64 binary digits	19 decimal digits

2. PRECISION must be less than or equal to the maximum precision of the form selected.
3. The minimum alignment of a FLOAT field is byte alignment.
4. If COMPLEX(TRUE) is specified, the length of the field is double the length of the same field specified with COMPLEX(FALSE).

Examples

1. Declare `ElectronCharge` to be a binary floating-point number represented with a length of 64 bits.

   ```
   ElectronCharge: FLOAT FORM(FB64);
   ```

2. A normalized single-precision hexadecimal FLOAT field:

```
x : FLOAT FORM(FH32);

The stored value of x is X'41600000'
6.0 is stored as a normalized FLOAT number.
s = sign = 0
C = characteristic = x'41' = 65
e = exponent       = C - bias = 65 - 64 = 1
S = significand    = x'600000' = 0.375
x = (-1)**0 * 0.375 * 16**1 = 6.0
```

3. A normalized single-precision binary FLOAT field:

```
x : FLOAT FORM(FB32);

The stored value of x is x'40C00000' = 6.0
s = sign = 0
C = characteristic = B'10000001' = X'81' = 129
e = exponent       = C - bias = 129 - 127 = 2
since 0 < C < 255, the number is normalized and
S = significand = 1 + B'10000000000000000000000' = 1 + 0.5 = 1.5
x = (-1)**0 * 1.5 * 2**2 = 1 * 1.5 * 4 = 6.0
```

<PACKED>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the PACKED type, a packed decimal numeric field.

1. A packed decimal field is a sequence of 4-bit strings representing decimal digits (0-9) followed by an optional 4-bit sign position. If needed, the field is extended on the left to a multiple of 8 bits length.
2. If a sign position exists, SIGNED(TRUE) is specified and the representation of the PACKED field is defined by the <SGNLOC attribute> with the <SGNPLS attribute> and <SGNMNS attribute> or the <SGNUNS attribute>.
3. If a sign position does not exist, SIGNED(FALSE) is specified and the whole field represents the number.
4. The maximum number of significant digits in a PACKED field is specified by the <PRECISION attribute>, but if PRECISION is even, the number of digits stored in a PACKED field is one greater than PRECISION.
5. Numbers are stored in PACKED fields as integers. The actual value depends on the <SCALE attribute>, as defined by:

   ```
   Actual_Value = Stored_Integer_Value*(10**(-SCALE))
   ```

   If the scaling factor is negative, then the actual value is greater than the stored value. For example, if decimal 123 is stored and it has a scaling factor of -3, then the actual value is 123000. If the scaling factor is positive, then the actual value is equal to or less than the stored value. For example, if decimal 123 is stored and it has a scaling factor of 3, then .123 is the actual value.

Syntax

```
<PACKED> ::=
  PACKED
  <PACKED attributes list>

<PACKED attributes list> ::=
  {
    <PACKED defaulted attributes list>
    <PACKED optional attributes list>
  }!

<PACKED defaulted attributes list> ::=
  {
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [PRECISION({1..31 | <constant identifier>})]
    [SCALE({-128..127 | <constant identifier>})]
    <PACKED SIGNED attributes list>
  }!

<PACKED SIGNED attributes list> ::=
  [
    SIGNED(TRUE)
    <PACKED SGNLOC attributes list>
  ]! |
  [SIGNED(FALSE)]

<PACKED SGNLOC attributes list> ::=
  SGNLOC(DGTLSTBYT)
  {
    [
      <SGNMNS attribute>
      <SGNPLS attribute>
    ]! |
    [<SGNUNS attribute>]
  }

<PACKED optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None.

General rules

1. The length in bits of a PACKED field is specified as follows:

   a. For SGNLOC(DGTLSTBYT) and an odd PRECISION, the length of the field in bits is equal to (PRECISION+1)*4.

b. For SGNLOC(DGTLSTBYT) and an even PRECISION, the length of the field in bits equals (PRECISION+2)*4.

c. If SIGNED(FALSE) is specified and the PRECISION is odd, the length of the field in bits equals (PRECISION +1)*4.

d. If SIGNED(FALSE) is specified and the PRECISION is even, the length of the field in bits equals PRECISION*4.

2. The minimum alignment of a PACKED field is byte alignment.

3. If COMPLEX(TRUE) is specified, the length of the field is double the length of the same field specified with COMPLEX(FALSE).

Examples

The following legend applies to the examples that follow:

p	Precision digit
s	Sign digit
o	Overflow digit
f	Fraction digit

1. A packed decimal field with scale, precision, and sign

```
X: PACKED PRECISION(7) SCALE(2);
byte    0       1       2       3
    ┌───┬───┬───┬───┬───┬───┬───┬───┐
    │   │   │   │   │   │ f │ f │   │
    │ p │ p │ p │ p │ p │ p │ p │ s │
    └───┴───┴───┴───┴───┴───┴───┴───┘
```

2. A packed decimal field with negative scaling:

```
X: PACKED PRECISION(4) SCALE(-5);
byte    0       1       2        Number Stored =
    ┌───┬───┬───┬───┬───┬───┐        ┌───┬───┬───┬───┬───┬───┐
    │ o │ p │ p │ p │ p │ s │        │ 0 │ 0 │ 0 │ 1 │ 7 │ C │
    └───┴───┴───┴───┴───┴───┘        └───┴───┴───┴───┴───┴───┘

                                 Value = 17 * 100000 = 1700000
```

3. A packed decimal field with positive scaling:

```
X: PACKED PRECISION(4) SCALE(5);
byte    0       1       2        Number Stored =
    ┌───┬───┬───┬───┬───┬───┐        ┌───┬───┬───┬───┬───┬───┐
    │ o │ p │ p │ p │ p │ s │        │ 0 │ 0 │ 0 │ 1 │ 7 │ C │
    └───┴───┴───┴───┴───┴───┘        └───┴───┴───┴───┴───┴───┘

                                 Value = 17 * 0.00001 = 0.00017
```

<SEQUENCE>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the SEQUENCE type, an ordered collection of fields and constructors.

Syntax

```
<SEQUENCE> ::=
  SEQUENCE
  <SEQUENCE attributes list>
  <BEGIN statement>
    {<data declaration statement> | <SKIP statement>}...
  <END statement>

<SEQUENCE attributes list> ::=
  <SEQUENCE optional attributes list>

<SEQUENCE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None.

General rules

1. The elements of a SEQUENCE are mapped contiguously onto a bit string.
2. Unused bits required for proper alignment of certain data types are specified by the <SKIP statement>.
3. The minimum alignment of an SEQUENCE field is byte alignment.
4. See “Identifiers” on page 7 for the rules that apply to the uniqueness of identifiers.

Examples

1. Declare `name` to be a sequence with the components: `lastname`, `firstname`, and `initial` .

   ```
   name: SEQUENCE
         BEGIN;
            lastname: CHAR LENGTH(12);
            firstname: CHAR LENGTH(12);
            initial: CHAR;
         END;
   ```

   The qualified names of the fields in `name` are:

   ```
   name.lastname   name.firstname   name.initial
   ```

2. Declare address to be a sequence with the components: street, city, state, and zip.

   ```
   address: SEQUENCE
            BEGIN;
               street: CHAR LENGTH(30);
               city: CHAR LENGTH(15);
               state: CHAR LENGTH(2);
               zip: CHAR LENGTH(10);
            END;
   ```

   The qualified names of the fields in address are:

   ```
   address.street   address.city   address.state   address.zip
   ```

<subtype instance>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of a user-defined subtype.

Syntax

```
 <subtype instance> ::=
   <subtype identifier>
   <subtype instance attributes list>

 <subtype instance attributes list> ::=
   {
     <ARRAY attributes list> |
     <ASIS attributes list> |
     <BIT attributes list> |
     <BITPRE attributes list> |
     <BOOLEAN attributes list> |
     <CASE attributes list> |
     <CHAR attributes list> |
     <CHARPRE attributes list> |
     <CHARSFX attributes list> |
     <BINARY attributes list> |
     <ENUMERATION attributes list> |
     <FLOAT attributes list> |
     <PACKED attributes list> |
     <SEQUENCE attributes list> |
     <ZONED attributes list>
   }
```

Syntax rules

The attribute list specified is of the ADL type specified in the <SUBTYPE statement> associated with the <subtype instance> either directly or through inheritance.

General rules

1. The attributes specified override the values of the attributes with the same key words specified in the <SUBTYPE statement> identified by the <subtype identifier>.
2. Any attributes not specified in the <subtype instance> declaration are inherited from the <SUBTYPE statement>.
3. If an <identifier> is specified for the <subtype instance> <data declaration statement>, the <identifier> replaces the <identifier> specified in the <SUBTYPE statement> in the fully-qualified names.

Examples

1. Declare a sequence that includes an instance of `GameBoard` :

```
piece: SUBTYPE OF ENUMERATION LENGTH(16)
           (king, queen, knight, bishop, rook, pawn);
GameBoard: SUBTYPE OF ARRAY DMNLST(DMNSIZE(8), DMNSIZE(8))
                      OF piece;
game: SEQUENCE
      NOTE('This sequence defines the state of '
           'a chess game that is being played.')
      BEGIN;
         move: ENUMERATION(white,black);
         incheck: BOOLEAN;
         checkmate: BOOLEAN;
         castled: ARRAY DMNLST(DMNSIZE(2))
                  OF BOOLEAN;
         ChessBoard: GameBoard;
      END;
```

2. Declare `A` to be a subtype of the array type with two dimensions.

```
A: SUBTYPE OF ARRAY  DMNLST(DMNSIZE(4), DMNSIZE(7))
                     OF CHAR LENGTH(4);
```

```
Declare B to be an
instance of A with dimensions of 3 and 5:
B: A DMNLST(DMNSIZE(3), DMNSIZE(5));
```

Note: The whole DMNLST attribute can be overridden, but not the description of individual dimensions.

<ZONED>

The following shows the function, syntax, rules, and examples:

Function

Declare an instance of the ZONED type, a zoned decimal numeric field.

1. A zoned decimal field is a sequence of bytes, each of which contains a representation of a decimal digit (0 to 9) in its rightmost 4 bits, and each of which normally contains a zone encoding in its leftmost 4 bits. The zone encoding is typically that which would cause the byte to print as a numeric character. For example, the digit 9 could be encoded as `X'F9'`, which represents the value 9 and would print as the character “9” in CCSID(500).

   If the field is signed, one of several variations on the above occurs:

   - The sign can consist of a separate byte, which appears at the left end or right end of the numeric field, depending on the value specified for the <SGNLOC attribute>. The sign is the character plus (+) or minus (-), as appropriate.
   - The sign can be encoded in place of the zone portion of the leftmost or rightmost digit, also depending on the value specified for the <SGNLOC attribute>.

2. The encoding of all zones is specified by the <ZONENC attribute>.
3. If a sign position exists, SIGNED(TRUE) is specified and its representation is defined by the <SGNLOC attribute>, <SGNPLS attribute>, and the <SGNMNS attribute>.
4. If a sign position does not exist, SIGNED(FALSE) is specified and the whole field represents the number.
5. The maximum number of significant digits in a ZONED field is specified by the <PRECISION attribute>.
6. Numbers are stored in ZONED fields as integers. The actual value depends on the <SCALE attribute>, as defined by:

   ```
   Actual_Value = Stored_Integer_Value*(10**(-SCALE))
   ```

   If the scaling factor is negative, then the actual value is greater than the stored value. For example, if decimal 123 is stored and it has a scaling factor of -3, then the actual value is 123000. If the scaling factor is positive, then the actual value is equal or less than the stored value. For example, if decimal 123 is stored and it has a scaling factor of 3, then .123 is the actual value.

Syntax

```
<ZONED> ::=
  ZONED
  <ZONED attributes list>
```

```
<ZONED attributes list> ::=
  {
    <ZONED defaulted attributes list>
    <ZONED optional attributes list>
  }!
<ZONED defaulted attributes list> ::=
  {
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [PRECISION({1..31 | <constant identifier>})]
    [SCALE({-128..127 | <constant identifier>})]
    <ZONED SIGNED attributes list>
    [<ZONENC attribute>]
  }!
<ZONED SIGNED attributes list> ::=
  [
    SIGNED(TRUE)
    <ZONED SGNLOC attributes list>
  ]! |
  [SIGNED(FALSE)]
<ZONED SGNLOC attributes list> ::=
  [
    SGNLOC({ZONLSTBYT | ZONFRSBYT})
    <SGNMNS attribute>
    <SGNPLS attribute>
  ]! |
  [
    SGNLOC({LSTBYT | FRSBYT})
    <CCSID attribute>
  ]!
<ZONED optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

Syntax rules

None.

General rules

1. The length of a ZONED field in bits is:

   a. PRECISION*8 if SIGNED(FALSE) is specified.

   b. PRECISION*8 if SGNLOC(ZONFRSBYT) or SGNLOC(ZONLSTBYT) is specified.

   c. (PRECISION*8)+8 if SGNLOC(FRSBYT) or SGNLOC(LSTBYT) is specified.

2. The minimum alignment of a ZONED field is byte alignment.
3. The <CCSID attribute> applies to the sign character only.
4. If COMPLEX(TRUE) is specified, the length of the field is double the length of the same field specified with COMPLEX(FALSE).

Examples

The following legend applies to the examples below:

```
Legend:
  p = precision nibble
  z = zone nibble
  s = sign nibble
  f = fraction nibble
```

1. ZONED with scale, precision, and sign

```
X: ZONED PRECISION(4) SCALE(2);
byte    0       1       2       3
      |   |   |   |   |   f   |   f |
      | z | p | z | p | z | p | s | p |
      └───┴───┴───┴───┴───┴───┴───┴───┘
```

2. ZONED with negative scale:

```
X: ZONED PRECISION(3) SCALE(-5);
byte    0       1       2        Number Stored =
      |   |   |   |   |   |           |   |   |   |   |   |   |
      | z | p | z | p | s | p |         | F | 0 | F | 1 | C | 7 |
      └───┴───┴───┴───┴───┴───┘         └───┴───┴───┴───┴───┴───┘

                                 Value = 17 * 100000 = 1700000
```

3. ZONED with scale greater than precision

```
X: ZONED PRECISION(3) SCALE(5);
byte    0       1       2        Number Stored =
      |   |   |   |   |   |           |   |   |   |   |   |   |
      | z | p | z | p | s | p |         | F | 0 | F | 1 | C | 7 |
      └───┴───┴───┴───┴───┴───┘         └───┴───┴───┴───┴───┴───┘

                                 Value = 17 * 0.00001 = 0.00017
```

4. A ZONED number of precision 3 and scale 0: The value of the number is positive 123:

SIGNED	SGNLOC	SGNMNS	SGNPLS	ZONENC	Hex Encoding	Equivalent Character Literal
FALSE				X'F'	X'F1F2F3'	'123'
TRUE	ZONLSTBYT	X'D'	X'C'	X'F'	X'F1F2C3'	'12C'
TRUE	ZONFRSBYT	X'D'	X'C'	X'F'	X'C1F2F3'	'A23'
TRUE	LSTBYT			X'F'	X'F1F2F34E'	'123+'
TRUE	FRSBYT			X'F'	X'4EF1F2F3'	'+123'

Chapter 5. Attributes

This chapter describes, in alphabetical order, the attributes that can be specified for ADL data types.

<BLNENC attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify how a BOOLEAN field is encoded.

Syntax

```
 <BLNENC attribute> ::=
   BLNENC ({<constant identifier> | <positive integer> })
```

Syntax rules

The attribute value is the value of the ADL constant LSTBIT.

General rules

LSTBIT

True - Last bit on
False - Last bit off

```
    |                  |
    |i|...|i|i|i|i|x|
    |_|___|_|_|_|_|_|
```

```
Legend:
  x = boolean encoding
  i = ignored bits
```

Examples

None.

<BYTRVS attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether a field is encoded in byte reversed order.

When data is byte reversed, each byte is moved to position ((M+1)-N) in the data representation, where

- N is the original position of a given byte.
- M is the total length of the data representation in bytes.
- Bytes are numbered from index 1.

Syntax

```
<BYTRVS attribute> ::=
  BYTRVS ({<boolean literal> | <constant identifier>})
```

Syntax rules

If BYTRVS(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.

General rules

1. BYTRVS(TRUE) must not be specified for a field less than 8 bits in length.
2. For a field of 8 bits in length, BYTRVS(TRUE) does not affect the representation of the field.
3. BYTRVS(TRUE) must not be specified for a field that is not a multiple of 8 bits.

Examples

1. If the 16-bit value X'1234' is stored in byte-reversed format at a location called NUM, its format is:

```
┌──┬──┐
│34│12│
└──┴──┘
 ↑  ↑
 │  └─ NUM plus 1
 └─ NUM
```

2. If the 32-bit value X'12345678' is stored in byte-reversed format at a location called NUM, its format is:

```
┌──┬──┬──┬──┐
│78│56│34│12│
└──┴──┴──┴──┘
 ↑  ↑  ↑  ↑
 │  │  │  └─ NUM plus 3
 │  │  └─ NUM plus 2
 │  └─ NUM plus 1
 └─ NUM
```

<CCSID attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the coded character set identifier (CCSID) of a field, as defined by the IBM Character Data Representation Architecture (CDRA).

Syntax

```
<CCSID attribute> ::=
  CCSID
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values for the <CCSID attribute> is 0 to 65535.
2. The hierarchy of CCSID attributes is as follows (low to high):
   - A <data declaration statement>
   - If the <data declaration statement> is a subtype instance, the referenced subtype declaration. This can be a chain of subtype declarations if the first subtype declaration is itself a subtype instance.
   - A PLAN parameter specification
   - A <DEFAULT statement> of the data type.
   - The system CCSID, during runtime of the Conversion Plan Executor component.

   If, according to this hierarchy, the first CCSID attribute found is in the <data declaration statement> of either the data type itself or a subtype instance and is equal to zero, then the remaining CCSID attributes found in the subtype instance chain of the data type are not evaluated. Starting with the PLAN parameter specification, the remaining hierarchy of CCSID attributes remains unchanged.
3. If CCSID(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> with a value that is in the range of the valid values of the attribute.
4. If CCSID(<qualified identifier>) is specified, the <identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).

General rules

1. If CCSID(0) is specified for a lower level entity in the hierarchy, the CCSID attribute is inherited from the next higher level of the hierarchy.
2. If CCSID(65535) is specified, the encoding of a string is undefined.

Examples

Declare `poem` to be a character string in CCSID 500:

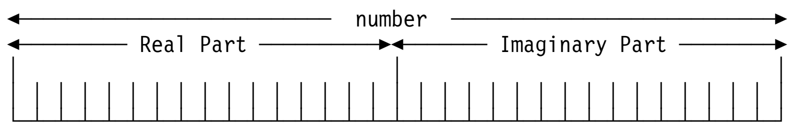
```
poem: CHAR LENGTH(40) CCSID(500);
```

<COMPLEX attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether a numeric field consists of two adjacent fields with the same attributes, with the first field representing the real component of the complex number, and the second field representing the coefficient of the imaginary component of the complex number.



Syntax

```
 <COMPLEX attribute> ::=
   COMPLEX
   ({<boolean literal> | <constant identifier>})
```

Syntax rules

If COMPLEX(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.

General rules

If COMPLEX(TRUE) is specified, all attributes of the number (such as PRECISION and LENGTH) apply to both the Real and Imaginary parts of the complex number separately.

Examples

Declare `number` to be a fixed binary field that is complex.

```
number: BINARY PRECISION(15) COMPLEX(TRUE);
```

<CONSTRAINED attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether the values that can be assigned to a field must be constrained to the range implied by the <RADIX attribute>, <SCALE attribute>, and <PRECISION attribute> of the field.

Syntax

```
 <CONSTRAINED attribute> ::=
   CONSTRAINED
   ({<boolean literal> | <constant identifier>})
```

Syntax rules

1. If CONSTRAINED(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.
2. If CONSTRAINED(TRUE) is specified, the field value must be within the range implied by the <RADIX attribute>, <PRECISION attribute>, and the <SCALE attribute>.
3. IF CONSTRAINED(FALSE) is specified, then any value that can be represented by the field is valid, and all other values are made to fit the field. Truncation of the most significant digits, rounding or truncation of the least significant digits, or both, can occur.

General rules

None.

Examples

1. Declare a fixed binary field to be constrained.

   ```
   number: BINARY PRECISION(2) CONSTRAINED(TRUE) RADIX(10);
   ```

   The values of the number are constrained to the range -99 to 99.

2. Declare a constrained field:

   ```
   X: BINARY RADIX(10) LENGTH(16) PRECISION(3) CONSTRAINED(TRUE);
   ```

   only values in the range -999 to +999 can be assigned to the field, regardless of the fact that a signed binary field 16 bits in length can actually represent values in the range -32768 to +32767

3. Declare a constrained field:

   ```
   X: PACKED PRECISION(2) CONSTRAINED(TRUE);
   ```

   only values in the range -99 to +99 can be assigned to the field, regardless of the fact that storage is available for an additional significant digit.

<DMNHIGH attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the highest subscript value for a dimension of an array.

Syntax

```
<DMNHIGH attribute> ::=
  DMNHIGH
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values for the <DMNHIGH attribute> is <min31> to <max31>.
2. If DMNHIGH(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.
3. If DMNHIGH(<qualified identifier>) is specified, the <qualified identifier> must name a field from which the value of the attribute is taken. The value must be greater than zero and one of the following fields:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).

General rules

None.

Examples

Refer to the examples under "<ARRAY>" on page 55.

<DMNLOW attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the lowest subscript value for a dimension of an array.

Syntax

```
<DMNLOW attribute> ::=
  DMNLOW
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values for the <DMNLOW attribute> is <min31> to <max31>.
2. If DMNLOW(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.
3. If DMNLOW(<qualified identifier>) is specified, the <identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).

General rules
None.

Examples
Refer to the examples under “<ARRAY>” on page 55.

<DMNLST attribute>

The following shows the function, syntax, rules, and examples:

Function
Specify the attributes of each dimension of an array.

Syntax
```
<DMNLST attribute> ::=
  DMNLST (<dimension> [{, <dimension>}...])

<dimension> ::=
  {
    [<DMNLOW attribute>]
    {
      <DMNHIGH attribute> |
      <DMNSIZE attribute>
    }
    [<DMNMAX attribute>]
  }!
```

Syntax rules
1. The <DMNMAX attribute> must be specified if either the <DMNSIZE attribute> is specified and is variable, or both the <DMNLOW attribute> and <DMNHIGH attribute> are specified and at least one of them is variable.
2. The <DMNHIGH attribute> must be greater than or equal to the <DMNLOW attribute>.
3. The number of elements in an ARRAY, implied by DMNHIGH and DMNLOW, must be less than or equal to DMNMAX.
4. The number of elements in an ARRAY, specified by DMNSIZE, must be less than or equal to DMNMAX.
5. The value of (DMNHIGH - DMNLOW + 1) must be less than or equal to <max31>.

General rules
None.

Examples
Refer to the examples under “<ARRAY>” on page 55.

<DMNMAX attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the maximum number of elements in a dimension of an array.

Syntax

```
<DMNMAX attribute> ::=
  DMNMAX({<positive integer> | <constant identifier>})
```

Syntax rules

1. The range of valid values is 1 to <max31>.
2. If DMNMAX(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.

General rules

None.

Examples

Refer to the examples under "<ARRAY>" on page 55.

<DMNSIZE attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the number of elements in a dimension of an array.

Syntax

```
<DMNSIZE attribute> ::=
  DMNSIZE
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values is 0 to <max31>.
2. If DMNSIZE(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.

3. If a DMNSIZE(<qualified identifier>) is specified, the <identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).

General rules
None.

Examples
Refer to the examples under "<ARRAY>" on page 55.

<FIT attribute>

The following shows the function, syntax, rules, and examples:

Function
Specify how to assign a numeric value into a target field.

Syntax
```
<FIT attribute> ::=
  FIT ({<constant identifier> | <positive integer> })
```

Syntax rules
The attribute value must be the value of the ADL constant ROUND, TRUNCATE, or EXACT:

1. If FIT(ROUND) is specified, the least significant binary or decimal digits of the value are rounded to fit.
2. If FIT(TRUNCATE) is specified, the least significant binary or decimal digits of the value are truncated to fit.
3. If FIT(EXACT) is specified, there is no loss of the least significant binary or decimal digits.

General rules
None.

Examples
Declare a 32-bit binary number which should be truncated to fit.

```
x: BINARY FIT(TRUNCATE);
```

<FORM attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the form of a floating point number. See "<FLOAT>" on page 87 for a description of these formats.

Syntax

```
 <FORM attribute> ::=
   FORM ({<constant identifier> | <positive integer> })
```

Syntax rules

The attribute value must be the value of the ADL constant FB32, FB64, FB80, FH32, FH64, FH128 or FI128.

General rules

None.

Examples

Declare a 32-bit binary floating-point number.

```
x: FLOAT FORM(FB32);
```

<HELP attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify an extended description of an entity for presentation to end users.

Syntax

```
 <HELP attribute> ::=
   HELP
     ({
       <character literal> |
       <encoded hex literal> |
       <constant identifier>
     })
```

Syntax rules

1. The maximum length of HELP is <max15> bytes.
2. If HELP(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <character literal> or <encoded hex literal>.

General rules

None.

Examples

None.

<HIGH attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the integer value that is associated with the last position of a character string.

Syntax

```
<HIGH attribute> ::=
  HIGH
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values is 1 to <max31>.
2. If HIGH(<qualified identifier>) is specified, the <identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).
3. If HIGH(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.

General rules

None.

Examples

See the examples under "<CHAR>" on page 78.

<JUSTIFY attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether a character string assigned to a CHAR field is to be padded or truncated on the left or on the right.

Syntax

```
<JUSTIFY attribute> ::=
  JUSTIFY ({<constant identifier> | <positive integer> })
```

Syntax rules

1. The attribute value must be the value of the ADL constant LEFT or RIGHT.
2. If JUSTIFY(LEFT) is specified, padding or truncation occurs on the right.
3. If JUSTIFY(RIGHT) is specified, padding or truncation occurs on the left.

General rules

None.

Examples

```
Source                      Target
 Length  Content     Length  JUSTIFY(LEFT)   JUSTIFY(RIGHT)
   3     'ABC'         4      'ABC '            ' ABC'
   3     'AB '         4      'AB  '            ' AB '
   3     ' BC'         4      ' BC '            '  BC'
   4     'ABCD'        4      'ABCD'            'ABCD'
   4     ' BCD'        4      ' BCD'            ' BCD'
   4     'ABC '        4      'ABC '            'ABC '
   5     'ABCDE'       4      'ABCD'            'BCDE'
   5     ' BCDE'       4      ' BCD'            'BCDE'
   5     'ABCD '       4      'ABCD'            'BCD '
```

<LENGTH attribute>

The following shows the function, syntax, rules, and examples:

Function

If the <LENGTH attribute> is used with data types or default statements, it specifies the maximum number of units occupied by the field. The unit of measurement varies according to the field being described.

If the <LENGTH attribute> is used with input plan parameters, it specifies the actual buffer size passed to the Conversion Plan Executor component.

Syntax

```
<LENGTH attribute> ::=
  LENGTH
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier> |
      *
    })
```

Syntax rules

1. The range of valid values for the <LENGTH attribute> depends on the ADL type being described.
2. The length in bits of each unit of an ASIS, CHAR, CHARPRE, or CHARSFX field is specified by the UNITLEN attribute.
3. The length in bits of each unit of a BINARY, BIT, FLOAT, BOOLEAN, or ENUMERATION field is the total number of bits occupied by the representation of the field.
4. If the <LENGTH attribute> is specified with an input plan parameter, the length is specified in bytes.
5. If LENGTH(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid lengths of the data type.
6. If LENGTH(<qualified identifier>) is specified, the <identifier> must name a variable from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).
7. LENGTH(*), LENGTH(-1), or LENGTH with a <constant identifier> having the value -1 can be specified only in an ASIS, BIT, or CHAR field, where the length is a multiple of the UNITLEN value from the position of the field in the data to the end of the data.

General rules

None.

Examples

None.

<LOW attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the integer value that is associated with the first position of a character string.

Syntax

```
<LOW attribute> ::=
  LOW
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values is 1 to <max31>.
2. If LOW(<qualified identifier>) is specified, the <identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).
3. If LOW(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.

General rules

None.

Examples

See the examples under “<CHAR>” on page 78.

<MAXALC attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether space for the maximum number of units occupied by a field is allocated.

Syntax

```
<MAXALC attribute> ::=
  MAXALC ({<boolean literal> | <constant identifier>})
```

Syntax rules

1. If MAXALC(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.
2. If MAXALC(TRUE) is specified, subsequent data may be aligned on the proper boundary of a field specified by the field, by using the <SKIP statement> following the data item.

General rules

Data declaration statements that are contained within a constructor or SUBTYPE statement can only contain one variable-length field, which must be positioned at the end of the constructor or SUBTYPE statement. Therefore, MAXALC(FALSE) is allowed for this last field and all constructors that include this field. For all other fields, MAXALC(TRUE) is required.

Examples

None.

<MAXLEN attribute>

The following shows the function, syntax, rules, and examples:

Function

If the <MAXLEN attribute> is used with data types or default statements, it specifies the maximum number of units occupied by the field. The unit of measurement varies according to the type of field being described.

If the <MAXLEN attribute> is used with output plan parameters, it specifies the actual buffer size passed to the Conversion Plan Executor component.

Syntax

```
<MAXLEN attribute> ::=
  MAXLEN
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })
```

Syntax rules

1. The range of valid values for the <MAXLEN attribute> depends on the ADL type being described.
2. The length in bits of each unit of an ASIS, CHAR, CHARPRE, or CHARSFX field is specified by the UNITLEN attribute.
3. The unit of measurement for BIT and BITPRE fields is bits.
4. If the <MAXLEN attribute> is specified with an output plan parameter, the length is specified in bytes.

5. If MAXLEN(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.
6. The <MAXLEN attribute> does not include the prefix for CHARPRE or BITPRE fields.
7. The <MAXLEN attribute> includes the suffix for CHARSFX fields.
8. If MAXLEN(<qualified identifier>) is specified, the <qualified identifier> must name a field from which the value of the attribute is taken. The value must be one of the following:
   - A BINARY field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A PACKED field with the attributes SCALE(0) and COMPLEX(FALSE).
   - A ZONED field with the attributes SCALE(0) and COMPLEX(FALSE).

General rules
None.

Examples
None.

<NOTE attribute>

The following shows the function, syntax, rules, and examples:

Function
Specify programming commentary about an ADL entity.

Syntax
```
<NOTE attribute> ::=
  NOTE
    ({
      <character literal> |
      <encoded hex literal> |
      <constant identifier>
    })
```

Syntax rules
1. The maximum length of the <NOTE attribute> is <max15> bytes.
2. If NOTE(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <character literal> or an <encoded hex literal>.

General rules
None.

Examples

None.

<PREBYTRVS attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether the length prefix of a CHARPRE or BITPRE field is byte reversed in memory.

See “<BYTRVS attribute>” on page 103.

Syntax

```
<PREBYTRVS attribute> ::=
  PREBYTRVS ({<boolean literal> | <constant identifier>})
```

Syntax rules

If PREBYTRVS(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.

General rules

None.

Examples

None.

<PRECISION attribute>

The following shows the function, syntax, rules, and examples:

Function

For BINARY, PACKED, and ZONED fields, specify the maximum number of significant binary or decimal digits of interest in a field.

For FLOAT fields, specify the maximum number of significant digits of interest in a field. The <PRECISION attribute> does not affect the form or value of the stored floating-point number.

Syntax

```
<PRECISION attribute> ::=
  PRECISION
  ({<positive integer> | <constant identifier>})
```

Syntax rules

1. The <PRECISION attribute> is specified in:

    - bits for BINARY and FLOAT fields for which RADIX(2) is specified or implied.
    - decimal digits for BINARY, FLOAT, PACKED, and ZONED fields for which RADIX(10) is specified or implied.
2. If PRECISION(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of valid value of the attribute for the data type.

General rules

None.

Examples

1. Declare `number` to be an unsigned ZONED field with a precision of 6 significant digits:

   ```
   number: ZONED PRECISION(6) SIGNED(FALSE);
   ```

2. Declare `electron` to be a floating-point binary field represented with a precision of 52 significant bits:

   ```
   electron: FLOAT FORM(FB64) PRECISION(52);
   ```

3. Declare `population` to be an unsigned BINARY field represented with a precision of 32 significant bits:

   ```
   population: BINARY SIGNED(FALSE) PRECISION(32);
   ```

4. Declare `weight` to be a PACKED field represented in 5 significant decimal digits.

   ```
   weight: PACKED PRECISION(5);
   ```

<PRELEN attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the length in bits of the BINARY prefix that specifies the length of a CHARPRE or BITPRE field.

Syntax

```
<PRELEN attribute> ::=
  PRELEN ({8 | 16 | 32 | <constant identifier>})
```

Syntax rules

If PRELEN(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is 8, 16, or 32.

General rules

None.

Examples

None.

<PRESIGNED attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether the length prefix of a BITPRE or CHARPRE field includes a sign position in its representation.

Syntax

```
<PRESIGNED attribute> ::=
  PRESIGNED ({<boolean literal> | <constant identifier>})
```

Syntax rules

1. If PRESIGNED(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.
2. If PRESIGNED(TRUE) is specified, a position for the sign exists within the representation.
3. If PRESIGNED(FALSE) is specified, a position for the sign does NOT exist within the representation.

General rules

None.

Examples

None.

<RADIX attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the number system base assumed by the <SCALE attribute> and <PRECISION attribute> for a BINARY field. The <RADIX attribute>, together with the <PRECISION attribute> and the <SCALE attribute> determines the range of values that can be stored for a field.

Specify the number system base assumed by the <PRECISION attribute> of a FLOAT field.

Syntax

```
<RADIX attribute> ::=
  RADIX ({2 | 10 | <constant identifier>})
```

Syntax rules
If RADIX(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is 2 or 10.

General rules
None.

Examples
None.

<SCALE attribute>

The following shows the function, syntax, rules, and examples:

Function
Specify the scaling factor of a fixed-point numeric field.

The <SCALE attribute> along with the <RADIX attribute> determines the scaling factor to use to find the actual value of a number. The scaling factor is:

```
RADIX**-SCALE
```

Note: <PACKED> and <ZONED> fields are implicitly radix 10 numbers.

Syntax
```
 <SCALE attribute> ::=
   SCALE({<signed integer> | <constant identifier>})
```

Syntax rules
1. The <SCALE attribute> is specified in:
   - bits for BINARY fields for which RADIX(2) is specified or implied.
   - decimal digits for BINARY, FLOAT, PACKED, and ZONED fields for which RADIX(10) is specified or implied.
2. The range of valid values for the <SCALE attribute> is <min7> to <max7>.
3. If SCALE(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range <min7> to <max7>.

General rules
None.

Examples
None.

<SGNCNV attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify how the sign of a BINARY field or an ENUMERATION field is to be determined.

Syntax

```
<SGNCNV attribute> ::=
  SGNCNV ({<constant identifier> | <positive integer> })
```

Syntax rules

The attribute value is the value of the ADL constant ALGEBRAIC or LOGICAL.

General rules

1. If the source field and the target field are both signed, then ignore the target field's <SGNCNV attribute>, and set the sign of the target field to the same sign as the source field.
2. If the source field and the target field are both unsigned, then ignore the target field's <SGNCNV attribute>.
3. If the source field is signed and the target field is unsigned, then:
   a. If SGNCNV(ALGEBRAIC) is specified for the target field, then
      - Assign the absolute value of the source field to the target field.
      - If the source field is negative, exception ***12—Assignment of negative value to unsigned field*** occurs.

   b. If SGNCNV(LOGICAL) is specified for the target field, then
      1) If the target field is the same length as the source field, do a bit copy of the source field to the target field.
      2) If the target field is longer than the source field, do a bit copy of the source field to the low-order portion of the target field and replicate the sign bit of the source in the remaining high-order bits of the target field.
      3) If the target field is shorter than the source field, bit copy the low order bits to the target field.
4. If the source field is unsigned and the target field is signed, then:
   a. If SGNCNV(ALGEBRAIC) is specified for the target field, assume that the source field is positive and set the sign of the target field to positive.

   b. If SGNCNV(LOGICAL) is specified for the target field, then
      1) If target field is the same length as source field, do a bit copy of target to source (that is, preserve the bit pattern).

2) If target field is longer than the source field, extend the source field with zeros to the length of the target field, and then perform a bit copy to the target.

3) If target field is shorter than the source field, bit copy the low-order bits to the target field.

Signed to unsigned with SGNCNV(LOGICAL):

1. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(31) SIGNED(TRUE);
   y1: BINARY LENGTH(32) PRECISION(32) SIGNED(FALSE) SGNCNV(LOGICAL);
   x1 <- -1;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFFFF',
   y1 = x'FFFFFFFF', and
   y1 = 4294967295 as an unsigned integer.
   ```

2. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(31) SIGNED(TRUE);
   y1: BINARY LENGTH(8) PRECISION(8) SIGNED(FALSE) SGNCNV(LOGICAL);
   x1 <- -1;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFFFF',
   y1 = x'FF', and
   y1 = 255 as an unsigned integer.
   ```

3. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(8) PRECISION(7) SIGNED(TRUE);
   y1: BINARY LENGTH(32) PRECISION(32) SIGNED(FALSE) SGNCNV(LOGICAL);
   x1 <- -128 as signed integer
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'80',
   y1 = x'FFFFFF80', and
   y1 = 4294967168 as an unsigned integer.
   ```

4. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(31) SIGNED(TRUE);
   y1: BINARY LENGTH(8) PRECISION(8) SIGNED(FALSE) SGNCNV(LOGICAL);
   x1 <- -128;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFF80',
   y1 = x'80', and
   y1 = 128 as an unsigned integer.
   ```

Unsigned to signed with SGNCNV(LOGICAL):

1. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(32) SIGNED(FALSE);
   y1: BINARY LENGTH(32) PRECISION(31) SIGNED(TRUE) SGNCNV(LOGICAL);
   x1 <- 4294967295;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFFFF',
   y1 = x'FFFFFFFF', and
   y1 = -1 as a signed integer.
   ```

2. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(32) SIGNED(FALSE);
   y1: BINARY LENGTH(8) PRECISION(7) SIGNED(TRUE) SGNCNV(LOGICAL);
   x1 <- 4294967295;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFFFF',
   y1 = x'FF', and
   y1 = -1 as a signed integer.
   ```

3. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(8) PRECISION(8) SIGNED(FALSE);
   y1: BINARY LENGTH(32) PRECISION(31) SIGNED(TRUE) SGNCNV(LOGICAL);
   x1 <- 255;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FF',
   y1 = x'000000FF', and
   y1 = 255 as a signed integer.
   ```

4. Given the following ADL declaration and assignment statements,

   ```
   x1: BINARY LENGTH(32) PRECISION(32) SIGNED(FALSE);
   y1: BINARY LENGTH(8) PRECISION(7) SIGNED(TRUE) SGNCNV(LOGICAL);
   x1 <- 4294967168;
   y1 <- x1;
   ```

   then,

   ```
   x1 = x'FFFFFF80',
   y1 = x'80', and
   y1 = -128 as a signed integer.
   ```

<SGNLOC attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the location of the sign in the representation of a PACKED or ZONED number.

Syntax

```
<SGNLOC attribute> ::=
  SGNLOC ({<constant identifier> | <positive integer> })
```

Syntax rules

The attribute value is the value of the ADL constant DGTLSTBYT, ZONFRSBYT, ZONLSTBYT, FRSBYT, or LSTBYT.

1. For PACKED fields, the following can be specified:

   - DGTLSTBYT - last nibble of last byte.

     ```
     byte    0         1         2         3
         |    |    |    |    |    |    |    |    |
         | D  | D  | D  | D  | D  | D  | D  | S  |
         |____|____|____|____|____|____|____|____|

     Legend:
       D = digit (4 bits)
       s = sign  (4 bits)
     ```

2. For ZONED fields, the following can be specified:

   - ZONLSTBYT - first nibble of last byte.

     ```
     byte    0         1         2         3
         |    |    |    |    |    |    |    |    |
         | Z  | D  | Z  | D  | Z  | D  | S  | D  |
         |____|____|____|____|____|____|____|____|

     Legend:
       Z = zone  (4 bits)
       D = digit (4 bits)
       S = sign  (4 bits)
     ```

   - ZONFRSBYT - first nibble of first byte.

     ```
     byte    0         1         2         3
         |    |    |    |    |    |    |    |    |
         | S  | D  | Z  | D  | Z  | D  | Z  | D  |
         |____|____|____|____|____|____|____|____|

     Legend:
       Z = zone  (4 bits)
       D = digit (4 bits)
       S = sign  (4 bits)
     ```

   - FRSBYT - first byte, containing a character '+' or '-' in the specified CCSID.

```
byte     0         1         2         3
    ┌────┬────┬────┬────┬────┬────┬────┬────┐
    │ssss│ssss│ Z  │ D  │ Z  │ D  │ Z  │ D  │
    └────┴────┴────┴────┴────┴────┴────┴────┘
Legend:
  Z = zone  (4 bits)
  D = digit (4 bits)
  s = sign according to CCSID
```

- LSTBYT - last byte, containing a character '+' or '-' in the specified CCSID.

```
byte     0         1         2         3
    ┌────┬────┬────┬────┬────┬────┬────┬────┐
    │ Z  │ D  │ Z  │ D  │ Z  │ D  │ssss│ssss│
    └────┴────┴────┴────┴────┴────┴────┴────┘
Legend:
  Z = zone  (4 bits)
  D = digit (4 bits)
  s = sign according to CCSID
```

General rules

None.

Examples

None.

<SGNMNS attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the hexidecimal values that can be used to represent the sign of a negative number when either:

ZONED fields are specified, if SGNLOC(ZONLSTBYT, ZONFRSBYT), or
PACKED fields are specified, if SGNLOC(DGTLSTBYT).

Syntax

```
<SGNMNS attribute> ::=
  SGNMNS ({<hex literal> | <constant identifier>})
```

Syntax rules

1. The values specified for the <SGNPLS attribute>, <SGNMNS attribute>, and <SGNUNS attribute> must be mutually exclusive.
2. The first <hex digit> specified in the <hex literal> is the preferred sign. The preferred sign is the sign generated by arithmetic operations of a given system.
3. <hex literal> has a maximum length of 8 nibbles (32 bits).
4. If SGNMNS(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <hex literal>.

General rules

None.

Examples

None.

<SGNPLS attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the hexidecimal values that can be used to represent the sign of a positive number when either:

ZONED fields are specified, if SGNLOC(ZONLSTBYT, ZONFRSBYT), or
PACKED fields are specified, if SGNLOC(DGTLSTBYT).

Syntax

```
<SGNPLS attribute> ::=
  SGNPLS ({<hex literal> | <constant identifier>})
```

Syntax rules

1. The values specified for the <SGNPLS attribute>, <SGNMNS attribute>, and <SGNUNS attribute> must be mutually exclusive.
2. The first <hex digit> specified in the <hex literal> is the preferred sign. The preferred sign is the sign generated by arithmetic operations of a given system.
3. <hex literal> has a maximum length of 8 nibbles (32 bits).
4. If SGNPLS(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <hex literal>.

General rules

None.

Examples

None.

<SGNUNS attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the hexadecimal values that can be used when no sign has been specified, for PACKED numbers with a sign position, if SGNLOC(DGTLSTBYT) specified.

Syntax

```
<SGNUNS attribute> ::=
  SGNUNS ({<hex literal> | <constant identifier>})
```

Syntax rules

1. The values specified for the <SGNPLS attribute>, <SGNMNS attribute>, and <SGNUNS attribute> must be mutually exclusive.
2. The first <hex digit> specified in the <hex literal> is the preferred sign. The preferred value is the value generated by arithmetic operations of a given system.
3. <hex literal> has a maximum length of 8 nibbles (32 bits).
4. If (<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <hex literal>.

General rules

None.

Examples

None.

<SIGNED attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify whether a BINARY, PACKED, or ZONED field includes a sign position in its representation.

Syntax

```
<SIGNED attribute> ::=
  SIGNED ({<boolean literal> | <constant identifier>})
```

Syntax rules

1. If SIGNED(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <boolean literal>.
2. If SIGNED(TRUE) is specified, a position for the sign exists within the representation.

3. If SIGNED(FALSE) is specified, a position for the sign does NOT exist within the representation.

General rules
None.

Examples
None.

<SKIP attribute>

The following shows the function, syntax, rules, and examples:

Function
Specify how many bits are to be skipped before the beginning of each element of an array.

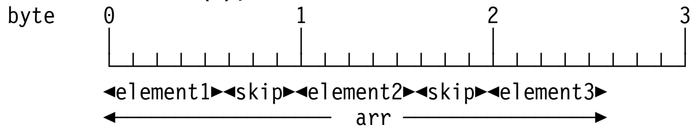
Syntax
```
 <SKIP attribute> ::=
   SKIP ({<positive integer> | <constant identifier>})
```

Syntax rules
1. The range of valid values for the <SKIP attribute> is 0 to <max31>.
2. The <SKIP attribute> does not apply to the first element of an ARRAY.
3. The <SKIP attribute> specifies the number of bits skipped before each succeeding element of the ARRAY.
4. If SKIP(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is in the range of the valid values of the attribute.

General rules
None.

Examples
Declare an array that contains 3 slack bits before each element:

```
arr: ARRAY SKIP(3) DMNLST(DMNLOW(1) DMNSIZE(3))
     OF BIT LENGTH(5);
```



<TITLE attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify a short description of an entity.

Syntax

```
<TITLE attribute> ::=
  TITLE
    ({
      <character literal> |
      <encoded hex literal> |
      <constant identifier>
    })
```

Syntax rules

1. The maximum length of TITLE is <max8> bytes.
2. If TITLE(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <character literal> or <encoded hex literal>.

General rules

None.

Examples

None.

<UNITLEN attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the length of each unit of an ASIS, CHAR, CHARPRE, or CHARSFX field.

Syntax

```
<UNITLEN attribute> ::=
  UNITLEN({ 1 | 8 | 16 | <constant identifier>})
```

Syntax rules

If UNITLEN(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is 1, 8, or 16.

General rules

None.

Examples

None.

<ZONENC attribute>

The following shows the function, syntax, rules, and examples:

Function

Specify the encoding of the zone portion of a <ZONED> field for all bytes except the sign.

Syntax

```
<ZONENC attribute> ::=
  ZONENC ({<hex literal> | <constant identifier>})
```

Syntax rules

1. The range of valid values is X'0' to X'F'.
2. If ZONENC(<constant identifier>) is specified, the <constant identifier> must be that of a <CONSTANT statement> whose value is a <hex literal>.

General rules

None.

Examples

1. x is a <ZONED> field that is printable in EBCDIC.

   ```
   x: ZONED PRECISION(3) SGNLOC(FRSBYT) CCSID(500);
   ```

2. y is a <ZONED> field that is printable in ASCII.

   ```
   y: ZONED PRECISION(3) SGNLOC(FRSBYT) ZONENC(x'3') CCSID(437);
   ```

3. z is a <ZONED> field that can be processed by 80386 machine instructions.

   ```
   z: ZONED PRECISION(3) SGNLOC(FRSBYT) ZONENC(x'0') CCSID(437);
   ```

Chapter 6. Functions

This chapter describes the LENGTH function that can be specified in ADL <assignment statements>s.

<LENGTH function>

The following shows the function, syntax, rules, and examples:

Function

Returns the actual field length of a data type, constructor, constant identifier, or literal. For the definition of “actual field length”, refer to Actual Field Length.

Syntax

```
<LENGTH function> ::=
LENGTH ({<literal> |
         <constant identifier> |
         <qualified identifier>
        })
```

Syntax rules

None.

General rules

1. The result of the <LENGTH function> is a BINARY value with BYTRVS(TRUE) for OS/2, or BYTRVS(FALSE) for AIX. The other attributes are the BINARY attributes. The result is specified in bytes if the field is CHAR, CHARPRE, CHARSFX, an encoded hex literal, or a character literal. Otherwise, the result of the <LENGTH function> is specified in bits (even for constructors composed exclusively of CHAR, CHARPRE, or CHARSFX fields).
2. The range of values returned is 0 to <max31> bits or 0 to <max28> bytes.

Examples

See Appendix A, “Scenarios” on page 175.

Chapter 7. Conversion of Data Types

Assignment statements, expressions, or comparisons of a plan cause data to be moved between declared variables of a module. If the data type and attribute of the source variable do not match the data type and attribute of the target variable, data conversions may be required. This chapter defines the rules that ADL conforms to when performing these conversions.

General Conversion Rules

The following rules apply to all of the conversions described in this chapter unless they are explicitly overridden for a particular conversion.

- All of the conversions between data types identified in Figure 6, and only those conversions, are supported.
- All supported conversions between data types are performed as defined in the following subsections of this book.
- The *1—Conversion not supported* exception occurs for all assignments that require data conversions not supported by ADL.

To Data Types

From Data Types	ARRAY	ASIS	BINARY	BIT	BITPRE	BOOLEAN	CASE	CHAR	CHARPRE	CHARSFX	ENUMERATION	FLOAT	PACKED	SEQUENCE	ZONED
ARRAY	X	X													
ASIS	X	X	X	X	X	X	X	X	X	X	X	X	X	X	X
BINARY		X	X								X	X	X		X
BIT		X		X	X										
BITPRE		X		X	X										
BOOLEAN		X				X									
CASE		X					X								
CHAR		X						X	X	X					
CHARPRE		X						X	X	X					
CHARSFX		X						X	X	X					
ENUMERATION		X	X								X	X	X		X
FLOAT		X	X								X	X	X		X
PACKED		X	X								X	X	X		X
SEQUENCE		X												X	
ZONED		X	X								X	X	X		X

Figure 6. ADL data-type conversion matrix.

X	*Conversion supported by DD&C*
blank	*Conversion not supported by DD&C*

Rules for Field Lengths

When describing conversion rules, information about the length of a field is often used. In an ADL source file, field lengths can be specified in a number of places:

- Default statements
- Subtype statements
- Data declaration statements
- Plan parameters.

Two length definitions are used:

1. The **maximum field length**. This is defined either with the MAXLEN attribute in a DECLARE statement or as the constant length of the field.
2. The **actual field length**. This is defined by the contents of a concrete buffer that contains information in the structure of the ADL field.

The actual field length need not be specified in a DECLARE statement. The maximum field length must, however, always be specified. The actual field length is always less than or equal to the maximum field length.

Maximum Field Length

Depending on the data type, the length in bits is defined as shown in Table 9.

Table 9. Determining the maximum field length

Field type	**Maximum length**
Fixed-length fields	Equal to the length of the field.
ARRAY	The length of an array with the maximum number of elements and the last element with its maximum length.
ASIS, CHAR, CHARSFX	MAXLEN * UNITLEN
BIT	MAXLEN
BITPRE	PRELEN + MAXLEN
CASE	Length of the alternative with the largest maximum length, or the value of LENGTH if the LENGTH attribute is specified and MAXALC is TRUE.
CHARPRE	PRELEN + (MAXLEN * UNITLEN).
SEQUENCE	The sum of all member data type lengths, including SKIP statements, and the maximum length of the last element.

Actual Field Length

The following definition also applies to terms such as “actual constructor length”, “actual source length”, and “actual target length”:

1. If the data type has a constant length, for example `FLOAT FORM(FB80)`, then the constant length is the actual data type length. In this example, it is 80 bits.
2. If the length of the storage allocated for the data type is fixed but the length of the information itself can vary, for example `BITPRE MAXALC(TRUE) MAXLEN(100) PRELEN(32)`, there are two possibilities:
   - If the field is not a target field, the actual data type length is calculated as shown in Table 10.

Table 10. Determining the actual field length

Field type	Actual length
ARRAY	The length of the array with the given number of elements and having the last element with its actual length.
ASIS, CHAR, CHARSFX	LENGTH * UNITLEN
BIT	LENGTH
BITPRE	PRELEN + LENGTH
CASE	Actual length of the currently valid alternative.
CHARPRE	PRELEN + (LENGTH * UNITLEN)
SEQUENCE	Sum of all member data type lengths, including SKIP statements and the actual length of the last element.

- If the field is a target field, the prefix value is updated during conversion. Therefore, the actual length before the conversion is equal to the maximum length (132 bits in the example).

3. If the storage allocated for the data type has a variable length, for example:

   ```
   BITPRE MAXALC(FALSE) MAXLEN(100) PRELEN(32)
   ```

   The following cases apply:

   - If the field is not a target field, the actual data type length is calculated in the same way as described in Table 10.
   - If the field is a target field, the prefix value is updated during conversion. Therefore, the actual target length is the minimum of the maximal length and either of the following:

     LENGTH * 8 - data type offset - 1
     MAXLEN * 8 - data type offset - 1

     Where LENGTH and MAXLEN are plan parameter attributes (if the field is contained in a plan parameter) and the plan parameter has a LENGTH or MAXLEN plan parameter attribute. This formula relies on the fact that variable-length fields can only appear at the end of their containing structures.

4. If the data type has a length that extends to the end of the record, for example:

   ```
   CHAR LENGTH(*) MAXLEN(80) UNITLEN(8)
   ```

   The actual field length then depends on whether the data type is contained in a plan parameter and whether the length of the outermost containing constructor is specified with a LENGTH plan parameter attribute, as in the case of an input parameter, or a MAXLEN plan parameter attribute, as in the case of an output parameter. In this case:

   - The actual data type length is the minimum of the maximal length and either FLOOR((LENGTH * 8 - data type offset - 1) / UNITLEN) * UNITLEN or FLOOR((MAXLEN * 8 - data type offset - 1) / UNITLEN) * UNITLEN, where LENGTH and MAXLEN are plan parameter attributes.

- Otherwise, the actual field length is the maximum length. In the previous example, the field has a length of 640 bits.

ARRAY to ARRAY

These are the general rules:

1. If the target array does not have the same number of dimensions as the source array, then the ***8—Nonconformable arrays*** exception occurs.
2. For each dimension of the source array:
   a. The low bound of the source dimension (LOW) is taken from:
      1) The <DMNLOW attribute> of the dimension
      2) A field referenced by the <DMNLOW attribute> of the dimension
      3) The <DMNLOW attribute> of the <ARRAY> declaration
      4) Or, the <DMNLOW attribute> specified in a <DEFAULT statement> for arrays.

   b. The high bound of the source dimension (HIGH) is taken from:
      1) The <DMNHIGH attribute> of the dimension
      2) A field referenced by the <DMNHIGH attribute> of the dimension
      3) Or, the formula

         ```
         source HIGH = source LOW + source SIZE - 1
         ```

   c. The size of the source dimension (SIZE) is taken from:
      1) The <DMNSIZE attribute> of the dimension
      2) A field referenced by the <DMNSIZE attribute> of the dimension
      3) Or, the formula

         ```
         source SIZE = source HIGH - source LOW + 1
         ```

3. For each dimension of the target array:
   a. Determine the values of target LOW, target HIGH, and target SIZE, as specified in Table 11 on page 141.
   b. If the source SIZE does not equal the target SIZE, then the ***8—Nonconformable arrays*** exception occurs. This rule covers all cases of array assignment to a target array dimension defined without reference fields.
   c. If the source SIZE is greater than the <DMNMAX attribute> of the target, then the ***8—Nonconformable arrays*** exception occurs. This rule covers all cases of array assignment to a target array dimension defined with reference fields.
   d. On completion of an <assignment statement>, if any target <field> is referenced by an array dimension specification, the required value for its referenced use is assigned to that field. This is done at the end of an assignment to ensure values are not changed by other operations of an <assignment statement>. Refer to Table 11 on page 141.
4. The elements of the source array are assigned to the elements of the target array such that the relative position of an element within each dimension is preserved.

The association of elements with specific source array indexes in each dimension is not be preserved.

5. Conversion of the elements of the source array to the representation required by the elements of the target array follows the conversion matrix specified in Figure 6 on page 136.

Table 11. Target array dimension considerations.

1. *In the condition values:*

   I *Indicates a constant specified or defaulted for the attribute.*
   R *Indicates a reference to another field.*
   — *Indicates a condition that cannot occur.*

2. **X** *indicates that an action is taken. Where several actions are possible, they are performed in the sequence specified by the action rows of the table. Actions can be dependent on previously selected actions.*

Target <DMNLOW attribute> specified by	I	I	I	I	R	R	R	R
Target <DMNHIGH attribute> specified by	I	R	—	—	I	R	—	—
Target <DMNSIZE attribute> specified by	—	—	I	R	—	—	I	R
Target LOW = target DMNLOW	X	X	X	X				
Target LOW = target DMNHIGH - source SIZE + 1					X			
Target LOW = source LOW						X		X
Target LOW = source HIGH - target DMNSIZE + 1							X	
Target HIGH = target DMNHIGH	X				X			
Target HIGH = target LOW + source SIZE - 1		X				X		
Target SIZE = target DMNSIZE			X				X	
Target SIZE = source SIZE		X		X		X		X
Target SIZE = target HIGH - target LOW + 1	X				X			

ASIS to Constructor

These are the general rules:

1. An ASIS-to-constructor conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source field and the target constructor for the lengths of the source field and the target constructor.

2. The length of the constructor is the actual constructor length, as defined in “Actual Field Length” on page 137.

3. If the source field is longer than the target constructor, the source data is truncated to the length of the target constructor.

4. If the source field is shorter than the target constructor, the source data is padded to the right with B'0' to the length of the target constructor.

5. If reference fields are specified in the constructor, the reference fields are not updated after the bit-string copy.

ASIS to Field Data Types

These are the general rules:

1. An ASIS-to-field conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source field and the target field for the lengths of the source field and the target field.
2. The target is considered to be a bit string, where:
   a. If the target field is ASIS or BIT:

      Refer to Table 12 on page 144 for target length determinations.

   b. If the target field is BINARY:
      1) The target length is obtained from LENGTH, if present.
      2) The target length is inferred from PRECISION.

   c. If the target field is BITPRE or CHARPRE:
      1) If MAXALC(TRUE) is specified for the target, the target length equals target(PRELEN + MAXLEN * UNITLEN).
      2) If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following two values:
         - target PRELEN + CEIL(source(LENGTH * UNITLEN) / target UNITLEN) * target UNITLEN
         - actual target length.

         The ASIS field is copied only into the value field of the target data type. The prefix field of the target data type is then updated with the correct length.

   d. If the target field is BOOLEAN, ENUMERATION, or FLOAT: the target length is obtained from its LENGTH attribute or implied length.

   e. If the target field is CHAR:
      1) Refer to "BIT to BIT" on page 143 for the length of the target if the <LENGTH attribute> is used.
      2) Refer to Table 11 on page 141 for the length of the target if the <HIGH attribute> and <LOW attribute> are used.
      3) The obtained length of the target must be multiplied by UNITLEN to obtain its bit-string length.

   f. If the target field is CHARSFX:
      1) If MAXALC(TRUE) is specified for the target, the target length equals the target (MAXLEN * UNITLEN).
      2) If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following two values:
         - CEIL(length of suffix + source(LENGTH * UNITLEN) / target UNITLEN) * target UNITLEN

        - actual target length.

    3) When all bits of the string have been copied to the target, a suffix is appended indicating the length of the target.

   g. If the target field is PACKED, refer to "General rules" on page 94 for the determination of the length.

   h. If the target field is ZONED, refer to "General rules" on page 100 for the determination of the length.

3. Refer to "BIT to BIT" for further conversion rules.
4. If the target is CHAR and reference fields were used in the target, refer to Table 15 on page 149 for update rules.
5. If the source field includes a reference to another field and either the value of the LENGTH attribute is greater than that of the MAXLEN attribute or the remaining buffer size is too small, the ***27—Invalid LENGTH value of ASIS, BIT, or BITPRE field*** exception occurs.

BINARY, FLOAT, PACKED, and ZONED to ENUMERATION

These are the general rules:

1. If the source is a PACKED or ZONED field, the source is converted to a BINARY field following the rules specified for that conversion.
2. The value of the source BINARY field is compared with the values associated with the <enumeration identifier>s of the target ENUMERATION field.
   a. If a match is found, the value of the source BINARY field is converted to the attributes of the target ENUMERATION field.
   b. If no match is found, the ***10—Invalid ENUMERATION value*** exception occurs.

BIT to BIT

These are the general rules:

1. The actual length of the source is defined in "Actual Field Length" on page 137.
2. This conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source and target fields for the lengths of the source and target fields.
3. If the target is BIT, Table 12 on page 144 specifies how much space is available for the target.
4. If the bit length of the source is longer than the target field, the source data is truncated on the right to the length of the target field.
5. If the bit length of the source is shorter than the target field, the source data is padded to the right with `B'0'` to the length of the target field.
6. If reference fields were used in the target, Table 12 on page 144 specifies how the target length reference fields are updated at the end of an assignment statement.

Interpret the values “source UNITLEN” and “target UNITLEN” as having the value 1.

Table 12. Determination of target ASIS, BIT length fields, and length reference field updates.

1. *In the condition values:*

I	*Indicates a constant specified or defaulted for the attribute.*
R	*Indicates a reference to another field.*
*****	*Indicates LENGTH(*).*
—	*Indicates a condition that cannot occur.*
T	*Indicates a “TRUE” value.*
F	*Indicates a “FALSE” value.*

2. **X** *indicates that an action is taken. Where several actions are possible, they are performed in the sequence specified by the action rows of the table. Actions can be dependent on previously selected actions.*

Target <LENGTH attribute> specified by	I	*	R	R
Target <MAXALC attribute> value	—	—	T	F
Target length = target LENGTH	X			
Target length = target MAXLEN			X	
Target length = MIN((actual target length / target UNITLEN), CEIL(actual source length / target UNITLEN))		X		X
Update referenced length field with MIN((actual target length / target UNITLEN), CEIL(actual source length / target UNITLEN))			X	X

BIT to BITPRE

These are the general rules:

1. The actual length of the source is defined in “Actual Field Length” on page 137.
2. If MAXALC(TRUE) is specified for the target, the target field length equals the target MAXLEN plus PRELEN.
3. If MAXALC(FALSE) is specified for the target, the target field length equals MIN(actual source length + target PRELEN, actual target length).
4. A BIT-to-BITPRE conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source and target fields for the lengths of the source and target fields.
5. If the source field is longer than the length of the target field, the source data is truncated to the length of the target field.
6. The length of the bit string copied into the target field is encoded in the prefix of the target field, as specified by its PRELEN and PREBYTRVS attributes.
7. If the source field includes a reference to another field and either the value of the LENGTH attribute is greater than that of the MAXLEN attribute or the remaining buffer size is too small, the ***27—Invalid LENGTH value of ASIS, BIT, or BITPRE field*** exception occurs.

BITPRE to BIT

These are the general rules:

1. A BITPRE-to-BIT conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source data portion of the field and target field for the lengths of the source and target fields.
2. The length of the source is obtained from the prefix of the source field, which is encoded as specified by its PRELEN and PREBYTRVS attributes.
3. Table 12 on page 144 specifies how the length of the target is obtained.
4. If the bit length of the source is longer than the target field, the source data is truncated on the right to the length of the target field.
5. If the bit length of the source is shorter than the target field, the source data is padded to the right with `B'0'` to the length of the target field.
6. If reference fields were used in the target, Table 12 on page 144 specifies how, at the end of an assignment statement, the reference fields of the target are updated.
7. If the source field includes a reference to another field and either the value of the LENGTH attribute is greater than that of the MAXLEN attribute or the remaining buffer size is too small, the ***27—Invalid LENGTH value of ASIS, BIT, or BITPRE field*** exception occurs.

BITPRE to BITPRE

These are the general rules:

1. A BITPRE-to-BITPRE conversion is a bit-string copy starting with the leftmost bit (bit 1) of the source and target fields for the lengths of the source data and the target field.
   a. The length of the source data is taken from the prefix of the source field, which is encoded as specified by its PRELEN and PREBYTRVS attributes.
   b. If MAXALC(TRUE) is specified for the target, the target length equals the target MAXLEN plus PRELEN.
   c. If MAXALC(FALSE) is specified for the target, the target field length is the minimum of the following values:
      - actual source length - source PRELEN + target PRELEN
      - actual target length.
2. If the source data is longer than the length of the target field, the source data is truncated to the length of the target field.
3. The length of the bit string copied into the target field is encoded in the prefix of the target field, as specified by its PRELEN and PREBYTRVS attributes.
4. If the source field includes a reference to another field and either the value of the LENGTH attribute is greater than that of the MAXLEN attribute or the remaining

buffer size is too small, the 27—Invalid LENGTH value of ASIS, BIT, or BITPRE field exception occurs.

BOOLEAN to BOOLEAN

This is the general rule:

General rule

The value of the source field (FALSE or TRUE), encoded as specified by the attributes of the source BOOLEAN field, is converted to the corresponding encoding, as specified by the attributes of the target BOOLEAN field. Only the significant bit is copied (see "<BLNENC attribute>" on page 103).

CASE to CASE

These are the general rules:

1. The <condition>s of the source <WHEN statement>s are evaluated in the order in which the <WHEN statement>s are specified in the source CASE until one of the evaluations returns TRUE:
   a. The <WHEN statement> of the target CASE with the same statement <identifier> or <positional identifier> is then selected.
   b. Conversion of data from the declaration specified by the selected source <WHEN statement> to the declaration specified by the selected target <WHEN statement> follows the general rules specified in "General Conversion Rules" on page 135.
   c. If no <WHEN statement> is found in the target CASE with the same statement <identifier> or <positional identifier>, then the ***24—Target CASE mismatch*** exception occurs.
   d. On completion of the <assignment statement>, the <condition> of the selected <WHEN statement> must evaluate to TRUE and the <condition> of all target <WHEN statement>s preceding the selected target <WHEN statement> must evaluate to FALSE. Otherwise, the ***6—Target CASE failure*** exception occurs.
2. If no source <WHEN statement> evaluates to TRUE, the <OTHERWISE statement> of both the source and target CASE is selected:
   a. If no <OTHERWISE statement> is specified on the source or target and all respective <WHEN statement>s evaluate to FALSE, the plan is terminated and the ***20—CASE rejected*** exception occurs.
   b. Conversion of data from the declaration specified by the source <OTHERWISE statement> to the declaration specified by the target <OTHERWISE statement> follows the general rules specified in "General Conversion Rules" on page 135.
   c. On completion of the <assignment statement>, the condition of all target <WHEN statement>s must evaluate to FALSE. Otherwise, the ***6—Target CASE failure*** exception occurs.

CHARxxx to CHARxxx

These are the general rules:

1. The characters of the source data are converted from the encodings specified by the source CCSID to the encodings specified by CDRA for the target CCSID.
   a. If the source or target CCSID is not defined by CDRA, the ***4—Undefined CCSID*** exception occurs.
   b. If the source or target CCSID is not supported, the ***2—CCSID not supported*** exception occurs.
   c. If the conversion between the source CCSID and the target CCSID is invalid, the ***3—Invalid CCSID pair*** exception occurs.

CHARxxx to CHAR

These are the general rules:

1. Table 13 on page 148 specifies the rules for obtaining the target length if the <LENGTH attribute> is specified.
2. Table 14 on page 148 specifies the rules for obtaining the target length if the <HIGH attribute> and <LOW ATTRIBUTE> are specified.
3. If JUSTIFY(LEFT) is specified for the target,
   a. If the converted data is greater than the target length, the converted data is truncated on the right to the length of the target.
   b. If the converted data is shorter than the target length, the converted data is padded to the right with the space character defined by CDRA for the target CCSID.
   c. The converted data is copied left-aligned to the target data area.
4. If JUSTIFY(RIGHT) is specified for the target,
   a. If the converted data is shorter than the target length, the converted data is padded to the left with the space character defined by CDRA for the target CCSID.
   b. If the converted data is greater than the target length, the converted data is truncated on the left to the length of the target.
   c. The converted data is copied right-aligned to the target data area.
5. If reference fields were used in the target, Table 15 on page 149 specifies how, at the end of an assignment statement, the length reference fields of the target are updated.

Table 13. CHARxxx-to-CHAR target length considerations with the LENGTH attribute.

1. *In the condition values,*

   I *Indicates a constant specified or defaulted for the attribute.*
   R *Indicates a reference to another field.*
   ***** *Indicates LENGTH(*).*
   — *Indicates a condition that cannot occur.*
   T *Indicates a "TRUE" value.*
   F *Indicates a "FALSE" value.*

2. **X** *indicates that an action is taken. Actions are performed in the sequence specified by the action rows of the table. Actions can be dependent on previously selected actions.*

Target <LENGTH attribute> specified by	I	*	R	R
Target <MAXALC attribute> value	—	—	T	F
Target length = target LENGTH	X			
Target length = target MAXLEN			X	
Target length = MIN((actual target length / target UNITLEN), CEIL((actual source length - source PRELEN) / target UNITLEN))		X		X

Table 14. CHARxxx-to-CHAR target length considerations with the HIGH and LOW attributes.

1. *In the condition values,*

   I *Indicates a constant specified or defaulted for the attribute.*
   R *Indicates a reference to another field.*

2. **X** *indicates that an action is taken. Where several actions are possible, they are performed in the sequence specified by the action rows of the table. Actions can be dependent on previously selected actions.*

Target <HIGH attribute> value	I	R	I	R
Target <LOW attribute> value	I	I	R	R
Target high = target HIGH	X		X	
Target low = default value for target LOW, or 1 if target default value for LOW does not exist.				X
Target high = target LOW + source length - 1		X		X
Target low = target LOW	X	X		
Target low = target HIGH - source length + 1			X	
Target length = target HIGH - target LOW +1	X	X	X	X

Table 15. CHARxxx-to-CHAR target reference field update considerations.

1. *In the condition values,*

I	*Indicates a constant specified or defaulted for the attribute.*
R	*Indicates a reference to another field.*
—	*Indicates a condition that cannot occur.*
T	*Indicates a "TRUE" value.*
F	*Indicates a "FALSE" value.*

2. **X** *indicates that an action is taken. Where several actions are possible, they are performed in the sequence specified by the action rows of the table. Actions can be dependent on previously selected actions.*

Target <LENGTH attribute> value	R	R	—	—	—
Target <MAXALC attribute> value	T	F	—	—	—
Target <HIGH attribute> value	—	—	R	I	R
Target <LOW attribute> value	—	—	I	R	R
Update referenced length field with MIN((actual target length / target UNITLEN), CEIL((actual source length - source PRELEN) / target UNITLEN))	X	X			
Update HIGH(qualifier) to its derived value.			X		X
Update LOW(qualifier) to its derived value.				X	X

CHARxxx to CHARPRE

These are the general rules:

1. If MAXALC(TRUE) is specified for the target, the target length equals the target MAXLEN.
2. If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following two values (source PRELEN and source suffix length are set to zero if they do not apply for the source data type):
   - CEIL((actual source length - source PRELEN - source suffix length) / target UNITLEN) * target UNITLEN + target PRELEN
   - Actual target length.
3. If the length of the converted source is greater than the target length, the data is truncated on the right to the target length.
4. The length of the character string copied into the target field is encoded in the prefix of the target field, as specified by its <PRELEN attribute>, <PREBYTRVS attribute>, and <UNITLEN attribute>.
5. The converted data is copied left-aligned to the target data portion of the field.

CHARxxx to CHARSFX

These are the general rules:

1. If MAXALC(TRUE) is specified for the target, the target length equals the target MAXLEN.
2. If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following two values (source PRELEN and source suffix length are set to zero if they do not apply for the source data type):
   - CEIL((actual source length - source PRELEN - source suffix length + target suffix length) / target UNITLEN) * target UNITLEN
   - Actual target length.
3. If the length of the converted source is greater than the target length, the data is truncated on the right to the target length.
4. If the CCSID of the target string is single byte or mixed byte, a suffix X'00' is appended. If the CCSID of the target string is double byte, a suffix X'0000' is appended.
5. The converted data is copied left-aligned to the target data area.

Constructor to ASIS

These are the general rules:

1. The length of the constructor is the inferred length of the constructor.
2. Table 12 on page 144 specifies how the length of the target is obtained.
3. A constructor-to-ASIS conversion is a bit-string copy starting with the leftmost bits (bit 1) of the source constructor and the target field for the lengths of the source constructor and target field.
4. If the bit length of the source constructor is longer than the target field, the source data is truncated on the right to the length of the target field.
5. If the bit length of the source constructor is shorter than the target field, the source data is padded to the right with B'0' to the length of the target field.
6. If reference fields were used in the target, Table 12 on page 144 specifies how, at the end of an assignment statement, the reference fields of the target are updated.

ENUMERATION to BINARY, FLOAT, PACKED, and ZONED

These are the general rules:

1. The value of the source field is assumed to be a BINARY field whose length is specified by the <LENGTH attribute> specified in the source declaration, with SCALE(0) and RADIX(2).
2. The source value is converted to the attributes of the target field as specified for BINARY-to-BINARY, BINARY-to-PACKED, BINARY-to-FLOAT, or BINARY-to-ZONED conversions.

ENUMERATION to ENUMERATION

These are the general rules:

1. The binary value of the source field is used to determine the <enumeration identifier> of the source declaration associated with that value. If no <enumeration identifier> of the source declaration can be associated with the value of the source field, the ***10—Invalid ENUMERATION value*** exception occurs.
2. The <enumeration identifier> of the source field is compared to the <enumeration identifier>s of the target declaration of the field.
3. If the <enumeration identifier> of the source matches one of the <enumeration identifier>s of the target, the binary value associated with the matching <enumeration identifier> of the target is used as the value of the field.
4. If the <enumeration identifier> of the source does not match an <enumeration identifier> of the target, the ***9—ENUMERATION mismatch*** exception occurs.

Examples

1.

- Specify `color` in the base record to be:

  ```
  color: ENUMERATION(red:1, green:3, blue:5, white:7);
  ```

  The values associated with each are:

  ```
  red=1, green=3, blue=5, white=7
  ```

- Specify `color` in the view record to be:

  ```
  color: ENUMERATION(red:1, green:3, blue, white);
  ```

  The values associated with each are:

  ```
  red=1, green=3, blue=4, white=5
  ```

- The identifiers are matched during conversions resulting in:

  ```
  base                  view
  red   is matched with  red   and the target value is 1
  green is matched with  green and the target value is 3
  blue  is matched with  blue  and the target value is 4
  white is matched with  white and the target value is 5
  ```

2.

- Specify `animal` in the base record to be:

  ```
  animal: ENUMERATION(cat:1, dog:3, monkey:5, elephant:7);
  ```

  The values associated with each are:

  ```
  cat=1, dog=3, monkey=5, elephant=7
  ```

- Specify `animal` in the view record to be:

  ```
  animal: ENUMERATION(monkey, elephant, cat, dog);
  ```

  The values associated with each are:

  ```
  monkey=0 elephant=1 cat=2 dog=3
  ```

- The identifiers are matched during conversions resulting in:

  ```
  base                     view
  cat      is matched with  cat       and the target value is 2
  dog      is matched with  dog       and the target value is 3
  monkey   is matched with  monkey    and the target value is 0
  elephant is matched with  elephant  and the target value is 1
  ```

Field Data Types to ASIS

These are the general rules:

1. A field data type-to-ASIS conversion consists of copying the encoded bits of the source field to the target ASIS field starting with the leftmost bits (bit 1) of the source field and the target field for the lengths of the source field and target field.
2. The target is considered to be a bit-string where:
   a. If the source field is ASIS or BIT:

      Refer to Table 12 on page 144 for target length determinations.

   b. If the source field is BINARY:
      1) The target length is obtained from LENGTH, if present.
      2) The target length is inferred from PRECISION.

   c. If the source field is BITPRE or CHARPRE, the value field, but not the prefix field of the BITPRE or CHARPRE field, is copied to the ASIS field:
      1) If MAXALC(TRUE) is specified for the target, the target length equals the target (MAXLEN * UNITLEN).
      2) If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following values:
         - CEIL((actual source length - source PRELEN) / target UNITLEN) * target UNITLEN
         - FLOOR(actual target length / target UNITLEN) * target UNITLEN.

   d. If the source field is BOOLEAN, ENUMERATION, or FLOAT: the target length is obtained from the source LENGTH attribute or implied length.

e. If the source field is CHAR:

    1) Refer to Table 13 on page 148 for the length of the target if the <LENGTH attribute> is used.

    2) Refer to Table 14 on page 148 for the length of the target if the <HIGH attribute> and <LOW attribute> are used.

    3) The obtained length of the target is multiplied by the source UNITLEN to get its bit-string length.

f. If the source field is CHARSFX:

    1) If MAXALC(TRUE) is specified for the target, the target length equals the target (MAXLEN * UNITLEN).

    2) If MAXALC(FALSE) is specified for the target, the target length equals the minimum of the following values:

        - CEIL((actual source length - source suffix length) / target UNITLEN) * target UNITLEN

        - FLOOR(actual target length / target UNITLEN) * target UNITLEN

    3) All characters up to, but not including, the suffix are copied to the target.

g. If the source field is PACKED, refer to “General rules” on page 94 for the determination of the length.

h. If the source field is ZONED, refer to “General rules” on page 100 for the determination of the length.

3. Refer to “BIT to BIT” on page 143 for further conversion rules.

4. If reference fields were used in the target, Table 12 on page 144 specifies how, at the end of an assignment statement, the reference fields of the target are updated.

5. If the bit-string encoding of the source field is longer than the bit length of the target field, the source data is truncated on the right to the length of the target field.

6. If the bit-string encoding of the source field is shorter than the bit length of the target field, the source data is padded to the right with `B'0'` to the length of the target field.

Numeric Conversions

This section describes the mathematical basis for the numeric conversions.

References to binary FLOAT types are to FLOAT types with FORM(FB32 | FB64 | FB80 | FI128).

References to hexadecimal FLOAT types are to FLOAT types with FORM(FH32 | FH64 | FH128).

Value of fixed-point numbers

The value represented by a fixed-point (ADL type BINARY, PACKED or ZONED) number is given by:

```
(01)  N = V*(R**-S)
        = V/(R**S), where
          N = the effective value being represented.
          V = the integral value which is stored in the representation
              of N.
          R = the radix of the representation of N, as specified by
              N's RADIX attribute if N is of type BINARY; otherwise
              implicitly equal to 10.
          S = the scale of the representation of N, as specified by
              N's SCALE attribute.
```

Thus for an unsigned BINARY number with a stored value of, say, B'11', the following values of N may result:

```
             When R = 2              When R = 10
  S =  2     N = 3*(1/4) =   .75     N = 3*(1/100) =     .03
  S =  1     N = 3*(1/2) =  1.5      N = 3*(1/10)  =     .3
  S =  0     N = 3*1     =  3        N = 3*1       =    3
  S = -1     N = 3*2     =  6        N = 3*10      =   30
  S = -2     N = 3*4     = 12        N = 3*100     =  300
  S = -3     N = 3*8     = 24        N = 3*1000    = 3000
```

In each case the *scale factor*, as given by R**-S, defines the weight assigned to the low-order digit in the stored values. For S=2, the stored value indicates the number of quarters when R equals 2, and the number of hundredths when R equals 10.

It is important to remember that the stored value for a BINARY number is, appropriately, encoded in binary—even though the RADIX attribute may specify 10.

Value of hexadecimal-FLOAT numbers

The value represented by a hexadecimal-FLOAT number is given by:

```
(02)  N = F*(16**e)
        = F*(16**(C-b))
        = F*(16**(C-64)), where:
          N = the effective value being represented.
          F = the value which is stored in the significand of N.
              F is understood to have an implicit radix point
              immediately to its left, with the result that:
                |F| < 1
          e = the exponent.  Equivalent to C-b, where:
              C = the characteristic stored in the representation of N.
              b = 64, the bias.
```

A hexadecimal-FLOAT number is considered to be *normalized* if the leftmost hexadecimal digit of the significand is nonzero (assuming N to be nonzero); that is:

```
(03)  (1/16) <= |F| < 1
```

Value of binary-FLOAT numbers

The following material deals with numeric values only; it does not deal with “Not a Number” (NaN) or infinity.

Normalized numbers: A normalized binary-FLOAT number has a nonzero characteristic, and the value represented is given by:

```
(04)  N = F*(2**e)
        = F*(2**(C-b)), where:
          N = The effective value being represented.
          F = The significand of N.  F is composed of a 1, followed
              by an implicit radix point, followed by a fraction; with
              the result that:
                1 <= |F| < 2
              If the field is extended-precision, then the entire
              significand, including the leading 1, is stored
              left-justified in the field's significand bits.
              If the field is not extended-precision, then only the
              fractional part is stored, also left-justified, and
              the implicit radix point stands immediately to the
              left.
          e = The exponent.  Equivalent to C-b, where:
              C = the characteristic stored in the representation of N.
              b = 127, when single-precision.
              b = 1023, when double-precision.
              b = 16383, when extended-precision.
```

Denormal numbers: A Denormal binary-FLOAT number has a zero characteristic, and the value represented is given by:

```
(05)  N = F*(2**e)
        = F*(2**(C-b)), where:
          N = The effective value being represented.
          F = The significand of N.  F is composed of an implicit
              radix point, followed by a stored fraction; with the
              result that:
                |F| < 1
          e = The exponent.  Equivalent to C-b, where:
              C = the characteristic stored in the representation of N.
              b = 127, when single-precision.
              b = 1023, when double-precision.
              b = 16383, when extended-precision.
```

A denormal binary-FLOAT number with a zero-significand represents a zero.

Fixed-point to fixed-point conversions

The objective of a numeric conversion is to set the stored value of the target field in such a way that the target's effective value, N_t, is as near as possible to the source field's effective value, N_s. That is:

(06) $N_t \simeq N_s$

Substituting from (01):

(07) $V_t*(R_t**-S_t) \simeq V_s*(R_s**-S_s)$

Solving for V_t, the value to be stored in the target:

(08) $V_t \simeq V_s*((R_t**S_t)/(R_s**S_s))$

The reasons why approximations (06), (07), and (08) may not be exact is that not all conversions can be exact. Consider a source BINARY such that:

$R_s = 10$ $S_s = 1$ $V_s = 15$
Then,
$N_s = 1.5$, from (01).

If this source BINARY is converted to radix-2 BINARY targets with various scales we get:

	--- V_t from (08) --	N_t from (01)
$S_t = 2$	$\simeq 15*(4/10) = 6$	1.5
$S_t = 1$	$\simeq 15*(2/10) = 3$	1.5
$S_t = 0$	$\simeq 15*(1/10) = 1.5$	1 or 2
$S_t = -1$	$\simeq 15*(1/20) = .75$	0 or 2
$S_t = -2$	$\simeq 15*(1/40) = .375$	0

Notice that when S_t equals 2 or 1 the corresponding V_t represents the number of quarters or halves, respectively, and N_t equals N_s exactly. In the other three cases, the requirement that V_t be an integer necessitates truncation or rounding, with various degrees of correspondence between N_t and N_s.

Another way to view the situation is that, for S_t equal to 0, -1, or -2, V_t represents the number of 1s, 2s, or 4s, respectively, and fractional values cannot be represented.

The preceding example demonstrates how digits may be lost from the right-hand end of V_t. Notice that digits may be lost from the left-hand end if the target precision, P_t, as given by the target's PRECISION attribute, is insufficient to accommodate the results given by (08). For example:

R_s = 10 R_t = 10
S_s = 2 S_t = 2
V_s = 123425 P_t = 4
N_s = 1234.25, from (01).

Then,

(09) V_t ≃ 123425*((10**2)/(10**2)) = 123425, from (08).
But, because P_t equals 4, the results of (09) must
be of the form MOD(x,10**4), yielding:
MOD(123425,10**4) = MOD(123425,10000) = 3425
Applying (01) for V_t = 3425 gives the result:
N_t = 34.25

In the case of unconstrained targets, the width of V_t may not be limited by the target's PRECISION attribute.

Rounding and truncation of fixed-point numbers: That (08) must be an approximation is due to the fact that in the computer environment numbers are constricted to finite-length representations, and, in some cases, the attributes of source and target numbers may necessarily be inexact, as shown in the examples.

The purpose of this section is to derive from approximation (08) an unambiguous equation which exactly describes fixed-point to fixed-point conversions. To this end, four new variables are introduced:

V_i An intermediate stored value derived directly from (08). It is introduced to help make clear how truncation and rounding are accomplished.

W_t The effective width of the target. This variable is introduced to accommodate high-order truncation.

B_{wt} The effective base against which W_t is applied. Its value is either 2 or 10.

H The half-round term that determines whether the low-order end of the target is truncated or rounded. Its value is either .5 or zero, depending on the FIT attribute of the target.

The derivation of an equality to replace approximation (08) is:

(10) V_i = V_s*((R_t**S_t)/(R_s**S_s)), from (08).
(11) V_t = MOD(FLOOR(V_i+H),(B_{wt}**W_t)), when V_i >= 0.
V_t = MOD(CEIL(V_i-H),(B_{wt}**W_t)), when V_i < 0.

Equation (10) accepts the results of (08) without constraints. Equation (11) applies the constraints. The first argument to the MOD function ensures that the stored target value is an integer, rounded or truncated if necessary. Application of the MOD function ensures that the stored target value will, if necessary, be truncated on the left to the proper width.

It remains to show how W_t, B_{wt} and H receive their values. All ADL attributes used are those of the target:

W_t—is equal to:

- The PRECISION attribute when the target is:
  - BINARY CONSTRAINED(TRUE)
  - PACKED CONSTRAINED(TRUE)
  - PACKED CONSTRAINED(FALSE) SIGNED(TRUE) with odd PRECISION
  - PACKED CONSTRAINED(FALSE) SIGNED(FALSE) with even PRECI-SION
  - ZONED
- The PRECISION attribute + 1 when the target is:
  - PACKED CONSTRAINED(FALSE) SIGNED(TRUE) with even PRECI-SION
  - PACKED CONSTRAINED(FALSE) SIGNED(FALSE) with odd PRECISION
- The LENGTH attribute when the target is:
  - BINARY CONSTRAINED(FALSE) SIGNED(FALSE)
- The LENGTH attribute - 1 when the target is:
  - BINARY CONSTRAINED(FALSE) SIGNED(TRUE)

B_{wt}—is equal to:

- The RADIX attribute when the target is:
  - BINARY CONSTRAINED(TRUE)
- 2 when the target is:
  - BINARY CONSTRAINED(FALSE)
- 10 when the target is:
  - PACKED
  - ZONED

H —is equal to:

- Zero when FIT(TRUNCATE)
- 0.5 when FIT(ROUND) or FIT(EXACT)

Hexadecimal-FLOAT to fixed-point conversions

This conversion is similar to the fixed-point-to-fixed-point conversions:

(12) $N_t \simeq N_s$
Substituting from (01) for the target and from (02) for the source:
(13) $V_t*(R_t**-S_t) \simeq F_s*(16**e_s)$
Solving for V_t, the value to be stored in the target:
(14) $V_t \simeq F_s*(16**e_s)*(R_t**S_t)$

Since the attributes specified for the target may preclude an exact conversion, provision is made for truncation and rounding, as is done for conversion between fixed-point numbers:
(15) $V_i = F_s*(16**e_s)*(R_t**S_t)$, from (14).
then, using equation (11) to apply constraints:
$V_t = \text{MOD}(\text{FLOOR}(V_i+H),(B_{wt}**W_t))$, when $V_i >= 0$.
$V_t = \text{MOD}(\text{CEIL}(V_i-H),(B_{wt}**W_t))$, when $V_i < 0$.

The variables H, B_{wt}, and W_t receive their values as previously described in “Rounding and truncation of fixed-point numbers” on page 157.

Fixed-point to hexadecimal-FLOAT conversions

This derivation is for nonzero values and is intended to yield a normalized target. The conversion is complicated somewhat by the fact that two values must be stored in the target—the significand and the characteristic. Portions of the following treatment are expressed in terms of absolute values in order to facilitate the use of logarithms:

(16) $|N_t| \simeq |N_s|$
Substituting from (02) for the target:
(17) $|F_t*(16**e_t)| \simeq |N_s|$, where:
$N_s = V_s*(R_s**-S_s)$, from (01).
Temporarily letting F_t equal 16**-1 (the minimum normalized significand) and taking the base-16 logarithm:
(18) $-1+e_t \simeq \log_{16}(|N_s|)$
(19) $e_t \simeq \log_{16}(|N_s|)+1$
Since the exponent must be an integer, set it as:
(20) $e_t = \text{FLOOR}(\log_{16}(|N_s|)+1)$
Application of (20) implies that F_t may no longer be equal to 16**-1, but lies in the range:
$(1/16) <= F_t < 1$
The final approximation of F_t, with consideration for the sign, is determined by:
$F_t*(16**e_t) \simeq N_s$, from (17).
(21) $F_t \simeq N_s*(16**-e_t)$

This produces an exact expression for the target exponent, e_t, and an approximation of the target significand, F_t; both are expressed as functions of the effective source value, N_s. The next step is to determine the stored target characteristic, C_t, and an exact expression for F_t, allowing for possible rounding or truncation to the target significand width.

The characteristic is:

(22) $C_t = e_t+64$

Using F_i as an intermediate value, the final expression of the target significand is derived by:

```
      Fi = Ns*(16**-et), from (21).
(23)  Ft = FLOOR((Fi*(16**Wt))+H) * (16**-Wt), when Fi > 0.
      Ft = CEIL((Fi*(16**Wt))-H) * (16**-Wt), when Fi < 0.
      where:
           Wt =  6, for single precision.
           Wt = 14, for double precision.
           Wt = 28, for extended precision.
           H = 0, when FIT(TRUNCATE).
           H = 0.5, when FIT(ROUND) or FIT(EXACT).
```

The expression within the FLOOR or CEIL function effectively moves the radix point right, to the width of the target significand, and rounds the integer part of the result. The FLOOR or CEIL function then truncates any remaining fraction. The FLOOR or CEIL function is then multiplied by a factor that effectively moves the radix point back to the left.

Finally, it should be noted that an exception can occur if C_t falls outside the range -128 through 127.

Binary-FLOAT to fixed-point conversions

The following material deals with numeric values only; it does not deal with “Not a Number” (NaN) or infinity.

This conversion is similar to the fixed-point-to-fixed-point conversions:

```
(24)  Nt ≃ Ns
      Substituting from (01) for the target and from (04) or (05)
      for the source:
(25)  Vt*(Rt**-St) ≃ Fs*(2**es)
      Solving for Vt, the value to be stored in the target:
(26)  Vt ≃ Fs*(2**es)*(Rt**St)
```

Since the attributes specified for the target may preclude an exact conversion, provision is made for truncation and rounding, as is done for conversion between fixed-point numbers:

```
(27)  Vi = Fs*(2**es)*(Rt**St), from (26).
      then, using equation (11) to apply constraints:
      Vt = MOD(FLOOR(Vi+H),(Bwt**Wt)), when Vi >= 0.
      Vt = MOD(CEIL(Vi-H),(Bwt**Wt)), when Vi < 0.
```

The variables H, B_{wt}, and W_t receive their values as previously described in “Rounding and truncation of fixed-point numbers” on page 157.

Notice that when the source is normalized, and not of extended-precision, the significand includes an implicit integer 1, which is not stored. Otherwise the 1 is explicit, and is stored.

When the source is denormal, the significand consists solely of the stored fraction.

Fixed-point to binary-FLOAT conversions

The following deals with nonzero values only; it does not deal with “Not a number” (NaN) or infinity.

The conversion is complicated by the fact that two values must be stored in the target—the significand and the characteristic. Portions of the following are expressed in terms of absolute values in order to facilitate the use of logarithms:

```
(28)  |N_t| ≃ |N_s|
      Substituting from (04) for the target:
(29)  |F_t*(2**e_t)| ≃ |N_s|, where:
        N_s = V_s*(R_s**-S_s), from (01).
      Temporarily letting F_t equal 1 (the minimum
      normalized significand) and taking the base-2 logarithm:
(30)  e_t ≃ log_2(|N_s|)
      Since the exponent must be an integer, we set it as:
(31)  e_t = FLOOR(log_2(|N_s|))
      Application of (31) implies that F_t may no longer be
      equal to 1, but lies in the range:
      1 <= F_t < 2
      The final approximation of F_t, with consideration for
      the sign, is determined by:
      F_t*(2**e_t) ≃ N_s, from (29).
(32)  F_t ≃ N_s*(2**-e_t)
```

This derives an exact expression for the target exponent, e_t, and an approximation of the target significand, F_t; both are expressed as functions of the effective source value, N_s. The next step is to determine the stored target characteristic, C_t, and an exact expression for F_t, allowing for possible rounding or truncation to the target significand width.

The characteristic is easily dealt with:

```
(33)  C_t = e_t+127, for single-precision.
      C_t = e_t+1023, for double-precision.
      C_t = e_t+16383, for extended-precision.
```

Using F_i as an intermediate value, the final expression of the target significand is derived by:

```
      F_i = N_s*(2**-e_t), from (32).
(34)  F_t = FLOOR((F_i*(2**W_t))+H)  * (2**-W_t), when F_i > 0.
      F_t = CEIL((F_i*(2**W_t))-H) * (2**-W_t), when F_i < 0.
      where:
           W_t = 23, for single precision.
           W_t = 52, for double precision.
           W_t = 63, for extended precision.
           H = 0, when FIT(TRUNCATE).
           H = 0.5, when FIT(ROUND).
```

The expression within the FLOOR or CEIL function effectively moves the radix point right, to the width of the target significand, and rounds the integer part of the result.

The FLOOR or CEIL function then truncates any remaining fraction. The FLOOR or CEIL function is then multiplied by a factor that effectively moves the radix point back to the left.

For a single precision target, an exception can occur if C_t is greater than 254. Also, if C_t is less than 1, then prior to application of (34):

- F_i should be divided by 2**(1-C_t).
- C_t should be set to zero.

The result will be a denormal number, or zero.

Finally, only the fractional part of F_t should be stored in the target, except in the case of extended precision, where the integer digit is also stored.

General Numeric Conversion Rules

The following sections list rules for general conversions.

Rules for retrieval of source values

- The stored value of a fixed-point field is retrieved from all digit positions across the full width of the field. All fixed-point source fields are treated as unconstrained.
- The stored significand of a floating-point field is retrieved from all bits across the full width of the significand, even though the field's PRECISION attribute may specify a precision smaller than the maximum for the field's length.

Rules for “Not a Number” (NaN), infinity, and negative zero

If the source field is a binary-FLOAT then:

1. If the target field is not a binary-FLOAT then:
    a. If the source value is “Not a Number” (NaN) then:
        - Set the target value to zero.
        - Exception ***14—Unable to convert “Not a Number” (NaN)*** occurs.
    b. If the source value is infinity then:
        - Set the target value to zero.
        - Exception ***15—Unable to convert infinity*** occurs
    c. If the source value is negative zero, then set the target value to zero.
2. If the target field is a binary-FLOAT, then encode the source value in the target field.

Rules for the BYTRVS attribute

There can be no general rule for handling this attribute. BYTRVS processing will depend on the environment in which the conversions are done and the instruction set used.

Rules for floating-point overflow and underflow

1. If the target field is a hexadecimal-FLOAT field, then:
   a. If a conversion results in an exponent less than -64 then:
      - Set the target characteristic to zero.
      - Store the target significand that results from the conversion.
      - Set exception ***13—Floating-point underflow***.

   b. If a conversion results in an exponent greater than 63 then:
      - Set the target characteristic to 127.
      - Store the target significand that results from the conversion.
      - Set exception ***5—Floating-point overflow***.
2. If the target field is a binary-FLOAT field, then:
   a. If a conversion results in an exponent greater than:
      - 127, for a single-precision target,
      - 1023, for a double-precision target, or
      - 16383, for an extended-precision target,

      then:
      - Set the target field to "Not a Number" (NaN).
      - Exception ***5—Floating-point overflow*** occurs.

Rules for the COMPLEX attribute

1. If COMPLEX(TRUE) is specified for the source field then:
   a. If COMPLEX(TRUE) is specified for the target field then:
      - The real part of the source field is assigned to the real part of the target field.
      - The imaginary part of the source field is assigned to the imaginary part of the target field.

   b. If COMPLEX(FALSE) is specified for the target field then:
      - Set the target field to zero.
      - Set exception ***18—Assignment of complex to scalar***.
2. If COMPLEX(FALSE) is specified for the source field then:
   a. If COMPLEX(FALSE) is specified for the target field, then assign the source field to the target field.

   b. If COMPLEX(TRUE) is specified for the target field then:
      - Assign the source field to the real part of the target field.
      - Set the imaginary part of the target field to zero.

Rules for target FIT attribute

If a conversion results in loss of low-order digits then:

1. If FIT(TRUNCATE) is specified for the target field, then store the result in the target field.
2. If FIT(ROUND) is specified for the target field, then round and store the result in the target field.
3. If FIT(EXACT) is specified for the target field then:
    a. If the target is a fixed-point field then:
        - Round and store the result in the target field.
        - Exception ***22—Fixed-point fit violation*** occurs.

    b. If the target is a floating-point field then:
        - Round and store the result in the target field.
        - Exception ***19—Floating-point fit violation*** occurs.

Rules for target CONSTRAINED attribute

1. If CONSTRAINED(FALSE) is specified for the target field then:
    a. If the number of significant digits required in the stored value exceeds the number of digits available in the full width of the field then:
        - Truncate the value to the full width, discarding excess high-order digits, and store the result.
        - Exception ***11—Fixed-point overflow*** occurs.
2. If CONSTRAINED(TRUE) is specified for the target field then:
    a. If the number of significant digits required in the stored value exceeds the number of digits available in the full width of the field then:
        - Truncate the value to the full width, discarding excess high-order digits, and store the result.
        - Exception ***11—Fixed-point overflow*** occurs.

    b. Else if the number of significant digits required in the stored value exceeds the number of digits specified by the PRECISION attribute then:
        - Store all significant digits.
        - Exception ***21—Fixed-point constraint violation*** occurs.

Note: Note for BINARY Targets

If RADIX(10) is specified for a BINARY field, the number of significant digits cannot be determined just by counting bits. The PRECISION attribute refers to *equivalent decimal digits*, and the equivalent number of significant digits must be determined by examining the magnitude of the stored value.

PRECISION(3) means that the absolute stored value should not exceed 999, for example.

Rules for signs

Rules for signed and unsigned numbers:

Definitions

- A **signed** number is:
  - A BINARY field with SIGNED(TRUE) specified.
  - A FLOAT field.
  - A PACKED field with SIGNED(TRUE), SGNMNS, and SGNPLS specified.
  - A ZONED field with SIGNED(TRUE) specified.
- An **unsigned** number is:
  - A BINARY field with SIGNED(FALSE) specified.
  - A PACKED field with SIGNED(TRUE) and SGNUNS specified.
  - A PACKED field with SIGNED(FALSE) specified.
  - A ZONED field with SIGNED(FALSE) specified.

Rules

1. If the source field and the target field are both of the BINARY or ENUMERATION data types, see the General Rules of "<SGNCNV attribute>" on page 123.
2. If a signed number is assigned to a signed number, then the equivalent sign of the source is encoded in the target, as appropriate for the target's data type and sign-related attributes.
3. If a signed number is assigned to an unsigned number then:
   - The absolute value of the source is assigned to the target.
   - If the source value is negative, the ***12—Assignment of negative value to unsigned field***. exception occurs.
4. If an unsigned number is assigned to an unsigned number, then there are no sign considerations.
5. If an unsigned number is assigned to a signed number, then the source is treated as positive, and the target is made set positive as appropriate for the target's data type and sign-related attributes.

Design Notes

The algorithms in this section, when dealing with BINARY data fields, require that stored values from negative sources be complemented to positive before processing. Subsequent encoding of the target should correspond with the sign of the source.

Conversion between fixed-point fields of like radix

The following explains conversion between fixed-point fields of like radix:

All cases except BINARY-RADIX(10) to BINARY-RADIX(10): The following is applicable to conversions between two BINARY fields, where RADIX(2) is specified for both fields, or between any combination of PACKED or ZONED fields. It is not applicable to conversions between two BINARY fields, where RADIX(10) is specified for both fields.

Recall approximation (08), which yields the raw target stored value, V_t:

$$V_t \simeq V_s*((R_t**S_t)/(R_s**S_s))$$

When R_t is equal to R_s, the expression can be reduced to:

$$V_t \simeq V_s*(R_t**(S_t-S_s))$$

It is apparent that V_t can be generated by performing shift operations on V_s:

1. Move the digits of V_s to an intermediate field, X. The width of X should be the greater of the effective widths of the source and target fields. The effective width of a field is determined as shown for W_t in "Rounding and truncation of fixed-point numbers" on page 157. V_s should be right-justified in X.
2. Shift X (S_t-S_s) digits; left if positive, right if negative. Zeros should be shifted in. Take note of any digits shifted out.
3. If any nonzero digits are shifted out of the right-hand end, then the target's FIT attribute becomes a concern:
   - If FIT(EXACT) is specified, then a fixed-point fit violation has occurred.
   - If FIT(EXACT) or FIT(ROUND) is specified, then X should be rounded, based on the value of the last digit shifted out.
4. If any nonzero digits are shifted out of the left-hand end, or if there are any nonzero digits to the left of the target's effective width, then a fixed-point overflow has occurred.
5. If there are any nonzero digits to the left of the target's PRECISION, and CONSTRAINED(TRUE) has been specified for the target, then a fixed-point constraint violation has occurred.
6. Truncate X on the left to the target's effective width. What remains are the digits for V_t.
7. Do not forget to properly encode the digits of X when moving them to the target field.

BINARY-RADIX(10) to BINARY-RADIX(10): The difficulty with this case is that the stored value is encoded in binary, even though RADIX(10) is specified. The result is a base-10 scaling applied to a base-2 number, and just shifting V_s will not work. V_s must first be converted to a base-10 encoding, then shifted, then converted back to a base-2 encoding:

1. Convert V_s to a base-10 encoded intermediate field, X, 20 digits wide.
2. Shift X (S_t-S_s) digits; left if positive, right if negative. Zeros should be shifted in. Take note of any digits shifted out.

3. If any nonzero digits are shifted out of the right-hand end, then the target's FIT attribute becomes a concern:
   - If FIT(EXACT) is specified, then a fixed-point fit violation has occurred.
   - If FIT(EXACT) or FIT(ROUND) is specified, then X should be rounded, based on the value of the last digit shifted out.
4. If any nonzero digits are shifted out of the left-hand end, or if there are any nonzero digits to the left of the target's effective width, then a fixed-point overflow has occurred.
5. If there are any nonzero digits to the left of the target's PRECISION, and CONSTRAINED(TRUE) has been specified for the target, then a fixed-point constraint violation has occurred.
6. Convert X to a base-2 encoded (binary) intermediate field, Y, using as many resultant digits as are necessary.
7. Truncate Y on the left to the target's effective binary width, as given by:
   - The LENGTH attribute when SIGNED(FALSE) is specified.
   - The LENGTH attribute - 1 when SIGNED(TRUE) is specified.
8. If any nonzero digits are lost as a result of the truncation, then a fixed-point overflow has occurred.
9. Y contains the digits for V_t.

Conversion from hexadecimal-FLOAT to BINARY

This conversion can also be viewed as a shifting operation:

When target is BINARY RADIX(2)

1. Move the digits of the source significand to an intermediate field, X. The width of X should be the greater of the effective widths of the source and target fields. The effective width of the target field may be determined as shown for W_t in “Rounding and truncation of fixed-point numbers” on page 157. The effective width of the source field is equal to:
   - 24, if the source is single-precision.
   - 56, if the source is double-precision.
   - 112, if the source is extended-precision.

   The digits of the source significand should be right-justified in X, and, if the source is extended-precision, should not include the digits from the secondary characteristic.
2. Set the following variables:

```
$R_s$ = 2, this will be used as a radix in a simulated binary.
$S_s$ = 4*(Y-$e_s$), a scale, where:
     Y = 6, if the source is single-precision.
     Y = 14, if the source is double-precision.
     Y = 28, if the source is extended-precision.
     $e_s$ = the source exponent.
```

3. Shift X (S_t-S_s) digits; left if positive, right if negative. Zeros should be shifted in. Take note of any digits shifted out.
4. If any nonzero digits are shifted out of the right-hand end, then the target's FIT attribute becomes a concern:
   - If FIT(EXACT) is specified, then a fixed-point fit violation has occurred.
   - If FIT(EXACT) or FIT(ROUND) is specified, then X should be rounded, based on the value of the last digit shifted out.
5. If any nonzero digits are shifted out of the left-hand end, or if there are any nonzero digits to the left of the target's effective width, then a fixed-point overflow has occurred.
6. If there are any nonzero digits to the left of the target's PRECISION, and CONSTRAINED(TRUE) has been specified for the target, then a fixed-point constraint violation has occurred.
7. Truncate X on the left to the target's effective width. What remains are the digits for V_t.

Conversion from BINARY to Hexadecimal-FLOAT

This conversion can also be viewed as a shifting operation:

When source is BINARY RADIX(2)

1. Move the bits of V_s to an intermediate 68-bit field, X. V_s should be left-justified in X.
2. Set the following variables:

```
Y  = MOD(|$S_s$|,4)
Z  = ($S_s$+Y)/4
$e_t$ = $W_t$-Z, where:
     $W_t$ = 6, if the target is single-precision.
     $W_t$ = 14, if the target is double-precision.
     $W_t$ = 28, if the target is extended-precision.
     $S_s$ = the source scale.
```

3. Repetitively shift X left until the first (W_t+Y) bits are nonzero. Note the number of shifts required.
4. Set:

   ```
   $e_t$ = $e_t$ - the number of shifts required.
   ```

   Now e_t is equal to the target exponent.
5. If e_t is less than -64, then a floating-point underflow has occurred.

6. If e_t is greater than 63, then a floating-point overflow has occurred.
7. Shift X to the right Z bits.
8. Truncate X on the right, to the width (W_t*4).
9. If any nonzero bits are lost as a result of the truncation, then the target's FIT attribute becomes a concern:
   - If FIT(EXACT) is specified, then a floating-point fit violation has occurred.
   - If FIT(EXACT) or FIT(ROUND) is specified, then X should be rounded, based on the value of the left-most bit lost as a result of the truncation.
10. X contains the bits for the target significand.
11. If the target is extended-precision, do not forget to leave space for the secondary characteristic when moving X to the target.

Specific Numeric Conversion Rules

The following sections list rules for specific conversions.

Rules for BINARY to BINARY

- "Rules for the COMPLEX attribute" on page 163
- "Rules for retrieval of source values" on page 162
- "Rules for the BYTRVS attribute" on page 162
- "Rules for target FIT attribute" on page 164
- "Rules for target CONSTRAINED attribute" on page 164
- "Rules for signs" on page 165.

Design notes

- "Conversion between fixed-point fields of like radix" on page 165.

Rules for BINARY to hexadecimal-FLOAT

- "Rules for the COMPLEX attribute" on page 163
- "Rules for retrieval of source values" on page 162
- "Rules for the BYTRVS attribute" on page 162
- "Rules for floating-point overflow and underflow" on page 163
- "Rules for target FIT attribute" on page 164
- "Rules for signs" on page 165.

Design notes

- "Conversion from BINARY to Hexadecimal-FLOAT" on page 168.

Rules for BINARY to binary-FLOAT

- "Rules for the COMPLEX attribute" on page 163
- "Rules for retrieval of source values" on page 162
- "Rules for the BYTRVS attribute" on page 162
- "Rules for floating-point overflow and underflow" on page 163
- "Rules for target FIT attribute" on page 164

- “Rules for signs” on page 165.

Rules for BINARY to PACKED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for BINARY to ZONED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for hexadecimal-FLOAT to BINARY

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion from hexadecimal-FLOAT to BINARY” on page 167.

Rules for hexadecimal-FLOAT to Hexadecimal-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for hexadecimal-FLOAT to binary-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162

- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for hexadecimal-FLOAT to PACKED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Rules for hexadecimal-FLOAT to ZONED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Rules for binary-FLOAT to BINARY

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for “Not a Number” (NaN), infinity, and negative zero” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Rules for binary-FLOAT to hexadecimal-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for “Not a Number” (NaN), infinity, and negative zero” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for binary-FLOAT to binary-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for binary-FLOAT to PACKED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162

- “Rules for the BYTRVS attribute” on page 162
- “Rules for “Not a Number” (NaN), infinity, and negative zero” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Rules for binary-FLOAT to ZONED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for “Not a Number” (NaN), infinity, and negative zero” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Rules for PACKED to BINARY

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for PACKED to hexadecimal-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for PACKED to binary-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for PACKED to PACKED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164

- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for PACKED to ZONED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for ZONED to BINARY

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for target FIT attribute” on page 164
- “Rules for target CONSTRAINED attribute” on page 164
- “Rules for signs” on page 165.

Design notes

- “Conversion between fixed-point fields of like radix” on page 165.

Rules for ZONED to hexadecimal-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for ZONED to binary-FLOAT

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for the BYTRVS attribute” on page 162
- “Rules for floating-point overflow and underflow” on page 163
- “Rules for target FIT attribute” on page 164
- “Rules for signs” on page 165.

Rules for ZONED to PACKED

- “Rules for the COMPLEX attribute” on page 163
- “Rules for retrieval of source values” on page 162
- “Rules for target FIT attribute” on page 164

- "Rules for target CONSTRAINED attribute" on page 164
- "Rules for signs" on page 165.

Design notes

- "Conversion between fixed-point fields of like radix" on page 165.

Rules for ZONED to ZONED

- "Rules for the COMPLEX attribute" on page 163
- "Rules for retrieval of source values" on page 162
- "Rules for target FIT attribute" on page 164
- "Rules for target CONSTRAINED attribute" on page 164
- "Rules for signs" on page 165.

Design notes

- "Conversion between fixed-point fields of like radix" on page 165.

SEQUENCE to SEQUENCE

These are the general rules:

1. The elements of the target are matched with the elements of the source on a qualified name basis.
2. If a qualified name of the target is not found on the source, set exception ***23—Sequence element not found***.
3. Conversions of the elements of the source sequence to the representations required by the elements of the target sequence follow the general rules specified in Figure 6 on page 136.
4. The naming of elements of the target SEQUENCE is optional. If the target element is named, then the source SEQUENCE is searched for an element with the same name. An exception occurs at conversion plan builder time if no such element is found.

   If the target element is not named, then sequence is searched for an element in the same position as the target element. If there is no element with this position, or if the source element is with this position is named, then an exception occurs at conversion plan builder time.

Appendix A. Scenarios

This appendix presents a variety of scenarios showing ADL modules and ways in which they can be used. These scenarios are not intended to include all features of the language, only those related to key concepts or uses.

Scenario 1: Calling a Conversion Plan

In this scenario, a PL/I application program calls an ADL conversion plan to perform conversions not available through PL/I. This is illustrated by Figure 7 on page 176. Why this conversion is required, and what the PL/I program does with the resulting data, are not of concern in this scenario. Many such conversions are available through ADL modules that are not available through other programming languages.

The ADL module converts a PL/I variable-length, EBCDIC-encoded character string into a PL/I fixed-length, ASCII-encoded character string with a suffix byte terminating the active data of the string.

The pertinent segment of the PL/I program is:

```
DCL X CHAR (1000) VARYING;
DCL Y CHAR (1001) /* ADL CHARSFX */;
X = 'some variable-length value';
CALL CNVCHR(X,Y);
```

The required ADL module is:

```
DECLARE BEGIN;
   X: CHARPRE MAXLEN(1000) CCSID(500);
END;
DECLARE BEGIN;
   Y: CHARSFX MAXLEN(1001) CCSID(437);
END;
CNVCHR1: PLAN (X: INPUT, Y: OUTPUT)
        BEGIN;
        Y  <- X;
        END;
```

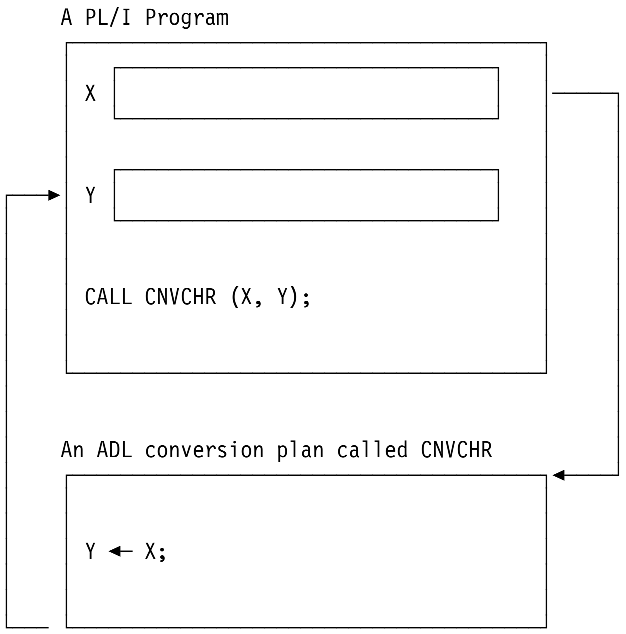


Figure 7. Calling a conversion plan

Scenario 2: Generalizing a Conversion Plan

The ADL module of the previous scenario can be generalized to handle variable-length inputs and outputs and varying CCSIDs. The caller of the CNVCHR plan is then required to provide additional information as CALL parameters.

The pertinent segment of the PL/I program is:

```
DCL X CHAR (1000) VARYING;
DCL Y CHAR (1001) /* ADL CHARSFX */;
DCL XCCSID FIXED BINARY(31);
DCL YMAXLEN FIXED BINARY(31);
DCL YCCSID FIXED BINARY(31);
X = 'some variable-length value';
XCCSID =500;
YMAXLEN = 1001;
YCCSID = 437;
CALL CNVCHR(XCCSID, X, YMAXLEN, YCCSID, Y);
```

The required ADL module is:

```
DECLARE BEGIN;
    X: CHARPRE;
    XCCSID: BINARY;
    YMAXLEN: BINARY;
    YCCSID: BINARY;
END;
DECLARE BEGIN;
    Y: CHARSFX;
END;
CNVCHR2: PLAN (
              XCCSID:  INPUT,
              X:       INPUT CCSID(XCCSID),
              YMAXLEN: INPUT,
              YCCSID:  INPUT,
              Y:       OUTPUT MAXLEN(YMAXLEN) CCSID(YCCSID))
        BEGIN;
           Y <- X;
        END;
```

Scenario 3: Calling a User-Written Program From a Plan

An ADL plan can call programs written in other programming languages to perform functions not available through ADL. One such function is the conversion of data that ADL cannot describe and convert. Other reasons for ADL plans to call user-written programs are presented in other scenarios.

In this scenario, illustrated by Figure 8 on page 180, a record contains a field that is of the WEIRD data type, which cannot be described by ADL. This field is to be converted to the STRANGE data type, which cannot be described by ADL either. Since ADL cannot describe either WEIRD or STRANGE data, these fields are described as ASIS data in ADL. And since ADL cannot perform conversions between the WEIRD and STRANGE types, a user-provided conversion program, `CNVW2S`, must be called to perform these conversions.

The pertinent segment of a PL/I program follows. How the PL/I program obtained the WEIRD data and what it will do with the STRANGE data are not of concern to this scenario.

```
DCL 1 X,
     2 A FIXED BINARY(31),
     2 B CHAR(1000) /* ADL ASIS NOTE('real data type is WEIRD') */,
     2 C CHAR(50)   /* ADL CCSID(500) */;
DCL 1 Y,
     2 A FIXED DECIMAL(9),
     2 B CHAR(1000) /* ADL ASIS NOTE('real data type is STRANGE') */,
     2 C CHAR(50)   /* ADL CCSID(437) */;
X.A = 57;
X.B = REPEAT('1A2B3C4D5E'X,100); /* the value of something WEIRD */
X.C = 'anything in CCSID 500';
CALL CNVREC(X,Y);
```

And the required ADL module is the following:

```
DECLARE BEGIN;
   X: SEQUENCE BEGIN;
      A: BINARY PRECISION(31);
      B: ASIS LENGTH(100) UNITLEN(8) NOTE('real data type is WEIRD');
      C: CHAR LENGTH(50) CCSID(500);
   END;
END;
DECLARE BEGIN;
   Y: SEQUENCE BEGIN;
      A: PACKED PRECISION(9);
      B: ASIS LENGTH(100) UNITLEN(8) NOTE('real data type is STRANGE');
      C: CHAR LENGTH(50) CCSID(437);
   END;
END;
CNVREC: PLAN (X: INPUT,
              Y: OUTPUT)
        BEGIN;
           Y.A <- X.A;
            use the following CALL statement for OS/2
           CALL '<USERDLL><CNVW2S>' (Y.B, X.B);
           use the following CALL statement for AIX
           CALL '<ddc/userexit:/u/userid/userexit><cnvw2s>' (Y.B,X.B);
           Y.C <- X.C;
        END;
```

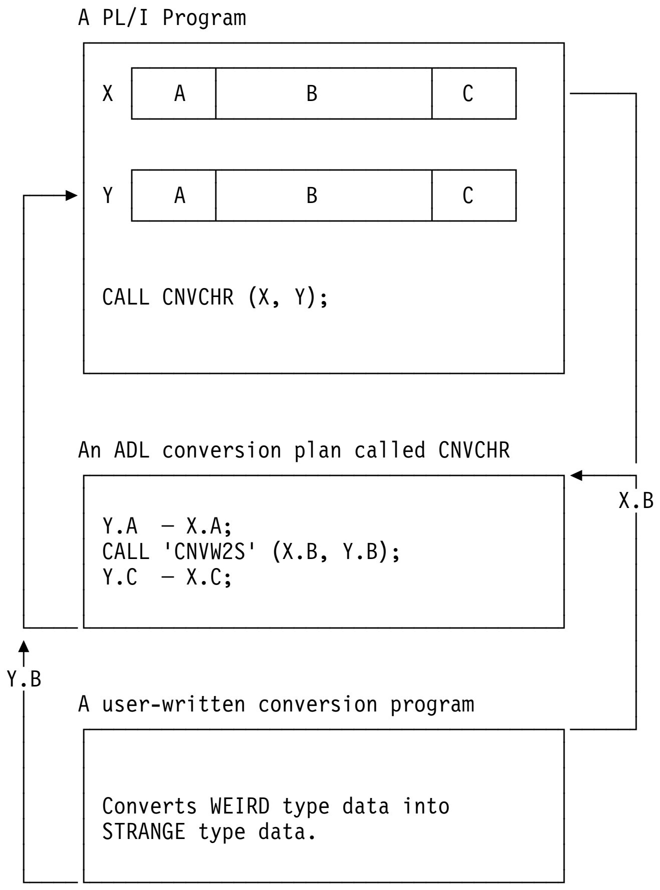

A PL/I Program
X
A
B
C
Y
A
B
C
CALL CNVCHR (X, Y);
An ADL conversion plan called CNVCHR
Y.A — X.A;
CALL 'CNVW2S' (X.B, Y.B);
Y.C — X.C;
X.B
Y.B
A user-written conversion program
Converts WEIRD type data into
STRANGE type data.

Figure 8. Calling a user-written program from a conversion plan

Scenario 4: Converting File Records by Calling ADL Plans

In this scenario, illustrated by Figure 9 on page 182, pgmA, of one representation domain, creates fileA and writes a set of records to it, each consisting of fields a, b, and c. If pgmB, of a different representation domain, is to read, process, and update those records, then the record fields must be encoded in formats a', b', and c', respectively. Therefore, the records must be converted as they are read from fileA, and they must be converted as they are rewritten to fileA. In this scenario, pgmB requests the read and write operations on fileA and calls ADL conversion plans to perform any necessary conversions.

The following ADL module, containing the pgmA and pgmB declarations of the fileA records and the getPlan and putPlan, is separately processed to produce the callable getPlan and putPlan programs.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
  Arec: SEQUENCE BEGIN;
        a: BINARY PRECISION(31) RADIX(2);
        b: CHAR LENGTH(10) CCSID(500);
        c: PACKED PRECISION(5);
        END;
   END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
  Brec: SEQUENCE BEGIN;
        a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
        b: CHAR LENGTH(10) CCSID(437);
        c: ZONED PRECISION(5);
        END;
   END;
getPlan: PLAN (Arec: INPUT,
               Brec: OUTPUT)
         BEGIN;
            Brec <- Arec;
         END;
putPlan: PLAN (Brec: INPUT,
               Arec: OUTPUT)
         BEGIN;
            Arec <- Brec;
         END;
```

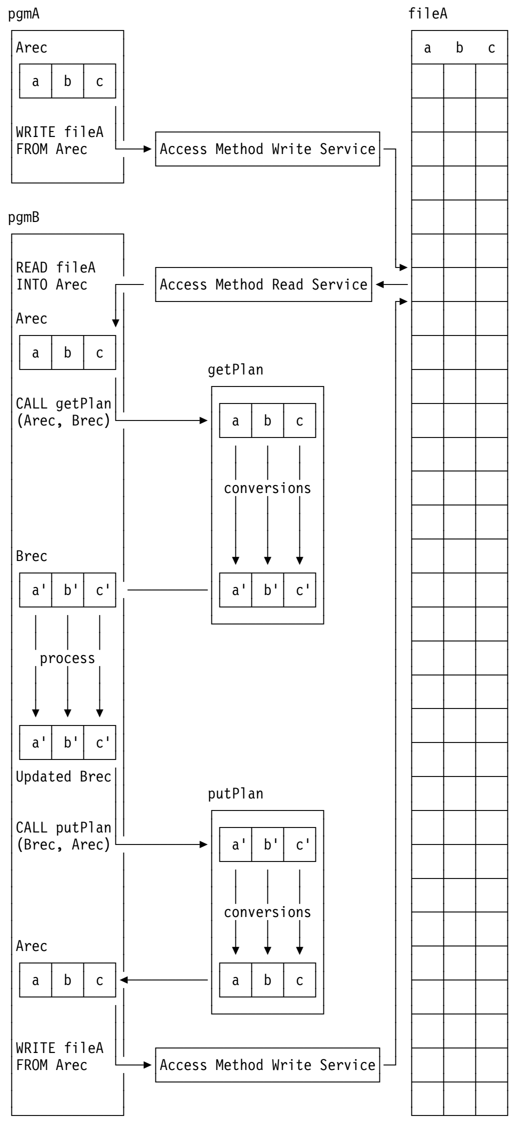


Figure 9. Converting file records by calling ADL plans

Scenario 5: Access Method Conversion of File Records

This scenario, illustrated by Figure 10 on page 185, shows an alternative to the calling of ADL plans by application programs. The goal here is for all programs that access fileA records to view its records encoded as the programs require them, as if those records were in the same representation domain as the accessing programs. Since the access method services of the file system must be used to read and write the records anyway, the Access Method services can call the ADL conversion plans as the records are read or written. Thus, to pgmB, working with fileA is no different than working with files in its own representation domain.

As in the previous scenario, the ADL module containing the declarations of pgmA and pgmB and the getPlan and putPlan ADL plans is separately processed to produce the callable getPlan and putPlan programs.

The primary advantage of this approach over that of the previous scenario is that fileA can be accessed by programs of still other representation domains without making any changes to those programs, thereby enhancing their portability between systems. However, the parameter lists of all ADL plans to be called by Access Method Services must conform to the argument lists provided by the access method services.

The following is a scenario of a module whose plans can be called by access method services. Note that in addition to the input and output records, the access method services also provide additional information about the records, including:

- The length of the input record. This allows the final field of the record to be of variable length, where the actual length is determined by the length of the stored record.
- The CCSID of the input record. This allows the declaration of character fields to be independent of the CCSID that is actually used for the record.
- The maximum allowed length of the output record.
- The required CCSID of the output record.

In addition to an OUTPUT parameter for the output record, an OUTPUT parameter is also defined to inform access method services of the actual length of the output record.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
  Arec: SEQUENCE BEGIN;
        a: BINARY PRECISION(31) RADIX(2);
        b: CHAR LENGTH(10) CCSID(0);
        c: PACKED PRECISION(5);
        END;
  Ainlen:     BINARY PRECISION(31);
  Ainccsid:   BINARY PRECISION(31);
  Aoutmaxlen: BINARY PRECISION(31);
  Aoutccsid:  BINARY PRECISION(31);
  Aoutlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
  Brec: SEQUENCE BEGIN;
        a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
        b: CHAR LENGTH(10) CCSID(0);
        c: ZONED PRECISION(5);
        END;
  Binlen:     BINARY PRECISION(31);
  Binccsid:   BINARY PRECISION(31);
  Boutmaxlen: BINARY PRECISION(31);
  Boutccsid:  BINARY PRECISION(31);
  Boutlen:    BINARY PRECISION(31);
END;
getPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Arec:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Brec:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
               Brec <- Arec;
               Boutlen <- LENGTH(Brec);
            END;
putPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Brec:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Arec:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
               Arec <- Brec;
               Aoutlen <- LENGTH(Arec);
            END;
```

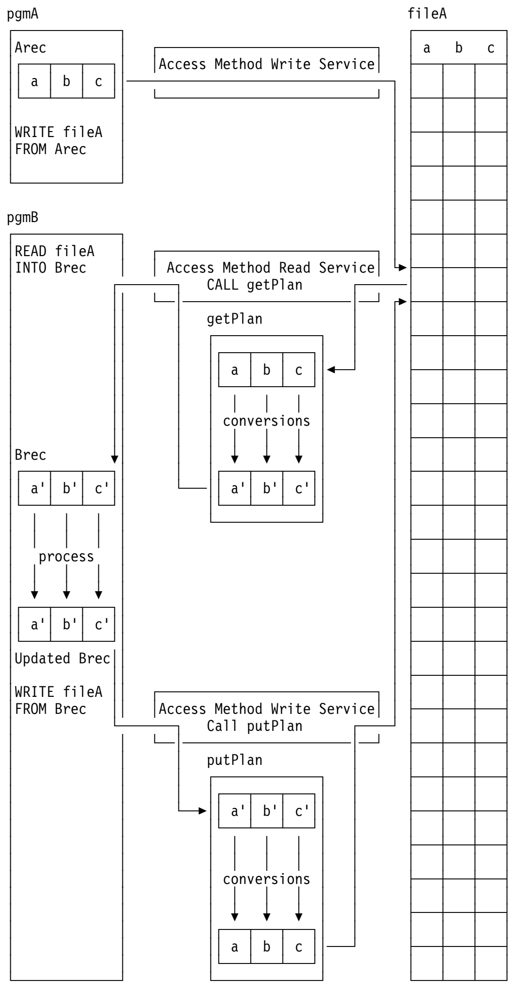


Figure 10. Access method conversion of file records

Scenario 6: View File Conversion of File Records

Using the same ADL module as in the previous scenario, conversions can be requested not by access method services, but by a pseudo-file, called a **VIEW**, whose purpose is to manage the creation, use, and integrity of ADL conversion programs. This is illustrated by Figure 11 on page 186. While this scenario may or may not apply to data other than file data, it serves to illustrate an important aspect of ADL; namely that the declarations and plans of a module can be managed as separate entities and brought together only when it is necessary to create conversion programs, as in Figure 12 on page 187.

The ADL declaration of fileA's records can be stored as an attribute of fileA and easily accessed whenever it is required by the data description utility (DDU) and whenever it is needed to create a view file. Similarly, the ADL declaration of viewB's view of fileA's record can be stored as an attribute of viewB. If a programmer defines ADL plans, they too can be stored as attributes of viewB, but if none are defined, then generic ADL plans for file record conversion can be supplied by default.

Other benefits can also be derived from the concept of view files, such as the management and use of keyed access paths to file records, but these are outside the scope of this scenario.

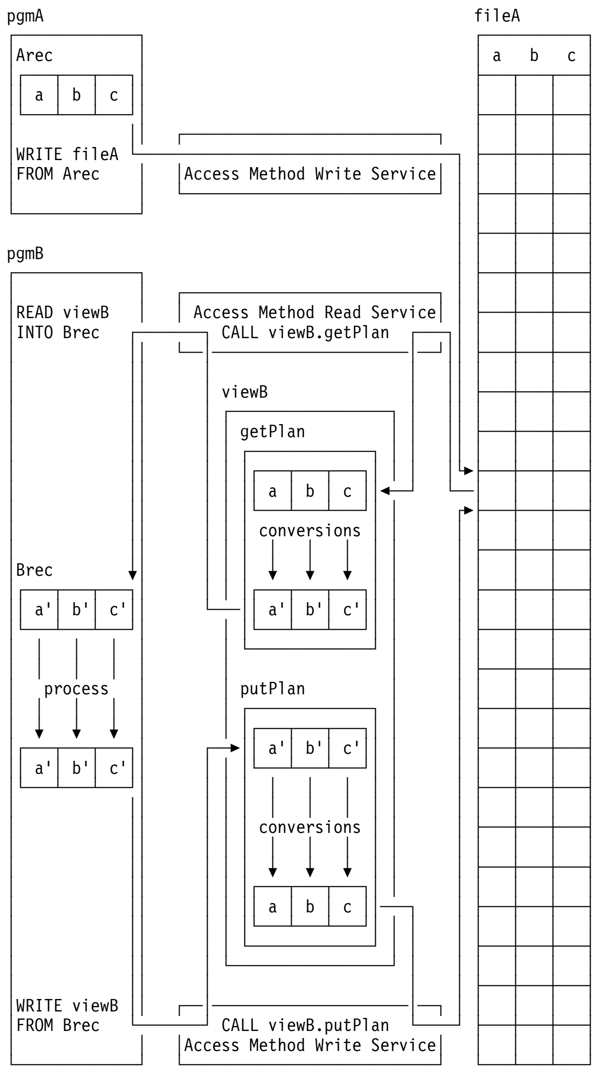


Figure 11. View file conversion of file records

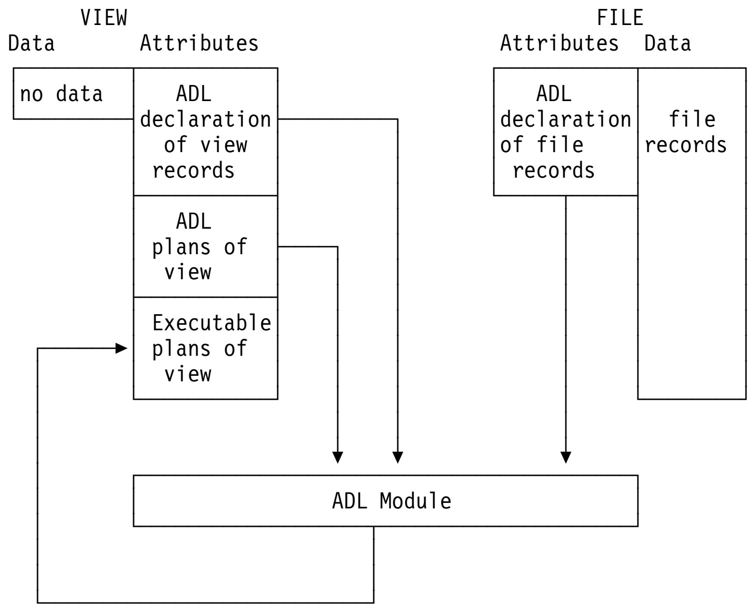


Figure 12. Assembling an ADL module from file and view attributes

Scenario 7: Selecting and Reordering Fields of Records

In this scenario, pgmB does not require all of the fields written to fileA by pgmA, and pgmB needs the fields it selects in a different order than provided by pgmA. In the following module, only the declaration of Brec differs from earlier scenarios. Here, the Brec SEQUENCE includes only the fields required by pgmB, in the order required by pgmB. No changes are needed to the ADL plans, although they result in different conversion programs because of the changes to Brec. In the getPlan, the assignment of Arec to Brec causes A.c and A.a to be copied with conversions to their Brec counterparts. Field A.b is ignored. And in the PutPlan, the assignment of Brec to Arec causes B.c and B.a to be copied with conversions to their Arec counterparts. However, since field A.rec.b is not matched by a field named B.rec.b, an exception is raised and the PutPlan is terminated. This prevents the file from being contaminated with incomplete records. Thus, field selection is not allowed when writing or updating file records.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
  Arec: SEQUENCE BEGIN;
        a: BINARY PRECISION(31) RADIX(2);
        b: CHAR LENGTH(10) CCSID(0);
        c: PACKED PRECISION(5);
        END;
  Ainlen:     BINARY PRECISION(31);
  Ainccsid:   BINARY PRECISION(31);
  Aoutmaxlen: BINARY PRECISION(31);
  Aoutccsid:  BINARY PRECISION(31);
  Aoutlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
  Brec: SEQUENCE BEGIN;
        c: ZONED PRECISION(5);
        a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
        END;
  Binlen:     BINARY PRECISION(31);
  Binccsid:   BINARY PRECISION(31);
  Boutmaxlen: BINARY PRECISION(31);
  Boutccsid:  BINARY PRECISION(31);
  Boutlen:    BINARY PRECISION(31);
END;

getPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Arec:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Brec:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
               Brec <- Arec;
               Boutlen <- LENGTH(Brec);
            END;
putPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Brec:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Arec:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
               Arec <- Brec;
               Aoutlen <- LENGTH(Arec);
            END;
```

Scenario 8: Updating File Records Using Workspace Variables

In this scenario, pgmB reads the records of fileA and updates only selected fields of each record. The getPlan converts all Arec fields to their corresponding Brec fields, including Arec fields that will not be updated by pgmB. If a standard putPlan is used, then all Brec fields are converted to their corresponding Arec fields, including those that were not updated by pgmB. Thus, the non-update fields are converted once for use by pgmB and then reconverted for replacement in fileA. However, not all conversions are fully reversible. For example, not all CCSID pairs allow conversions in both directions, and some numeric conversions lose precision in conversions and the conversions are, therefore, not reversible. Therefore, the non-update fields have been corrupted from their original values.

In this module, a special getPlan and a special putPlan are provided to prevent the corruption of non-updated fields. The getPlan copies the original values of the non-updated fields into ADL workspace variables aHold and bHold, and the putPlan uses the workspace values to restore the original values of the non-updated fields.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
  Arec: SEQUENCE BEGIN;
        a: BINARY PRECISION(31) RADIX(2);
        b: CHAR LENGTH(10) CCSID(0);
        c: PACKED PRECISION(5);
        END;
  aHold: BINARY PRECISION(31) RADIX(2); /* Used as workspace variable */
  bHold: CHAR LENGTH(10) CCSID(0);      /* Used as workspace variable */
  Ainlen:     BINARY PRECISION(31);
  Ainccsid:   BINARY PRECISION(31);
  Aoutmaxlen: BINARY PRECISION(31);
  Aoutccsid:  BINARY PRECISION(31);
  Aoutlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
 Brec: SEQUENCE
       BEGIN;  /* fields a and b are not updated */
       a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
       b: CHAR LENGTH(10) CCSID(0);
       c: ZONED PRECISION(5);
       END;
  Binlen:     BINARY PRECISION(31);
  Binccsid:   BINARY PRECISION(31);
  Boutmaxlen: BINARY PRECISION(31);
  Boutccsid:  BINARY PRECISION(31);
  Boutlen:    BINARY PRECISION(31);
END;
getPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Arec:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Brec:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
                                     /* Using workspace variables:    */
               aHold <- Arec.a;      /* Save original value of Arec.a */
               bHold <- Arec.b;      /* Save original value of Arec.b */
               Brec <- Arec;
               Boutlen <- LENGTH(Brec);
            END;
```

```
putPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Brec:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Arec:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
                                 /* Using workspace variables:       */
               Arec.a <- aHold;   /* Restore original value of Arec.a */
               Arec.b <- bHold;   /* Restore original value of Arec.b */
               Arec.c <- Brec.c;
               Aoutlen <- LENGTH(Arec);
            END;
```

Scenario 9: Converting Record Keys

Many files have an index that associates the values of certain key fields with individual records of the file. Programs can then access records randomly by specifying the key of each record or they can access them consecutively by ascending or descending key values. Programs accessing records randomly by key specify key values in terms of their own representation domain, and these keys must be converted to the representation domain of the file. Programs can also request that the keys of particular records be returned to them, and these keys must be converted from the representation domain of the file to that of the program.

In this scenario, an ADL module provides both record and key declarations for the file's representation domain and the program's representation domain. It also provides plans for converting records and keys.

```
DECLARE BEGIN; /* declarations corresponding to fileA fields */
   Arec: SEQUENCE BEGIN;
         a: BINARY PRECISION(31) RADIX(2);
         b: CHAR LENGTH(10) CCSID(0);
         c: PACKED PRECISION(5);
         END;
   Akey: SEQUENCE BEGIN;
         a: BINARY PRECISION(31) RADIX(2);
         c: PACKED PRECISION(5);
         END;
   Ainlen:     BINARY PRECISION(31);
   Ainccsid:   BINARY PRECISION(31);
   Aoutmaxlen: BINARY PRECISION(31);
   Aoutccsid:  BINARY PRECISION(31);
   Aoutlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
   Brec: SEQUENCE
         BEGIN;  /* fields a and b are not updated */
         a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
         b: CHAR LENGTH(10) CCSID(0);
         c: ZONED PRECISION(5);
         END;
   Bkey: SEQUENCE
         BEGIN;
         a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
         c: ZONED PRECISION(5);
         END;
   Binlen:     BINARY PRECISION(31);
   Binccsid:   BINARY PRECISION(31);
   Boutmaxlen: BINARY PRECISION(31);
   Boutccsid:  BINARY PRECISION(31);
   Boutlen:    BINARY PRECISION(31);
END;
getPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Arec:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Brec:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
               Brec <- Arec;
               Boutlen <- LENGTH(Brec);
            END;
putPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Brec:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Arec:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
               Arec <- Brec;
               Aoutlen <- LENGTH(Arec);
            END;
```

```
/* The following plan converts keys specified by pgmB to fileA */
keyPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Bkey:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Akey:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
               Akey <- Bkey;
               Aoutlen <- LENGTH(Akey);
            END;
/* The following plan converts keys fed back to pgmB from fileA */
kfbPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Akey:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Bkey:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
               Bkey <- Akey;
               Boutlen <- LENGTH(Bkey);
            END;
```

Scenario 10: Converting Files of Text Records

If a file consists solely of text records, as many do, then the following ADL module can be used to convert the text from one CCSID to another CCSID. Note that the same ADL plans are used as for more complex records.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
   Arec: CHAR LENGTH(*) MAXLEN(100);
   Ainlen:     BINARY PRECISION(31);
   Ainccsid:   BINARY PRECISION(31);
   Aoutmaxlen: BINARY PRECISION(31);
   Aoutccsid:  BINARY PRECISION(31);
   Aoutlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN;
   Brec: CHAR LENGTH(*) MAXLEN(100);
   Binlen:     BINARY PRECISION(31);
   Binccsid:   BINARY PRECISION(31);
   Boutmaxlen: BINARY PRECISION(31);
   Boutccsid:  BINARY PRECISION(31);
   Boutlen:    BINARY PRECISION(31);
END;
getPlan: PLAN (Ainlen:     INPUT,
               Ainccsid:   INPUT,
               Arec:      INPUT LENGTH(Ainlen) CCSID(Ainccsid),
               Aoutmaxlen: INPUT,
               Aoutccsid:  INPUT,
               Brec:      OUTPUT MAXLEN(Aoutmaxlen) CCSID(Aoutccsid),
               Boutlen:    OUTPUT)
            BEGIN;
               Brec <- Arec;
               Boutlen <- LENGTH(Brec);
            END;
putPlan: PLAN (Binlen:     INPUT,
               Binccsid:   INPUT,
               Brec:      INPUT LENGTH(Binlen) CCSID(Binccsid),
               Boutmaxlen: INPUT,
               Boutccsid:  INPUT,
               Arec:      OUTPUT MAXLEN(Boutmaxlen) CCSID(Boutccsid),
               Aoutlen:    OUTPUT)
            BEGIN;
               Arec <- Brec;
               Aoutlen <- LENGTH(Arec);
            END;
```

Scenario 11: Defining Default Plans

As has been seen in many of the preceding scenarios, standard ADL plans can be used for most get and put operations on file records. Specialized plans are only required when additional functions are required. Therefore, it would be desirable to be able to define a set of plans that could be used when user-written plans are not provided; and indeed to eliminate the need for most user-written plans.

The concept of <positional identifier>s was incorporated in ADL for this purpose. If the DECLARE statements for the representation domains of a file and a view are included in a module in precisely that order, then they can be referred to as the first DECLARE statement and the second DECLARE statement, or with positional identifiers `1` and `2`, respectively. And if the SEQUENCE declaration of the record appears as the first declaration in each DECLARE statement, then their fully qualified positional identifiers are "1"."1" and "2"."1", respectively.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
            /* <positional identifier> "1" within the module*/
   Arec: SEQUENCE BEGIN;
        a: BINARY PRECISION(31) RADIX(2);
        b: CHAR LENGTH(10) CCSID(0);
        c: PACKED PRECISION(5);
        END;
   inlen:     BINARY PRECISION(31);
   inccsid:   BINARY PRECISION(31);
   outmaxlen: BINARY PRECISION(31);
   outccsid:  BINARY PRECISION(31);
   outlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
            /* <positional identifier> "2" within the module*/
   Brec: SEQUENCE BEGIN;
        a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
        b: CHAR LENGTH(10) CCSID(0);
        c: ZONED PRECISION(5);
        END;
   inlen:     BINARY PRECISION(31);
   inccsid:   BINARY PRECISION(31);
   outmaxlen: BINARY PRECISION(31);
   outccsid:  BINARY PRECISION(31);
   outlen:    BINARY PRECISION(31);
END;


getPlan: PLAN ("1".inlen:     INPUT,
               "1".inccsid:   INPUT,
               "1"."1":       INPUT
                              LENGTH("1".inlen)
                              CCSID("1".inccsid),
               "1".outmaxlen: INPUT,
               "1".outccsid:  INPUT,
               "2"."1":       OUTPUT
                              MAXLEN("1".outmaxlen)
                              CCSID("1".outccsid),
               "2".outlen:    OUTPUT)
            BEGIN;
               "2"."1" <- "1"."1";
               "2".outlen <- LENGTH("2"."1");
            END;

putPlan: PLAN ("2".inlen:     INPUT,
               "2".inccsid:   INPUT,
               "2"."1":       INPUT
                              LENGTH("2".inlen)
                              CCSID("2".inccsid),
               "2".outmaxlen: INPUT,
               "2".outccsid:  INPUT,
               "1"."1":       OUTPUT
                              MAXLEN("2".outmaxlen)
                              CCSID("2".outccsid),
               "1".outlen:    OUTPUT)
            BEGIN;
               "1"."1" <- "2"."1";
               "1".outlen <- LENGTH("1"."1");
            END;
```

Scenario 12: Multiformat Files

In this scenario, pgmA has written records of two different formats to fileA. The order and frequency of these records is unpredictable by readers of the file, but each record has a discriminator field that serves to identify its format. The ADL declaration of the records of fileA clearly must accommodate both record formats and must include a facility for discriminating between them. Similarly, each view of the file must also accommodate them such that ADL plans only convert like formats.

In the following declarations, the ADL CASE data type is used to distinguish between the record formats. No change is needed to the ADL conversion plans. The assignment of Arec to Brec determines the format of an Arec record, matches it with the equivalent format of the Brec record, and performs appropriate conversions.

```
DECLARE BEGIN; /* declarations corresponding to pgmA variables */
   Arec: CASE BEGIN;
         WHEN rf1.a = 1
         THEN
         rf1: SEQUENCE BEGIN;
              a: BINARY PRECISION(31) RADIX(2);
              b: CHAR LENGTH(10) CCSID(0);
              c: PACKED PRECISION(5);
              END;
         WHEN rf2.a = 2
         THEN
         rf2: SEQUENCE BEGIN;
              a: BINARY PRECISION(31) RADIX(2);
              b: CHAR LENGTH(10) CCSID(0);
              c: BIT LENGTH(24);
              END;
         END;
   inlen:     BINARY PRECISION(31);
   inccsid:   BINARY PRECISION(31);
   outmaxlen: BINARY PRECISION(31);
   outccsid:  BINARY PRECISION(31);
   outlen:    BINARY PRECISION(31);
END;
DECLARE BEGIN; /* declarations corresponding to pgmB variables */
   Brec: CASE BEGIN;
         WHEN rf1.a = 1
         THEN
         rf1: SEQUENCE BEGIN;
              a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
              b: CHAR LENGTH(10) CCSID(0);
              c: ZONED PRECISION(5);
              END;
         WHEN rf2.a = 2
         THEN
         rf2: SEQUENCE BEGIN;
              a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
              b: CHAR LENGTH(10) CCSID(0);
              c: BIT LENGTH(24);
              END;
         END;
   inlen:     BINARY PRECISION(31);
   inccsid:   BINARY PRECISION(31);
   outmaxlen: BINARY PRECISION(31);
   outccsid:  BINARY PRECISION(31);
   outlen:    BINARY PRECISION(31);
END;
```

```
getPlan: PLAN ("1".inlen:     INPUT,
               "1".inccsid:   INPUT,
               "1"."1":       INPUT
                              LENGTH("1".inlen)
                              CCSID("1".inccsid),
               "1".outmaxlen: INPUT,
               "1".outccsid:  INPUT,
               "2"."1":       OUTPUT
                              MAXLEN("1".outmaxlen)
                              CCSID("1".outccsid),
               "2".outlen:    OUTPUT)
            BEGIN;
               "2"."1" <- "1"."1";
               "2".outlen <- LENGTH("2"."1");
            END;
putPlan: PLAN ("2".inlen:     INPUT,
               "2".inccsid:   INPUT,
               "2"."1":       INPUT
                              LENGTH("2".inlen)
                              CCSID("2".inccsid),
               "2".outmaxlen: INPUT,
               "2".outccsid:  INPUT,
               "1"."1":       OUTPUT
                              MAXLEN("2".outmaxlen)
                              CCSID("2".outccsid),
               "1".outlen:    OUTPUT)
            BEGIN;
               "1"."1" <- "2"."1";
               "1".outlen <- LENGTH("1"."1");
            END;
```

Scenario 13: Converting Program Call Parameters

In this scenario, illustrated by Figure 13 on page 201, a program, `pgmA`, of one representation domain, needs to call a program, `pgmB` , of another representation domain. However, the encodings of the arguments specified by pgmA do not match the encodings of the parameters required by pgmB. The following ADL module can be defined by programmers or by appropriate software engineering tools to convert arguments as they flow to pgmB, and to convert values to be returned to pgmA.

```
A: DECLARE BEGIN; /* declarations corresponding to pgmA variables */
   a: BINARY PRECISION(31) RADIX(2);
   b: CHAR LENGTH(10) CCSID(500);
END;
C: DECLARE BEGIN; /* declarations corresponding to pgmA variables */
   c: PACKED PRECISION(5);
   d: ARRAY DMNLST(DMNSIZE(5)) OF PACKED PRECISION(9);
   END;
B: DECLARE BEGIN; /* declarations corresponding to pgmB variables */
   a: BINARY PRECISION(5) RADIX(10) CONSTRAINED(TRUE);
   b: CHAR LENGTH(10) CCSID(437);
   c: ZONED PRECISION(5);
   d: ARRAY DMNLST(DMNSIZE(5)) OF ZONED PRECISION(9);
   END;
gluePlan: PLAN (A.a: INPUT,
                A.b: INPUT,
                C.c: OUTPUT,
                C.d: OUTPUT)
          BEGIN;
             B.a <- A.a;
             B.b <- A.b;
             use the followng CALL statement for OS/2:
             CALL '<USERDLL><pgmB>' (B.a, B.b, B.c, B.d);
             use the following CALL statement for AIX:
             CALL '<search path><pgmB>' (B.a, B.b, B.c, B.d);
             C.c <- B.c;
             C.d <- B.d;
          END;
```

In this scenario,

- pgmA calls the ADL plan, named gluePlan, passing it the addresses of input parameters A.a and A.b, and of output parameters A.c and A.d.
- The module workspace includes space for local variables B.a, B.b, B.c and B.d
- gluePlan assigns input parameters A.a and A.b to B.a and B.b, respectively, thereby converting them to the encodings required by pgmB.
- gluePlan calls pgmB, passing it the converted input parameters and providing it with addressability to local variables corresponding to pgmB's encodings of its output parameters.

- On return from pgmB, gluePlan assigns output parameters B.c and B.d to its output parameters, A.c and A.d, respectively, thereby converting them to the encodings required by pgmA.
- gluePlan returns to its caller, pgmA.

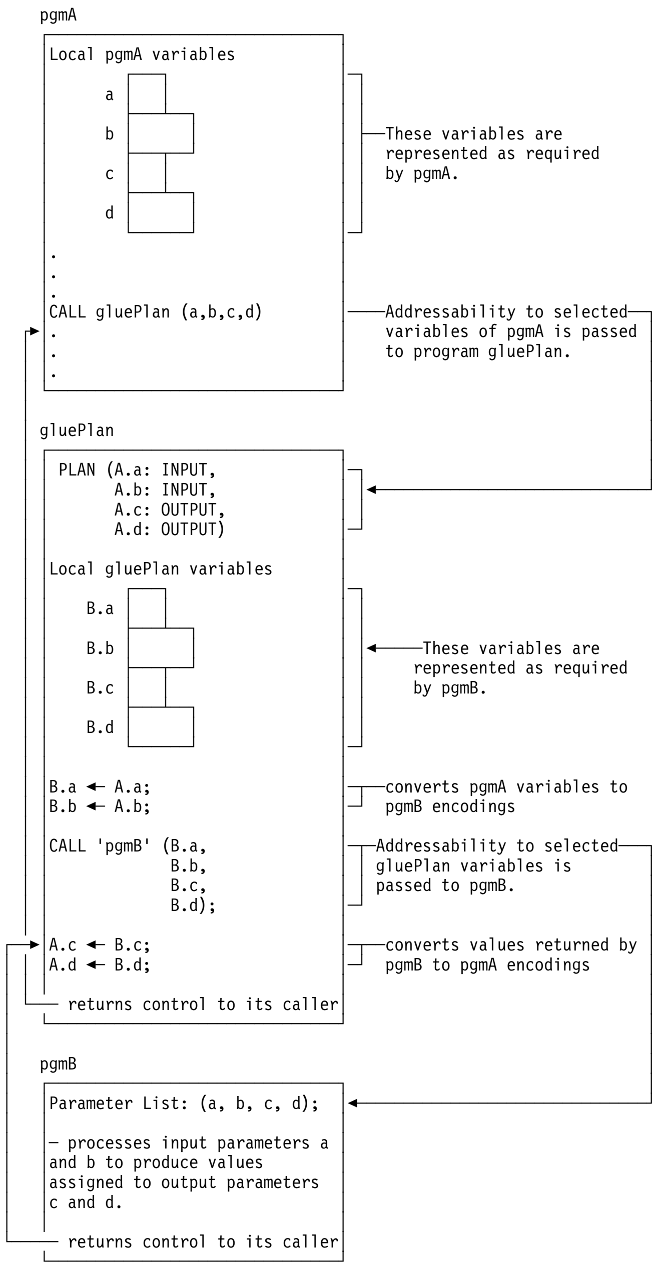

pgmA
Local pgmA variables
a
b
c
d
These variables are represented as required by pgmA.
CALL gluePlan (a,b,c,d)
Addressability to selected variables of pgmA is passed to program gluePlan.
gluePlan
PLAN (A.a: INPUT,
A.b: INPUT,
A.c: OUTPUT,
A.d: OUTPUT)
Local gluePlan variables
B.a
B.b
B.c
B.d
These variables are represented as required by pgmB.
B.a ← A.a;
B.b ← A.b;
converts pgmA variables to pgmB encodings
CALL 'pgmB' (B.a,
B.b,
B.c,
B.d);
Addressability to selected gluePlan variables is passed to pgmB.
A.c ← B.c;
A.d ← B.d;
converts values returned by pgmB to pgmA encodings
returns control to its caller
pgmB
Parameter List: (a, b, c, d);
– processes input parameters a and b to produce values assigned to output parameters c and d.
returns control to its caller

Figure 13. Data conversions during program calls

Appendix B. The DD&C User Exit

DD&C provides a user-exit facility that allows you to call your own programs from within ADL source files. The following describes the conventions you must respect when making calls to your own programs.

The calling conventions for the DD&C user exit are as follows:

1. To call your program, you insert a <CALL statement> in the ADL source file.
2. The program name must be an ADL <character literal>.
3. For AIX, your program should be an executable file and it may be either a share or nonshare object. You must specify both the search path name and the object file name to load the correct program and execute the function. Therefore, the character literal in the CALL statement must have the following format:

   ```
   <search path name><object file name>
   ```

   where: search path name is a string containing one or more directory path names separated by a colon. If the object file name is not found, the search continues, using the search path specified. The first instance of the object file name found is used.

   ```
   The search path name must be enclosed within angle brackets (< >).
   ```

   object file name is the name of the object file to be loaded. if the object file name contains no / (slash) symbols, it is treated as a base name, and should be in one of the directories listed in the search path name. If the object file name is not a base name (if it contains at least one / character), the name is used as it is, and no library path searches are performed to locate the object file (essentially, search path name will get ignored).

   ```
   The object file name must be enclosed within angle brackets (< >).
   ```

   Examples of valid CALL statements are:

   ```
   CALL '</u/myid/tools:/ddc/tools><myuserexit>' (parm1, parm2);
   or
   CALL '</usr/include>'</ddc/tools/myuserexit>' (parm1, parm2);
   ```

4. For OS/2, your program must be coded as a function contained in a dynamic link library (DLL). You must specify both the library name and the function name to load the correct DLL and execute the DLL function. Therefore, the character literal in the CALL statement must have the following format:

   <*libraryname*><*functionname*>

   Where:

   libraryname Is the filename of the user-provided DLL, without the ".DLL" extension.

   Alternatively, you can specify the fully-qualified filename of the DLL, complete with the path and the ".DLL" extension.

The library name must be enclosed within angle brackets (**<** **>**).

functionname	Is the name of the function to call in the DLL.
	The function name must be enclosed within angle brackets (**<** **>**).

Examples of valid CALL statements are:

```
CALL '<MYDLL><MyFunction>' (parameter1, parameter2);
```

or

```
CALL '<C:\MYPATH\MYDLL.DLL><MyFunction>' (parameter1, parameter2);
```

The user-provided program is called at conversion plan execution time. SMARTdata UTILITIES for AIX: Data Description and Conversion describes the format of the call to the user-provided program and includes an example of a user-exit program.

Appendix C. List of CCSID Values

lists the CCSID values that can be specified in DD&C. This list is updated periodically.

Note: Not all combinations of CCSID values are possible.

CCSID	Description	CCSID	Description
37	COM EUROPE EBCDIC	861	ICELAND PC-DATA
256	NETHERLAND EBCDIC	862	HEBREW PC-DATA
259	SYMBOLS SET 7	863	CANADA PC-DATA
273	AUS/GERM EBCDIC	864	ARABIC PC-DATA
277	DEN/NORWAY EBCDIC	865	DEN/NORWAY PC-DAT
278	FIN/SWEDEN EBCDIC	866	CYRILLIC PC-DATA
280	ITALIAN EBCDIC	868	URDU PC-DATA
282	PORTUGAL EBCDIC	869	GREEK PC-DATA
284	SPANISH EBCDIC	870	ROECE EBCDIC
285	UK EBCDIC	871	ICELAND EBCDIC
286	AUS/GER 3270 EBCD	874	THAI PC-DISPLAY
290	JAPANESE EBCDIC	875	GREEK EBCDIC
297	FRENCH EBCDIC	880	CYRILLIC EBCDIC
300	JAPAN DB PC-DATA	891	KOREA SB PC-DATA
301	JAPAN DB PC-DATA	895	JAPAN 7-BIT LATIN
367	US ANSI X3.4 ASCI	896	JAPAN 7-BIT KATAK
420	ARABIC EBCDIC	897	JAPAN SB PC-DATA
421	MAGHR/FREN EBCDIC	899	SYMBOLS - PC
423	GREEK EBCDIC	903	S-CHINESE PC-DATA
424	HEBREW EBCDIC	904	T-CHINESE PC-DATA
437	USA PC-DATA	905	TURKEY EBCDIC
500	INTL EBCDIC	912	ISO 8859-2 ASCII
803	HEBREW EBCDIC	915	ISO 8859-5 ASCII
813	GREEK/LATIN ASCII	916	ISO 8859-8 ASCII
819	ISO 8859-1 ASCII	918	URDU EBCDIC
833	KOREAN EBCDIC	920	ISO 8859-9 ASCII
834	KOREAN DB EBCDIC	926	KOREA DB PC-DATA
835	T-CHINESE DB EBCD	927	T-CHINESE PC-DATA
836	S-CHINESE EBCDIC	928	S-CHINESE PC-DATA
837	S-CHINESE EBCDIC	929	THAI DB PC-DATA
838	THAILAND EBCDIC	930	JAPAN MIX EBCDIC
839	THAI DB EBCDIC	931	JAPAN MIX EBCDIC
850	LATIN-1 PC-DATA	932	JAPAN MIX PC-DATA
851	GREEK PC-DATA	933	KOREA MIX EBCDIC
852	ROECE PC-DATA	934	KOREA MIX PC-DATA
853	TURKISH PC-DATA	935	S-CHINESE MIX EBC
855	CYRILLIC PC-DATA	936	S-CHINESE PC-DATA
856	HEBREW PC-DATA	937	T-CHINESE MIX EBC
857	TURKISH PC-DATA	938	T-CHINESE MIX PC
860	PORTUGESE PC-DATA	939	JAPAN MIX EBCDIC

CCSID	Description
942	JAPAN MIX PC-DATA
944	KOREA MIX PC-DATA
946	S-CHINESE PC-DATA
948	T-CHINESE PC-DATA
949	KOREA KS PC-DATA
951	IBM KS PC-DATA
1008	ARABIC ISO/ASCII
1010	FRENCH ISO-7 ASCI
1011	GERM ISO-7 ASCII
1012	ITALY ISO-7 ASCII
1013	UK ISO-7 ASCII
1014	SPAIN ISO-7 ASCII
1015	PORTUGAL ISO7 ASC
1016	NOR ISO-7 ASCII
1017	DENMK ISO-7 ASCII
1018	FIN/SWE ISO-7 ASC
1019	BELG/NETH ASCII
1020	CANADA ISO-7
1021	SWISS ISO-7
1023	SPAIN ISO-7
1025	CYRILLIC EBCDIC
1026	TURKEY LATIN-5 EB
1027	JAPAN LATIN EBCD
1040	KOREA PC-DATA
1041	JAPAN PC-DATA
1042	S-CHINESE PC-DATA
1043	T-CHINESE PC-DATA
1046	ARABIC - PC
1047	LATIN OPEN SYS EB
1051	HP EMULATION
1088	KOREA KS PC-DATA
1089	ARABIC ISO 8859-6
1097	FARSI EBCDIC
1098	FARSI - PC
1100	MULTI EMULATION
1101	BRITISH ISO-7 NRC
1102	DUTCH ISO-7 NRC
1103	FINNISH ISO-7 NRC
1104	FRENCH ISO-7 NRC
1105	NOR/DAN ISO-7 NRC
1106	SWEDISH ISO-7 NRC
1107	NOR/DAN ISO-7 NRC
4133	USA EBCDIC
4369	AUS/GERMAN EBCDIC
4370	BELGIUM EBCDIC

CCSID	Description
4371	BRAZIL EBCDIC
4372	CANADA EBCDIC
4373	DEN/NORWAY EBCDIC
4374	FIN/SWEDEN EBCDIC
4376	ITALY EBCDIC
4378	PORTUGAL EBCDIC
4380	LATIN EBCDIC
4381	UK EBCDIC
4386	JAPAN EBCDIC SB
4393	FRANCE EBCDIC
4396	JAPAN EBCDIC DB
4516	ARABIC EBCDIC
4519	GREEK EBCDIC 3174
4520	HEBREW EBCDIC
4533	SWISS PC-DATA
4596	LATIN AMER EBCDIC
4929	KOREA SB EBCDIC
4932	S-CHIN SB EBCDIC
4934	THAI SB EBCDIC
4946	LATIN-1 PC-DATA
4947	GREEK PC-DATA
4948	LATIN-2 PC-DATA
4949	TURKEY PC-DATA
4951	CYRILLIC PC-DATA
4952	HEBREW PC-DATA
4953	TURKEY PC-DATA
4960	ARABIC PC-DATA
4964	URDU PC-DATA
4965	GREEK PC-DATA
4966	ROECE LATIN-2 EBC
4967	ICELAND EBCDIC
4970	THAI SB PC-DATA
4976	CYRILLIC EBCDIC
4993	JAPAN SB PC-DATA
5014	URDU EBCDIC
5026	JAPAN MIX EBCDIC
5028	JAPAN MIX PC-DATA
5029	KOREA MIX EBCDIC
5031	S-CH MIXED EBCDIC
5033	T-CHINESE EBCDIC
5035	JAPAN MIX EBCDIC
5045	KOREA KS PC-DATA
5047	KOREA KS PC DATA
5143	LATIN OPEN SYS
8229	INTL EBCDIC

CCSID	Description
8448	INTL EBCDIC
8476	SPAIN EBCDIC
8489	FRANCE EBCDIC
8612	ARABIC EBCDIC
8629	AUS/GERM PC-DATA
8692	AUS/GERMAN EBCDIC
9025	KOREA SB EBCDIC
9026	KOREA DB EBCDIC
9047	CYRILLIC PC-DATA
9056	ARABIC PC-DATA
9060	URDU PC-DATA
9089	JAPAN PC-DATA SB
9122	JAPAN MIX EBCDIC
9124	JAPAN MIX PC-DATA
9125	KOREA MIX EBCDIC
12325	CANADA EBCDIC
12544	FRANCE EBCDIC
12725	FRANCE PC-DATA
12788	ITALY EBCDIC
13152	ARABIC PC-DATA
13218	JAPAN MIX EBCDIC
13219	JAPAN MIX EBCDIC
13221	KOREA MIX EBCDIC
16421	CANADA EBCDIC
16821	ITALY PC-DATA
16884	FIN/SWEDEN EBCDIC
20517	PORTUGAL EBCDIC
20917	UK PC-DATA
20980	DEN/NORWAY EBCDIC
24613	INTL EBCDIC
24877	JAPAN DB PC-DISPL
25013	USA PC-DISPLAY
25076	DEN/NORWAY EBCDIC
25426	LATIN-1 PC-DISP
25427	GREECE PC-DISPLAY
25428	LATIN-2 PC-DISP
25429	TURKEY PC-DISPLAY
25431	CYRILLIC PC-DISP
25432	HEBREW PC-DISPLAY
25433	TURKEY PC-DISPLAY
25436	PORTUGAL PC-DISP
25437	ICELAND PC-DISP
25438	HEBREW PC-DISPLAY
25439	CANADA PC-DISPLAY
25440	ARABIC PC-DISPLAY

CCSID	Description
25441	DEN/NOR PC-DISP
25442	CYRILLIC PC-DISP
25444	URDU PC-DISPLAY
25445	GREECE PC-DISPLAY
25450	THAILAND PC-DISP
25467	KOREA SB PC-DISP
25473	JAPAN SB PC-DISP
25479	S-CHIN SB PC-DISP
25480	T-CHINESE PC-DISP
25502	KOREA DB PC-DISP
25503	T-CHINESE PC-DISP
25504	S-CHINESE PC-DISP
25505	THAILAND PC-DISP
25508	JAPAN PC-DISPLAY
25510	KOREA PC-DISPLAY
25512	S-CHINESE PC-DISP
25514	T-CHINESE PC-DISP
25518	JAPAN PC-DISPLAY
25520	KOREA PC-DISPLAY
25522	S-CHINESE PC-DISP
25524	T-CHINESE PC-DISP
25525	KOREA KS PC-DISP
25527	KOREA KS PC-DISP
25616	KOREA SB PC-DISP
25617	JAPAN PC-DISPLAY
25618	S-CHINESE PC-DISP
25619	T-CHINESE PC-DISP
25664	KOREA KS PC-DISP
28709	T-CHINESE EBCDIC
29109	USA PC-DISPLAY
29172	BRAZIL EBCDIC
29522	LATIN-1 PC-DISP
29523	GREECE PC-DISPLAY
29524	ROECE PC-DISPLAY
29525	TURKEY PC-DISPLAY
29527	CYRILLIC PC-DISP
29528	HEBREW PC-DISPLAY
29529	TURKEY PC-DISPLAY
29532	PORTUGAL PC-DISP
29533	ICELAND PC-DISP
29534	HEBREW PC-DISPLAY
29535	CANADA PC-DISPLAY
29536	ARABIC PC-DISPLAY
29537	DEN/NOR PC-DISP
29540	URDU PC-DISPLAY

CCSID	Description
29541	GREECE PC-DISPLAY
29546	THAILAND PC-DISP
29614	JAPAN PC-DISPLAY
29616	KOREA PC-DISPLAY
29618	S-CHINESE PC-DISP
29620	T-CHINESE PC-DISP
29621	KOREA KS MIX PC
29623	KOREA KS PC-DISP
29712	KOREA PC-DISPLAY
29713	JAPAN PC-DISPLAY
29714	S-CHINESE PC-DISP
29715	T-CHINESE PC-DISP
29760	KOREA KS PC-DISP
32805	JAPAN LATIN EBCDC
33058	JAPAN EBCDIC
33205	SWISS PC-DISPLAY
33268	UK/PORTUGAL EBCDC
33618	LATIN-1 PC-DISP
33619	GREECE PC-DISPLAY
33620	ROECE PC-DISPLAY
33621	TURKEY PC-DISPLAY
33623	CYRILLIC PC-DISP
33624	HEBREW PC-DISPLAY
33632	ARABIC PC-DISPLAY
33636	URDU PC-DISPLAY
33637	GREECE PC-DISPLAY
33665	JAPAN PC-DISPLAY
33698	JAPAN KAT/KAN EBC
33699	JAPAN LAT/KAN EBC
33700	JAPAN PC-DISPLAY
33717	KOREA KS PC-DISP
37301	AUS/GERM PC-DISP
37364	BELGIUM EBCDIC
37719	CYRILLIC PC-DISP
37728	ARABIC PC-DISPLAY
37732	URDU PC-DISPLAY
37761	JAPAN SB PC-DISP
37796	JAPAN PC-DISPLAY
37813	KOREA KS PC-DISP
41397	FRANCE PC-DISPLAY
41460	SWISS EBCDIC
41824	ARABIC PC-DISPLAY
41828	URDU PC-DISPLAY
45493	ITALY PC-DISPLAY
45556	SWISS EBCDIC

CCSID	Description
45920	ARABIC PC-DISPLAY
49589	UK PC-DISPLAY
49652	BELGIUM EBCDIC
53748	INTL EBCDIC
61696	GLOBAL SB EBCDIC
61697	GLOBAL SB PC-DATA
61698	GLOBAL PC-DISPLAY
61699	GLBL ISO-8 ASCII
61700	GLBL ISO-7 ASCII
61710	GLOBAL USE ASCII
61711	GLOBAL USE EBCDIC
61712	GLOBAL USE EBCDIC

Appendix D. Implementation Differences of ADL

This book describes the DD&C implementation of the ADL language, which is part of the Distributed Data Management (DDM) architecture.

There are some differences between the DD&C implementation of the ADL programming language and the formal specification published as the *Distributed Data Management Architecture: Specifications for A Data Language*. This appendix describes those differences for those who are familiar with the formal specification.

The following is true for the implementation of ADL described in this book.

1. The <DEC> floating point numbers are not supported as attributes for FLOAT.
2. The optional <WHEN clause> for data declarations is not supported.
3. Wherever a limit of <max34> is given in the formal specifications, this is replaced by <max31>. For example, the maximum length of a bit string is <max31>.
4. The IN, BETWEEN, and LIKE predicates are not supported.
5. The SUBSTR function is not supported.
6. The ARRAY, BINARY, BITPRE, BOOLEAN, CASE, and SEQUENCE data types are all byte-aligned.
7. The SFXENC attribute is not supported. Instead, the suffix value is obtained from the CCSID type. For further information, see "<CHARSFX>" on page 83.
8. A character literal cannot contain a <newline> token.
9. The maximum length of a bit literal is 32760 bits.
10. The maximum length of character literals and encoded hex literals is 32760 single-byte characters.
11. The maximum length of hexadecimal literals is 32760 nibbles.
12. Subscript lists are not supported. This implies that a reference to a data declaration inside an array declaration can only be made from within this array.
13. No optional identifier is allowed for the INCLUDE statement, which can appear between any two ADL tokens. No comment is allowed in an INCLUDE statement. The include name must be a character literal.
14. The maximum number of nested INCLUDE statements is 32.
15. The meaning of the term *ADL module* has been modified:
    - A *module* is the collection of all ADL declaration and plan spaces provided as inputs to a single call of the DD&C conversion plan builder component. Workspace variables are contained within a module.
    - A *parse unit* is the root of the ADL syntax and contains DECLARE statements and PLAN statements.

In the ADL specifications, this is the definition used for a *module*, except that with DD&C, a parse unit must contain at least one DECLARE statement and one PLAN statement.

One module can therefore consist of several parse units.

16. A <positional identifier> is not assigned to SKIP statements.
17. The parse function of DD&C does not check whether any of the variables or constants used in an ADL plan are defined in a DECLARE statement. This task is performed by the Conversion Plan Builder component.
18. The size and complexity of the ADL source text is restricted by the stack size of the application program calling the Parse function of the ADL declaration translator.
19. The maximum possible record size is <max31> bits.
20. The maximum length of a BINARY data type is 32 bits.
21. The maximum number of plan parameters allowed for each plan is <max8> input parameters and <max8> output parameters.

Appendix E. Bachus Naur Form Summary

```
<argument> ::= <value expression>

<argument list> ::=
  (<argument>{[,<argument>]...})

<ARRAY> ::=
  ARRAY
  <ARRAY attributes list>
  OF
    {
      {<field><terminator>} |
      <CASE> |
      <SEQUENCE> |
      {<subtype instance><terminator>}
    }

<ARRAY attributes list> ::=
  {
    <DMNLST attribute>
    <ARRAY defaulted attributes list>
    <ARRAY optional attributes list>
  }!

<ARRAY defaulted attributes list> ::=
  {
    [<DMNLOW attribute>]
    [<MAXALC attribute>]
    [<SKIP attribute>]
  }!

<ARRAY optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<arrow> ::=
  <-

<ASIS> ::=
  ASIS
  <ASIS attributes list>

<ASIS attributes list> ::=
  {
    <ASIS defaulted attributes list>
    <ASIS optional attributes list>
  }!
```

```
<ASIS defaulted attributes list> ::=
  [
    LENGTH({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})
  ]! |
  [
    LENGTH(*)
    MAXLEN({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})]
  ]! |
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max ASIS> | <constant identifier>})
    UNITLEN({1 | 8 | 16 | <constant identifier>})]
  ]!

<ASIS optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<assignment statement> ::=
  [<identifier>:]
  <qualified identifier>
  <arrow>
  <value expression>
  <terminator>

<BEGIN statement> ::=
  [<identifier>:]
  BEGIN
  <terminator>

<BINARY> ::=
  BINARY
  <BINARY attributes list>

<BINARY attributes list> ::=
  {
    <BINARY defaulted attributes list>
    <BINARY optional attributes list>
  }!
```

```
<BINARY defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [<PRECISION attribute>]
    [<RADIX attribute>]
    [<SCALE attribute>]
    [<SGNCNV attribute>]
    [<SIGNED attribute>]
  }!

<BINARY optional attributes list> ::=
  {
    [<HELP attribute>]
    [LENGTH({1..32 | <constant identifier>})]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<BIT> ::=
  BIT
  <BIT attributes list>

<BIT attributes list> ::=
  {
    <BIT defaulted attributes list>
    <BIT optional attributes list>
  }!

<BIT defaulted attributes list> ::=
  [LENGTH({1..<max31> | <constant identifier>})] |
  [
    LENGTH(*)
    MAXLEN({1..<max31> | <constant identifier>})
  ]! |
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max31> | <constant identifier>})
  ]!

<bit literal> ::=
  {b | B}<bit string> [<separator><bit string>]...

<BIT optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

```
<bit string> ::= '[0 | 1]...'

<BITPRE> ::=
  BITPRE
  <BITPRE attributes list>

<BITPRE attributes list> ::=
  {
    <BITPRE defaulted attributes list>
    <BITPRE optional attributes list>
  }!

<BITPRE defaulted attributes list> ::=
  {
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max BITPRE> |
          <constant identifier>
        })
    ]
    [<PREBYTRVS attribute>]
    [<PRELEN attribute>]
    [<PRESIGNED attribute>]
  }!

<BITPRE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<BLNENC attribute> ::=
 BLNENC ({<constant identifier> | <positive integer> })

<BOOLEAN> ::=
  BOOLEAN
  <BOOLEAN attributes list>

<BOOLEAN attributes list> ::=
  {
    <BOOLEAN defaulted attributes list>
    <BOOLEAN optional attributes list>
  }!
```

```
<BOOLEAN defaulted attributes list> ::=
  {
    [<BLNENC attribute>]
    [<BYTRVS attribute>]
    [LENGTH({1..64 | <constant identifier>})]
  }!

<boolean factor> ::=
  [NOT]<boolean primary>

<boolean literal> ::= FALSE | TRUE

<BOOLEAN optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<boolean primary> ::=
  <boolean literal> |
  <qualified identifier> |
  <predicate> |
  (<condition>)

<boolean term> ::=
  <boolean factor> |
  <boolean term> AND <boolean factor>

<BYTRVS attribute> ::=
  BYTRVS ({<boolean literal> | <constant identifier>})

<CALL statement> ::=
  [<identifier>:]
  CALL
  <program name>
  <argument list>
  <terminator>

<CASE> ::=
  CASE
  <CASE attributes list>
  <BEGIN statement>
    <WHEN statement>...
    [<OTHERWISE statement>]
  <END statement>

<CASE attributes list> ::=
  {
    <CASE defaulted attributes list>
    <CASE optional attributes list>
  }!
```

```
<CASE defaulted attributes list> ::=
  [<MAXALC attribute>]

<CASE optional attributes list> ::=
  {
    [<LENGTH attribute>]
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<CCSID attribute> ::=
  CCSID
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })

<CHAR> ::=
  CHAR
  <CHAR attributes list>

<CHAR attributes list> ::=
  {
    <CHAR defaulted attributes list>
    <CHAR optional attributes list>
  }!

<CHAR defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<JUSTIFY attribute>]
    <CHAR length attributes list>
  }!

<CHAR HIGHLOW1 attributes list> ::=
  [
    HIGH({1..<max CHAR> | <constant identifier>})
    LOW({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR HIGHLOW2 attributes list> ::=
  [
    HIGH({1..<max CHAR> | <constant identifier>})
    LOW(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!
```

```
<CHAR HIGHLOW3 attributes list> ::=
  [
    HIGH(<qualified identifier>)
    LOW({1..<max CHAR> | <constant identifier>})
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR HIGHLOW4 attributes list> ::=
  [
    HIGH(<qualified identifier>)
    LOW(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR LENGTH asterisk attributes list> ::=
  [
    LENGTH(*)
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR length attributes list> ::=
  [
    <CHAR LENGTH asterisk attributes list> |
    <CHAR LENGTH fixed attributes list> |
    <CHAR LENGTH identifier attributes list> |
    <CHAR HIGHLOW1 attributes list> |
    <CHAR HIGHLOW2 attributes list> |
    <CHAR HIGHLOW3 attributes list> |
    <CHAR HIGHLOW4 attributes list>
  ]

<CHAR length fixed attributes list> ::=
  [
    LENGTH({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!

<CHAR LENGTH identifier attributes list> ::=
  [
    LENGTH(<qualified identifier>)
    <MAXALC attribute>
    MAXLEN({1..<max CHAR> | <constant identifier>})
    UNITLEN({8 | 16 | <constant identifier>})
  ]!
```

```
<CHAR optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<character> ::=
  <digit> |
  <letter> |
  <special character> |
  <underscore> |
  <space>

<character literal> ::=
  <character string>[<separator><character string>]...
  [<CCSID attribute>]

<character string> ::=
  '[<source character> | <quote representation>]...'

<CHARPRE> ::=
  CHARPRE
  <CHARPRE attributes list>

<CHARPRE attributes list> ::=
  {
    <CHARPRE defaulted attributes list>
    <CHARPRE optional attributes list>
  }!

<CHARPRE defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max CHARPRE> |
          <constant identifier>
        })
    ]
    [<PREBYTRVS attribute>]
    [<PRELEN attribute>]
    [<PRESIGNED attribute>]
    [UNITLEN({8 | 16 | <constant identifier>})]
  }!
```

```
<CHARPRE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<CHARSFX> ::=
  CHARSFX
  <CHARSFX attributes list>

<CHARSFX attributes list> ::=
  {
    <CHARSFX defaulted attributes list>
    <CHARSFX optional attributes list>
  }!

<CHARSFX defaulted attributes list> ::=
  {
    [<CCSID attribute>]
    [<MAXALC attribute>]
    [
      MAXLEN
        ({
          1..<max CHARSFX> |
          <constant identifier>
        })
    ]
    [UNITLEN({8 | 16 | <constant identifier>})]
  }!

<CHARSFX optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<comment> ::=
  /* <source character>... */

<comparison operator> ::=
  = | < | > | <> | <= | >=

<comparison predicate> ::=
  <value expression>
  <comparison operator>
  <value expression>

<parse unit> ::= {<DECLARE statement> | <PLAN statement>}...
```

```
<COMPLEX attribute> ::=
  COMPLEX
  ({<boolean literal> | <constant identifier>})

<condition> ::=
  <boolean term> |
  <condition> OR <boolean term>

<constant identifier> ::=
 <identifier> |
 ALGEBRAIC |  DGTLSTBYT | EXACT | FB32 |
 FB64 | FB80 | FH32 | FH64 | FH128 | FI128 |
 FRSBYT | LEFT | LOGICAL | LSTBIT | LSTBYT |
 RIGHT | ROUND | TRUNCATE | ZONFRSBYT | ZONLSTBYT

<CONSTANT statement> ::=
  <constant identifier> :
  CONSTANT
  {<literal> | <constant identifier>}
  <terminator>

<CONSTRAINED attribute> ::=
  CONSTRAINED
  ({<boolean literal> | <constant identifier>})

<constructor> ::=
  <ARRAY> |
  <CASE> |
  <SEQUENCE>

<data> ::=
  <constructor> |
  {<subtype instance> <terminator>} |
  {<field> <terminator>}

<data declaration statement> ::=
  [<identifier>:]... <data>

<DECLARE attributes list> ::=
  {
    [<NOTE attribute>]
    [<HELP attribute>]
    [<TITLE attribute>]
  }!
```

```
<DECLARE statement> ::=
  [<identifier>:]
  DECLARE
  <DECLARE attributes list>
  <BEGIN statement>
    {
      [<DEFAULT statement>] |
      [<CONSTANT statement>] |
      [<SUBTYPE statement>] |
      <data declaration statement>
    }...
  <END statement>

<DEFAULT statement> ::=
  [<identifier>:]
  DEFAULT
    {
      {ARRAY <ARRAY defaulted attributes list>} |
      {ASIS <ASIS defaulted attributes list>} |
      {BINARY <BINARY defaulted attributes list>} |
      {BIT <BIT defaulted attributes list>} |
      {BITPRE <BITPRE defaulted attributes list>} |
      {BOOLEAN <BOOLEAN defaulted attributes list>} |
      {CASE <CASE defaulted attributes list>} |
      {CHAR <CHAR defaulted attributes list>} |
      {CHARPRE <CHARPRE defaulted attributes list>} |
      {CHARSFX <CHARSFX defaulted attributes list>} |
      {ENUMERATION <ENUMERATION defaulted attributes list>} | .
      {FLOAT <FLOAT defaulted attributes list>} |
      {PACKED <PACKED defaulted attributes list>} |
      {ZONED <ZONED defaulted attributes list>}
    }
  <terminator>

<delimiter token> ::=
  <character literal> |
  <special symbol>

<digit> ::=
  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<dimension> ::=
  {
    [<DMNLOW attribute>]
    {
      <DMNHIGH attribute> |
      <DMNSIZE attribute>
    }
    [<DMNMAX attribute>]
  }!
```

```
<DMNHIGH attribute> ::=
  DMNHIGH
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })

<DMNLOW attribute> ::=
  DMNLOW
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })

<DMNLST attribute> ::=
  DMNLST (<dimension> [{, <dimension>}...])

<DMNMAX attribute> ::=
  DMNMAX({<positive integer> | <constant identifier>})

<DMNSIZE attribute> ::=
  DMNSIZE
    ({
      <signed integer> |
      <constant identifier> |
      <qualified identifier>
    })

<encoded hex literal> ::= <hex literal> <CCSID attribute>

<END statement> ::=
  [<identifier>: ]
  END
  <terminator>

<ENUMERATION> ::=
  ENUMERATION
  <ENUMERATION attributes list>
  <enumeration list>

<ENUMERATION attributes list> ::=
  {
    <ENUMERATION defaulted attributes list>
    <ENUMERATION optional attributes list>
  }!
```

```
<ENUMERATION defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [LENGTH({8 | 16 | 32 | <constant identifier>})]
    [<SGNCNV attribute>]
    [<SIGNED attribute>]
  }!

<enumeration identifier> ::=
  <identifier>

<enumeration list> ::=
  (<enumeration value>[, <enumeration value>]...)

<ENUMERATION optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<enumeration value> ::=
  <enumeration identifier>[:<signed integer>]

<escape character> ::= <value specification>

<field> ::=
  <ASIS> |
  <BINARY> |
  <BIT> |
  <BITPRE> |
  <BOOLEAN> |
  <CHAR> |
  <CHARPRE> |
  <CHARSFX> |
  <ENUMERATION> |
  <FLOAT> |
  <PACKED> |
  <ZONED>

<file name> ::=
  <character literal>

<FIT attribute> ::=
 FIT ({<constant identifier> | <positive integer> })

<FLOAT> ::=
  FLOAT
  <FLOAT attributes list>
```

```
<FLOAT attributes list> ::=
  {
    <FLOAT defaulted attributes list>
    <FLOAT optional attributes list>
  }!

<FLOAT defaulted attributes list> ::=
  {
    [<BYTRVS attribute>]
    [<COMPLEX attribute>]
    [<FIT attribute>]
    [<FORM attribute>]
    [<RADIX attribute>]
  }!

<FLOAT optional attributes list> ::=
  {
    [<HELP attribute>]
    [<PRECISION attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<FORM attribute> ::=
 FORM ({<constant identifier> | <positive integer> })

<HELP attribute> ::=
  HELP
    ({
      <character literal> |
      <encoded hex literal> |
      <constant identifier>
    })

<hex digit> ::= <digit> | A | B | C | D | E | F

<hex literal> ::=
  {x | X}<hex string> [<separator><hex string>]...

<hex string> ::= '[<hex digit>]...'

<HIGH attribute> ::=
  HIGH
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })
 <identifier> ::=
  <identifier string>
  | " <identifier string> "
<identifier string> ::=
  {<letter> | <digit> | <identifier character>}...

 <identifier character> ::=
   ? | % | & | <underscore>

 <INCLUDE statement> ::=
   INCLUDE
   <file name>
   <terminator>

 <INPUT parameter> ::=
   <qualified identifier>
     [: INPUT
       {
         [<LENGTH attribute>]
         [<CCSID attribute>]
       }!
     ]

 <JUSTIFY attribute> ::=
  JUSTIFY ({<constant identifier> | <positive integer> })

 <key word> ::=
   ALGEBRAIC | AND | ARRAY | ASIS | BEGIN
    | BINARY | BIT | BITPRE | BLNENC
    | BOOLEAN | BYTRVS | CALL | CASE | CCSID
    | CHAR | CHARPRE | CHARSFX | COMPLEX | CONSTANT
    | CONSTRAINED | DECLARE | DEFAULT | DGTLSTBYT | DMNHIGH
    | DMNLOW | DMNLST | DMNMAX | DMNSIZE | END
    | ENUMERATION | ESCAPE | EXACT | FALSE | FB32
    | FB64 | FB80 | FH32 | FH64 | FH128
    | FI128 | FIT | FLOAT | FORM | FRSBYT
    | HELP | HIGH | INCLUDE | INPUT
    | JUSTIFY | LEFT | LENGTH | LOGICAL
    | LOW | LSTBIT | LSTBYT | MAXALC | MAXLEN
    | NOT | NOTE | OF | OR | OTHERWISE
    | OUTPUT | PACKED | PLAN | PREBYTRVS | PRECISION
    | PRELEN | RADIX | REJECT | RIGHT
    | ROUND| SCALE | SEQUENCE | SGNLOC | SGNMNS
    | SGNPLS | SGNUNS | SIGNED | SKIP | SUBSTR
    | SUBTYPE | THEN | TITLE | TRUE | TRUNCATE
    | UNITLEN | WHEN | ZONED | ZONENC | ZONFRSBYT
    | ZONLSTBYT
 <LENGTH attribute> ::=
   LENGTH
     ({
       <signed integer> |
       <constant identifier> |
       <qualified identifier> |
        *
     })

<LENGTH function> ::=
  LENGTH ({<literal> | <constant identifier> | <qualified identifier> })

 <letter> ::=
   <upper case letter> | <lower case letter>

 <literal> ::=
   <character literal> |
   <noncharacter literal>

 <LOW attribute> ::=
   LOW
     ({
       <positive integer> |
       <constant identifier> |
       <qualified identifier>
     })

 <lower case letter> ::=
   a | b | c | d | e | f | g | h | i |
   j | k | l | m | n | o | p | q | r |
   s | t | u | v | w | x | y | z

 <max ASIS> ::=
   See Syntax Rule 3 on page 61.

 <max CHAR> ::=
   See Syntax Rule 2 on page 80.

 <max CHARPRE> ::=
   See Syntax Rule 1 on page 83.

 <max CHARSFX> ::=
   See Syntax Rule 2 on page 85.

 <MAXALC attribute> ::=
   MAXALC ({<boolean literal> | <constant identifier>})
```

```
<MAXLEN attribute> ::=
  MAXLEN
    ({
      <positive integer> |
      <constant identifier> |
      <qualified identifier>
    })

<max15> ::= 32,767

<max28> ::= 268,435,455

<max31> ::= 2,147,483,647

<max7>  ::= 127

<max8>  ::= 255

<min31> ::= -2,147,483,648

<min7>  ::= -128

<new line> ::=
  See “Syntax rules” on page 33.

<noncharacter literal> ::=
  <bit literal> |
  <signed integer> |
  <positive integer> |
  <hex literal> |
  <boolean literal> |
  <encoded hex literal>

<nondelimiter token> ::=
  <qualified identifier> |
  <key word> |
  <noncharacter literal>

<NOTE attribute> ::=
  NOTE
    ({
      <character literal> |
      <encoded hex literal> |
      <constant identifier>
    })
```

```
<OTHERWISE clause> ::=
  OTHERWISE
    {
      <data declaration statement> |
      <REJECT statement> |
      <SKIP statement> |
      <terminator>
    }

<OTHERWISE statement> ::=
  [<identifier>: ]
  <OTHERWISE clause>

<OUTPUT parameter> ::=
  <qualified identifier>
  : OUTPUT
    {
      [<MAXLEN attribute>]
      [<CCSID attribute>]
    }!

<PACKED> ::=
  PACKED
  <PACKED attributes list>

<PACKED attributes list> ::=
  {
    <PACKED defaulted attributes list>
    <PACKED optional attributes list>
  }!

<PACKED defaulted attributes list> ::=
  {
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [PRECISION({1..31 | <constant identifier>})]
    [SCALE({-128..127 | <constant identifier>})]
    <PACKED SIGNED attributes list>
  }!

<PACKED optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!
```

```
<PACKED SGNLOC attributes list> ::=
  SGNLOC(DGTLSTBYT)
  {
    [
      <SGNMNS attribute>
      <SGNPLS attribute>
    ]! |
    [<SGNUNS attribute>]
  }

<PACKED SIGNED attributes list> ::=
  [
    SIGNED(TRUE)
    <PACKED SGNLOC attributes list>
  ]! |
  [SIGNED(FALSE)]

<parameter> ::=
  {<INPUT parameter> | <OUTPUT parameter>}

<parameter list> ::=
  (<parameter>[,<parameter>]...)

<pattern> ::= <value specification>

<PLAN statement> ::=
  <identifier>:
  PLAN
  <parameter list>
  <BEGIN statement>
    {<assignment statement> | <CALL statement>}...
  <END statement>

<positional identifier> ::= " <digit>... "

<positive integer> ::= [+] <digit>...

<PREBYTRVS attribute> ::=
  PREBYTRVS ({<boolean literal> | <constant identifier>})

<PRECISION attribute> ::=
  PRECISION
  ({<positive integer> | <constant identifier>})

<predicate> ::=
  <comparison predicate>

<PRELEN attribute> ::=
  PRELEN ({8 | 16 | 32 | <constant identifier>})

<PRESIGNED attribute> ::=
  PRESIGNED ({<boolean literal> | <constant identifier>})
```

```
<program name> ::= <character literal>

<qualified identifier> ::=
  <qualifier list>

<qualifier> ::=
  <identifier> | <positional identifier>

<qualifier list> ::=
  [<qualifier>.]...<qualifier>

<quote representation> ::= ''

<RADIX attribute> ::=
  RADIX ({2 | 10 | <constant identifier>})

<REJECT statement> ::=
  [<identifier>:]
  REJECT <terminator>

<SCALE attribute> ::=
  SCALE({<signed integer> | <constant identifier>})

<separator> ::=
  {<comment> | <space> | <new line> | <tab>}...

<SEQUENCE> ::=
  SEQUENCE
  <SEQUENCE attributes list>
  <BEGIN statement>
    {<data declaration statement> | <SKIP statement>}...
  <END statement>

<SEQUENCE attributes list> ::=
  <SEQUENCE optional attributes list>

<SEQUENCE optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<SGNCNV attribute> ::=
 SGNCNV ({<constant identifier> | <positive integer> })

<SGNLOC attribute> ::=
 SGNLOC ({<constant identifier> | <positive integer> })

<SGNMNS attribute> ::=
  SGNMNS ({<hex literal> | <constant identifier>})
```

```
<SGNPLS attribute> ::=
  SGNPLS ({<hex literal> | <constant identifier>})

<SGNUNS attribute> ::=
  SGNUNS ({<hex literal> | <constant identifier>})

<SIGNED attribute> ::=
  SIGNED ({<boolean literal> | <constant identifier>})

<signed integer> ::= [+ | -] <digit>...

<SKIP attribute> ::=
  SKIP ({<positive integer> | <constant identifier>})

<SKIP statement> ::=
  [<identifier>:]
  SKIP({<positive integer> | <constant identifier>})
  <terminator>

<source character> ::=
  See “Syntax rule” on page 20.

<space> ::=
  See “Syntax rules” on page 33.

<special character> ::=
  . | , | : | ; | ? | ( | ) | ' | " |
  / | - | & | + | % | = | * | > | <

<special symbol> ::=
  ' | * | . | , | : | ; | = | + | - | ( | ) |
  > | < | <> | >= | <= | /* | */ | <arrow>

<subtype identifier> ::= <identifier>

<subtype instance> ::=
  <subtype identifier>
  <subtype instance attributes list>
```

```
<subtype instance attributes list> ::=
  {
    <ARRAY attributes list> |
    <ASIS attributes list> |
    <BIT attributes list> |
    <BITPRE attributes list> |
    <BOOLEAN attributes list> |
    <CASE attributes list> |
    <CHAR attributes list> |
    <CHARPRE attributes list> |
    <CHARSFX attributes list> |
    <BINARY attributes list> |
    <ENUMERATION attributes list> |
    <FLOAT attributes list> |
    <PACKED attributes list> |
    <SEQUENCE attributes list> |
    <ZONED attributes list>
  }

<SUBTYPE statement> ::=
  <subtype identifier>:
  SUBTYPE
  OF
    {
      <constructor> |
      {<field> <terminator>} |
      {<subtype instance> <terminator>}
    }

<tab> ::=
  See “Syntax rules” on page 33.

<terminator> ::= ;

<THEN clause> ::=
  THEN
    {
      <data declaration statement> |
      <REJECT statement> |
      <SKIP statement> |
      <terminator>
    }

<TITLE attribute> ::=
  TITLE
    ({
      <character literal> |
      <encoded hex literal> |
      <constant identifier>
    })
```

```
<token> ::=
  <nondelimiter token> | <delimiter token>

<underscore> ::= _

<UNITLEN attribute> ::=
  UNITLEN({ 1 | 8 | 16 | <constant identifier>})

<upper case letter> ::=
  A | B | C | D | E | F | G | H | I |
  J | K | L | M | N | O | P | Q | R |
  S | T | U | V | W | X | Y | Z

<value expression> ::=
  <value specification> |
  (<value expression>)

<value specification> ::=
  <literal> |
  <constant identifier> |
  <qualified identifier> |
  <LENGTH function>

<WHEN clause> ::=
  WHEN <condition>

<WHEN statement> ::=
  [<identifier>:]
  <WHEN clause>
  <THEN clause>

<ZONED> ::=
  ZONED
  <ZONED attributes list>

<ZONED attributes list> ::=
  {
    <ZONED defaulted attributes list>
    <ZONED optional attributes list>
  }!

<ZONED defaulted attributes list> ::=
  {
    [<COMPLEX attribute>]
    [<CONSTRAINED attribute>]
    [<FIT attribute>]
    [PRECISION({1..31 | <constant identifier>})]
    [SCALE({-128..127 | <constant identifier>})]
    <ZONED SIGNED attributes list>
    [<ZONENC attribute>]
  }!
```

```
<ZONED optional attributes list> ::=
  {
    [<HELP attribute>]
    [<NOTE attribute>]
    [<TITLE attribute>]
  }!

<ZONED SGNLOC attributes list> ::=
  [
    SGNLOC({ZONLSTBYT | ZONFRSBYT})
    <SGNMNS attribute>
    <SGNPLS attribute>
  ]! |
  [
    SGNLOC({LSTBYT | FRSBYT})
    <CCSID attribute>
  ]!

<ZONED SIGNED attributes list> ::=
  [
    SIGNED(TRUE)
    <ZONED SGNLOC attributes list>
  ]! |
  [SIGNED(FALSE)]

<ZONENC attribute> ::=
  ZONENC ({<hex literal> | <constant identifier>})
```

Glossary

This glossary defines many of the terms and abbreviations used in this manual. If you do not find the term you are looking for, please refer to the index or to the *Dictionary of Computing*, SC20-1699.

A Data Language. A language for describing the fields, arrays, etc. of data records in a programming environment so the records can be transparently accessed by other programming environments.

abend. Abnormal end of task.

access method. The part of the DDM architecture which accepts commands to access and process the records of a file.

ADL. A Data Language

ADLCA. ADL Communications Area, which contains control information for exception handling.

alternate index file. A file that has a different key path over a base file. The base file can be a keyed, direct, or sequential file.

API. Application Programming Interface

array. An object consisting of an ordered collection of homogeneous objects mapped onto N dimensions.

attribute. An object that specifies information about another object, such as the length of a character string, field or the date at which a record was last accessed.

Backus Naur Form. BNF

BNF. The metalanguage, Backus Naur Form.

case. An ordered collection of selections for the declaration of a field.

CCS. Common Communication Support.

CCSID. Coded Character Set Identifier.

CDRA. Character Data Representation Architecture.

character string. A string of bytes containing characters encoded as specified by its CCSID attribute.

CM. Communications Manager

complete path name. The specifications for a file which includes the drive (if OS/2), directory, filename and file extension.

constructor. A data type that consists of zero or more instances of other data types. ADL examples of constructors are ARRAYs, CASEs, AND SEQUENCESs.

CPGID. Code Page Global Identifier.

CTOK. Condition Token. A 12-byte area in which information about the execution of a called program is returned by that program.

CUA. Common User Access.

data conversion. A set of programs that convert data according to defined data descriptions. For example, characters can be converted from EBCDIC to ASCII, and numeric data can be converted from System /370 packed decimal to IEEE floating point or ASCII character (or vice versa).

data description. Specification of the layout of data. The data description of data stored in a file can be viewed as a file attribute.

data security. The protection of data against unauthorized disclosure, transfer, modifications or destruction, whether accidental or intentional.

data set. The major unit of data storage and retrieval. It consists of a collection of data in one of several prescribed arrangements which is described by control information that the system has access to.

data stream. All data transmitted through a data channel in a single read or write operation.

DBCS. Double Byte Character Set - characters that are encoded in two bytes.

DD&C. Data Description and Conversion. An architecture extension to DDM.

DDM. A set of interfaces that gives users access to data files that reside on remote systems connected by a communication network. The DDM interfaces enable an application program to retrieve, add, update and delete data records in a file existing on a remote system. The DDM interfaces can be used to communicate between systems that have different architectures.

deadlock. Unresolved contention for the use of a resource. Each element in a process is waiting for an action by, or a response from, the other.

declaration. An ADL statement specifying the type, attributes, and entities of a record or an object.

DFM client. Translates requests from the source system for access to file data on a remote system into a standard architected DDM request.

DFM server. A DFM component that accepts remote requests to access data and translates the requests into data management requests on the target system.

direct file. A file that is organized so that there is a relationship between the contents of the records and their positions.

discriminant. A field that can be tested by a WHEN statement of a CASE to determine if the data declaration clause of the WHEN statement is to be selected.

Distributed Data Management (DDM). Architecture for accessing distributed data located in files and distributed relational databases.

Distributed File Management (DFM). Strategy for a set of programming facilities that implement the file aspects of the DDM architecture on those systems which represent distributed environments.

intersystem communication. Communication between different systems by means of SNA facilities.

DRBA. Distributed relational data base access.

element. An instance of a data type that is a component of a constructor data type.

entity. A record or an object.

fixed-point number. An object representing a number whose precision and scale are fixed.

floating-point number. An object representing a number with fixed precision and floating scale.

FSD. File System Driver.

HLL. High Level Language

HPFS. High Performance File System.

IFS. Installable File System.

keyed file. A file organization that supports keyed access to the records of the file.

LAN. Local Area Network.

Local Area Network. LAN

LDM. Local Data Management.

LDMI. Local Data Management Interface.

local file. A file that resides on the same system as the application program that is accessing it.

LU. Logical unit.

mixed character string. A character string consisting of both SBCS and DBCS characters.

module. A set of data declarations and plans used to convert data.

object. An instance of a type, such as a field of a record or an attribute of a field.

parse unit. The amount of ADL text that is parsed with one call of the parse function of DD&C. It consists of declarations, plans or both.

PL. Programming Language

plan. A program for converting data from one representation to another.

protocol. A set of rules to be followed by communication systems.

RACF. Resource Access Control Facility. An external security management facility.

record. The basic unit of data stored in a file and transferred between DDM source and target servers. An instance of a field or constructor type.

record file. Record files consist of data fields organized into records that can be accessed as a set of bytes.

remote file. A file that resides on a system other than the system where the application program requesting access to the file resides.

Remote Record Access Support. The DFM function that allows VSAM applications to access remote file data.

SBCS. Single Byte Character Set - characters that are encoded in one byte.

SCM. Source Communications Manager. The DDM layer responsible for interfacing with the local communications facilities. It coordinates the sending and receiving of data on the source system.

sequence. An object consisting of an ordered collection of heterogeneous objects.

sequential file. A file in which records are arranged in exactly the same sequence as they were stored into the file.

SNA. Systems Network Architecture.

source system. A system that requests access to data on another system. In a client/server relationship, it is the client system.

Stream Agent. The DDM program responsible for transformation of data between the stream oriented API requests and the DDM byte requests.

subtype. In the type hierarchy, a lower level type which inherits characteristics and attributes from a higher level type.

supertype. In the type hierarchy, a higher level type from which a subtype inherits its characteristics and attributes.

Systems Network Architecture (SNA). The description of the logical structure, formats, protocols, and operational sequences for transmitting information units through and controlling the configuration and operation of networks.

target system. The system that contains data that is being accessed by another system. In a client/server relationship, it is the server system.

target system data. Data considered to be owned and maintained according to the rules and functions prescribed by the data manager on the target system.

TP. Transaction Program

user exit. A point in an IBM-supplied program at which a user-exit routine may be given control.

type. A set of representable values encapsulated by a set of operations on those values. Each programming language defines its own set of data types, such as the *PIC* data types of COBOL or the *integer* data type of C.

type domain. The set of programs and objects that process or store data of the same set of programming language data types and representations. Examples of type domains are MVS COBOL and OS/2 C.

type manager. A facility for a single type domain capable of:

- Mapping programming language data descriptions into ADL.
- Mapping the ADL of another type domain into ADL of its own type domain.
- Mapping ADL into programming language data descriptions.

VTAM. Virtual Telecommunications Access Method.

Index

IBM

Program Number: 5765-548
5765-549
5622-793
5622-794

Printed in U.S.A.

SC26-7092-00