The Data Access Protocol: DAP Version 4.0

Volume 1: Data Model and Serialized Representation

|  |  |
| --- | --- |
| Date: | May 31, 2012 |
| Revised: | June 26, 2012 |
| Status: | Draft |
|  |  |
| Authors: | John Caron (Unidata) |
|  | Ethan Davis (Unidata) |
|  | David Fulker (OPeNDAP) |
|  | James Gallagher (OPeNDAP) |
|  | Dennis Heimbigner (Unidata) |
|  | Nathan Potter (OPeNDAP) |
|  |  |
| Copyright: | 2012 University Corporation for Atmospheric Research and Opendap.org |
|  |  |
|  |  |

Abstract

This document defines the Data Access Protocol (DAP) version 4.0 (referred to also as DAP4). This data transmission protocol is intended to supersede all previous versions of the DAP protocol. DAP4 is designed specifically for science data. The protocol relies on the widely used and stable standards, and is capable of representing a wide variety of scientific data types.

Distribution of this document is unlimited.

This document takes material from the DAP2 specification and the OPULS Wiki page.

|  |  |
| --- | --- |
| Changes: |  |
| 2012.05.24: | Initial Draft |
| 2012.05.27 | Added specification of chunk order |
| 2012.05.28 | Added specification and interpretation of simple queries |
| 2012.05.28 | Added discussion about nested sequences. |
| 2012.05.29 | Formatting changes |
| 2012.6.05 | Removed serialized representation sections and constraint sections until James provides direction. |
| 2012.6.24 | Merge all changes from Gallagher, Potter, and Caron, except as noted. |
| 2012.6.24 | Removed all references to Sequences. |
| 2012.6.24 | Inserted James’ version of serialized representation. |
| 2012.6.25 | Added DMR RELAX-NG Grammar. |
| 2012.6.24 | Added (semi-)formal description of the DAP4 serialization scheme. |
| 2012.6.26 | Added: (1) Revised Char type (2) Revised unlimited dimension rules (3) revised MAP rules. (4) Removed HTTP references |
| 2012.7.09 | Added discussion of identifier |
| 2012.7.10 | Added discussion of XML escaping |
| 2012.7.10 | Fix discrepancies between the formal definition of the on-th-wire format and the examples. |
| 2012.7.12 | Removed UByte and made Byte == UInt8 |
| 2012.8.21 | Added draft constraints section |
| 2012.8.25 | Improved the discussion of named slices in constraints. |
| 2012.9.4 | Minor change to the grammar for simple constraints. |
| 2012.9.6 | Updated the Data Response section so that it no longer mentions Multipart MIME; edited the sections on FQNs and Attributes. I’ve added ‘nested attributes’ back into the text. I also added ‘Sequence’ in several places where we will need it once we’ve worked out how those are to be handled. |
| 2012.11.1 | Integrate Jame’s changes with recent changes |

Contents

[1. Introduction 5](#_Toc339628562)

[2. Requirements 5](#_Toc339628563)

[3. Overall Operation 5](#_Toc339628564)

[4. Characterization of a Data Source 5](#_Toc339628565)

[5. DMR Declarations 6](#_Toc339628566)

[5.1 XML Escaping Within the DMR 6](#_Toc339628567)

[5.2 Names 6](#_Toc339628568)

[5.3 Fully Qualified Names 6](#_Toc339628569)

[5.4 Non-Data Bearing Declarations versus Data Bearing Declarations 7](#_Toc339628570)

[5.5 Groups 7](#_Toc339628571)

[5.6 Dimensions 8](#_Toc339628572)

[5.7 Enumeration Types 8](#_Toc339628573)

[5.8 Atomic Types 8](#_Toc339628574)

[5.9 Container Types 10](#_Toc339628575)

[5.10 Variables 10](#_Toc339628576)

[5.11 Attributes and Arbitrary XML 12](#_Toc339628577)

[5.12 Namespaces 14](#_Toc339628578)

[6. Data Representation 14](#_Toc339628579)

[6.1 Response Structure 14](#_Toc339628580)

[6.2 The DAP4 Serialized Representation (DSR) 15](#_Toc339628603)

[6.3 Example responses 17](#_Toc339628604)

[7. DAP4 Chunked Data Representation 20](#_Toc339628605)

[7.1 Chunked Format Grammar 21](#_Toc339628606)

[7.2 Lexical Structure 21](#_Toc339628607)

[7.3 Error Chunk Schema 21](#_Toc339628608)

[8. Constraints 22](#_Toc339628609)

[8.1 Syntax 22](#_Toc339628610)

[8.2 Interpretation 23](#_Toc339628611)

[Appendix A. FQN Syntax 25](#_Toc339628612)

[Appendix B. DAP4 Lexical Elements 25](#_Toc339628613)

[B.1 DMR Lexical Element Syntax 25](#_Toc339628614)

[Appendix C. DAP4 DMR Syntax as a RELAX NG Schema 28](#_Toc339628615)

# Introduction

This specification defines the protocol referred to as the Data Access Protocol, version 4.0 (“DAP4”). In this document ‘DAP’ refers to DAP4 unless otherwise noted.

DAP is intended to be the successor to all previous versions of the DAP (specifically DAP version 2.0). The goal is to provide a very general data model capable of representing a wide variety of existing data sets. The DAP builds upon a number of existing data representation schemes. Specifically, it is influenced by DAP version 2.0[], netCDF-4[], HDF5[], and CDM[].

The DAP is a protocol for access to data organized as variables. It is particularly suited to accesses by a client computer to data stored on remote (server) computers that are networked to the client computer. DAP was designed to hide the implementation of different collections of data. The assumption is that a wide variety of data sets using a wide variety of data schemas can be translated into the DAP protocol for transmission from the server holding that dataset to a client computer for processing.

It is important to stress the discipline neutrality of the DAP and the relationship between this and adoption of the DAP in disciplines other than the Earth sciences. Because the DAP is agnostic as relates to discipline, it can be used across the very broad range of data types encountered in oceanography - biological, chemical, physical and geological. There is nothing that constrains the use of the DAP to the Earth sciences.

# Requirements

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “MAY” and “OPTIONAL” in this document are to be interpreted as described in RFC 2119 [].

# Overall Operation

The DAP is a stateless protocol that governs clients making requests from servers, and servers issuing responses to those requests. This section provides an overview of the requests and responses (i.e. the messages) that DAP-compliant software MUST support. These messages are used to request information about a server and data made accessible by that server, as well as requesting data values themselves.

The DAP uses two responses to represent a data source. One response, the DMR returns metadata information describing the structure of a request for data. That is, it characterizes the variables, their datatypes, names and attributes. The second response, the Data Response, returns both the metadata about the request, but also the data that was requested. The DMR and the metadata part of the Data Response are represented using a specific XML[] representation. The syntax of that representation is defined in Section ?.

The DAP returns error information using an Error response. If a request for any of the three basic responses cannot be completed then an Error response is returned in its place.

The two responses (DMR and Data Response) are complete in and of themselves so that, for example, a client can use the data response without ever requesting either of the two other responses. In many cases, client programs will request the DMR response first before requesting the Data Response but there is no requirement they do so and no server SHALL require that behavior on the part of clients.

Operationally, communication between a DAP client and a DAP server uses some underlying already existing protocol. Volume 2 discusses the appropriate choices for the underlying protocol. Whatever protocol is used, the actual Data Response must conform to the Multi-Part MIME encoding described in Section ?.

In addition to these data objects, a DAP server MAY provide additional “services” which clients may find useful. For example, many DAP-compliant servers provide HTML-formatted representations or ASCII representations of a data source’s structure and data. Such additional services are discussed in Volume 2 of this specification.

# Characterization of a Data Source

The DAP characterizes a data source as a collection of variables, dimensions, and enumeration types. Each variable consists of a name, a type, a value, and a collection of Attributes. Dimensions have a name and a size. Enumerations list names and values of the enumeration constants. These elements may be grouped into collections using the concept of a “group” that has an identifier and defines a naming scope for the elements within it. Groups may contain other groups.

The distinction between information in a variable and in an Attribute is somewhat arbitrary. However, the intention is that Attributes hold information that aids in the interpretation of data held in a variable. Variables, on the other hand, hold the primary content of a data source.

Section ? provides a formal syntax for DAP DMR characterizations. It is defined using the RelaxNG standard [] for describing the context-free syntax of a class of XML documents, the DMR in this case. The following discussion closely follows that RELAX NG syntax specification. It should be noted that any syntax specification requires a specification of the lexical elements of the syntax. The XML specification [] provides most of the lexical context for the syntax, but there are certain places where additional lexical elements must be used. Section ? describes those additional lexical elements, and those elements are discussed at appropriate points in the following discussion.

Since the syntax is context-free, there are semantic limitations on what is legal in a DMR. These semantic limitations are defined at appropriate places in the following documentation. It should also be noted that if there are conflicts between what is described here and the RelaxNG syntax, then the syntax takes precedence.

# DMR Declarations

## XML Escaping Within the DMR

Any string of characters appearing within an XML attribute in the DMR must apply the standard XML escapes. Specifically, any attribute value containing any of the following characters must replace them with the corresponding XML escape form.

|  |  |
| --- | --- |
| Character | Escaped Form |
| & | &amp; |
| < | &lt; |
| > | &gt; |
| " | &quot; |

So for example, given the occurrence of the attribute

name="&<>"

it must be re-written to this form

name="&amp;&lt;&gt;"

## Names

A name (aka identifier) in DAP4 consists of a sequence of any legal non-control UTF-8 characters. A control character is any UTF-8 character in the inclusive range [0x00 – 0x1F].

## Fully Qualified Names

Every object in a DAP4 Dataset has a Fully Qualified Name (FQN), which provides a way to unambiguously name variables and groups in a dataset and which can be used in several contexts such as in the DMR in a *constraint expression* (see Section ?). These FQNs follow the common conventions of names for lexically scoped identifiers. In DAP4 two kinds of lexical items provide lexical scoping: Groups and Structures . Just as with hierarchical file systems or variables in many programming languages, a simple grammar formally defines how the names are built using the names of the FQN’s components (see Appendix ?). Consider the following simple dataset,which contains a structure name “inner” within a Structure named “outer” all contained in the Group “G”

<Group name=”G”>

<Structure name=”places”>

<String name=”name”/>

<Structure name=”weather”>

<Float64 name=”temperature”/>

<Float64 name=”dew\_point”/>

</Structure>

</Structure>

</Group>

The FQN for the field ‘temperature’ is ‘/G/places.weather.temperature’. As is the case with Structure variables, Groups can be nested to form hierarchies, too, and this example shows that case.

<Group name=”G”>

<Group name=”environmental\_data”>

<Structure name=”places”>

<String name=”name”/>

<Structure name=”weather”>

<Float64 name=”temperature”/>

<Float64 name=”dew\_point”/>

</Structure>

</Structure>

</Group>

<Group name=”demographic\_data”>

…

</Group

</Group>

The FQN to the field ‘temperature’ in the dataset shown is ‘/G/environmental \_data/places.weather.temperature’

Notes:

1. Every dataset has a ‘root Group’. Whatever name that group has is ignored for the purposes of forming the FQN and instead is treated as if it has the empty name (“”).
2. There is no limit to the nesting of groups or the nesting of Structures.

The characters “/” and “.” have special meaning in the context of a fully qualified name. This means that if a name is added to the FQN and that name contains either of those two characters, then those characters must be specially escaped so that they will not be misinterpreted. The defined escapes are as follows.

|  |  |
| --- | --- |
| Character | Escaped Form |
| . | \. |
| / | \/ |
| \ | \\ |

Note that the escape character itself must be escaped. Also note that this form of escape using ‘\’ is independent of any required XML escape (Section ?).

## Non-Data Bearing Declarations versus Data Bearing Declarations

The declarations in a DMR can be grouped into two classes. One class is non-data bearing. That is, it provides syntactic or structural metadata about a dataset. The non-data bearing declarations are Groups, Dimensions, and Enumerations. Such declarations do not contain data values themselves, although they may define constants such as the dimension size. In many cases these declarations will not be explicitly represented in the original dataset. Instead, their existence and value(s) will be inferred based on various standards and conventions. The data bearing classes of declarations are Variables and Attributes. These elements of the data model are used to house data values or semantic metadata read from the dataset (or, in the latter case) synthesized from the values and standards/conventions that the dataset is known to follow.

## Groups

A group is specified using this XML form:

|  |
| --- |
| <Group name=“name”>  …  </Group> |

A group defines a name space and contains other DAP elements. Specifically, it can contain groups, variables, dimensions, and enumerations. The fact that groups can be nested means that the set of groups in a DMR form a tree structure. For any given DMR, there exists a root group that is the root of this tree.

A nested set of groups defines a variety of name spaces and access to the contents of a group is specified using a notation of the form “/g1/g2/…/gn”. This is called a “path”. By convention “/” refers to the root group. Thus the path “/g1/g2/g3” indicates that one should start in the root group, move to group g1 within that root group, then to group g2 within group g1, and finally to group g3. This is more fully described in the section on Fully Qualified names (Section ?).

For comparison purposes, DAP groups correspond to netCDF-4 groups and not to the more complex HDF5 Group type.

The top-level group, called the root group, serves the role that the “Dataset” construct did in DAP2. As mentioned in Section ? on FQNs, the name of the root group is treated as the empty string for purposes of forming an FQN. In addition, the root group (and only the root group) can have the following special attributes.

|  |  |  |
| --- | --- | --- |
| Attribute Name | Type | Description |
| dapVersion | String | The version number of the DAP protocol. Should be “4.0” |
| dmrVersion | String | Should be “1.0” |
| ns | URI |  |
| base | URI |  |

Semantic Limitations

1. If declared, Groups must be named. This includes the root group, but for that group the name is ignored for the purposes of forming fully qualified names.
2. A Group can contain any number of objects, including other Groups.
3. Each Group declares a new lexical scope for the objects it contains.
4. A Group cannot have dimensions and a Group cannot be defined within a Structure.

## Dimensions

A dimension declaration is specified using this XML form.

|  |
| --- |
| <Dimension name=“name” size=“size”/> |

The size is a positive integer with a maximum value of 2\*\*63-1. A dimension declaration will be referenced elsewhere in the DMR by specifying its name. It should also be noted that anonymous dimensions also exist. They have a size but no name. Anonymous dimensions SHOULD NOT be declared.

Semantic Limitations

1. Dimension declarations are not associated with a data type.
2. Dimension sizes that are not ‘anonymous’ MUST be a capable of being represented as a signed 64-bit integer.

## Enumeration Types

An enumeration type defines a set of names with specific values: enumeration constants. As will be seen in Section ?, enumeration types may be used as the type for variables or attributes. The values that can be assigned to such typed objects must come from the set of enumeration constants.

An enumeration type specifies a set of named, integer constants. When a data source has a variable of type 'Enumeration' a DAP 4 server MUST represent that variable using a specified *integer type*, up to an including a 64-bit unsigned integer.

An Enumeration type is declared using this XML form.

|  |
| --- |
| <Enumeration name=“name”> |
| <basetype=”Byte|Int8|UInt8|Int16|UInt16|Int32|UInt32|Int64|UInt64”/> |
| <EnumConst name=“name” value=”integer”/> |
| … |
| </Enumeration> |

Semantic Limitations

1. The optional “basetype” XML attribute defines the type for the value XML attribute of each enumeration constant. This basetype must be one of the integer types (see Section ?). If unspecified, then it defaults to the Atomic type “Int32”.

## Atomic Types

The DAP4 specification assumes the existence of certain pre-defined, declared types called atomic types. As their name suggests, atomic data types are conceptually indivisible. Atomic variables are used to store integers, real numbers, strings and URLs. There are five classes of atomic types, with each family containing one or more variations: integer, floating-point, string, enumerations, and opaque.

### Integer Types

The integer types are summarized in Table 1. The lexical structure for integer constants is defined in Appendix ?.

1. The DAP Integer Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Int8 | Signed 8-bit integer | [-(2^7), (2^7) - 1] |
| UInt8 | Unsigned 8-bit integer | [0, (2^8) - 1] |
| Byte | Synonym for UInt8 | [0, (2^8) - 1] |
| Char | Synonym for UInt8 | [0, (2^8) - 1] |
| Int16 | Signed 16-bit integer | [-(2^15), (2^15) - 1] |
| UInt16 | Unsigned 16-bit integer | [0, (2^16) - 1] |
| Int32 | Signed 32-bit integer | [-(2^31), (2^31) - 1] |
| UInt32 | Unsigned 32-bit integer | [0, (2^32) - 1] |
| Int64 | Signed 64-bit integer | [-(2^63), (2^63) - 1] |
| UInt64 | Unsigned 64-bit integer | [0, (2^64) - 1] |

Note that for historical reasons, the Char type is defined to be a synonym of UInt8, this mean that technically, the Char type has no associated character set encoding. However, servers and clients are free to infer typical character semantics to this type. The inferred character set encoding is chosen purely at the discretion of the server or client using whatever conventions they agree to use.

### Floating-point Types

The floating-point data types are summarized in Table 2. The two floating-point data types use IEEE 754[] to represent values. The two types correspond to ANSI C’s float and double data types. The lexical structure for floating point constants is defined in Section ?.

1. The DAP Floating-Point Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| Float32 | 32-bit Floating-point number | Refer to the IEEE Floating Point Standard [] |
| Float64 | 64-bit Floating-point number | Refer to the IEEE Floating Point Standard [] |

### String Types

The string data types are summarized in Table 3. Again, the lexical structure for these is defined in Section ?.

Strings are individually sized. This means that in an array of strings, for example, each instance of that string MAY be of a different size.

1. The String Data types.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Range of Legal Values |
| String | A variable length string of UTF-8 characters | As defined in [] |
| URI | A Uniform Resource Identifier | As defined in IETF RFC 2396[3] |

### The Opaque Type

The XML scheme for declaring an Opaque type is as follows.

|  |
| --- |
| <Opaque> |

The Opaque type is use to hold objects like JPEG images and other Binary Large Object (BLOB) data that have significant internal structure which might be understood by clients (e.g., an image display program) but that would be very cumbersome to describe using the DAP4 built-in types. Defining a variable of type “Opaque” does not communicate any information about its content, although an attribute could be used to do that.

Semantic Limitations

1. The content of an opaque object is completely un-interpreted by the DAP4 implementation. The Opaque type is an Atomic Type, which might seem odd because instances of Opaque can be of different sizes. However, by thinking of Opaque as equivalent to a byte-string type, the analogy with strings makes it clear that it should be an Atomic type.

### A Note Regarding Implementation of the Atomic Types

When implementing the DAP, it is important to match information in a data source or read from a DAP response to the local data type which best fits those data. In some cases an exact match may not be possible. For example Java lacks unsigned integer types [13]. Implementations faced with such limitations MUST ensure that clients will be able to retrieve the full range of values from the data source. If this is impractical, then the server or client may implement this rule by hiding the variable in question or returning an error.

## Container Types

There is currently one container type, namely the Structure type.

### The Structure Type

A Structure groups a list of variables so that the collection can be manipulated as a single item. The variables in a Structure may also be referred to as “fields” to conform to conventional use of that term, but there is otherwise no distinction between fields and variables. The Structure’s fields MAY be of any type, including the Structure type. The order of items in the Structure is significant only in relation to the serialized representation of that Structure.

## Variables

Each variable in a data source MUST have a name, a type and one or more values. Using just this information and armed with an understanding of the definition of the DAP data types, a program can read any or all of the information from a data source.

The DAP variables come in several different types. There are several atomic types, the basic indivisible types representing integers, floating point numbers and the like, and a container type — the Structure type — that supports aggregation of other variables into a single unit. A container type may contain both atomic typed variable as well as other container typed variables, thus allowing nested type definitions.

The DAP variables describe the data when it is being transferred from the server to the client. It does not necessarily describe the format of the data inside the server or client. The DAP defines, for each data type described in this document, a serialized representation, which is the information actually communicated between DAP servers and DAP clients. The serialized representation consists of two parts: the declaration of the type and the serialized encoding of its value(s). The data representation is presented in Section ? and Appendix ?.

### Arrays

Most (but not all) types may be arrays. An Array is a multi-dimensional indexed data structure. An Array’s member variable MUST be of some DAP data type. Array indexes MUST start at zero. Arrays MUST be stored in row-magjor order (as is the case with ANSI C), which means that the order of declaration of dimensions is significant. The size of each Array’s dimensions MUST be given, except for variable length dimensions. The total number of elements in an Array MUST NOT exceed 2^64-1. There is no prescribed limit on the number of dimensions an Array may have except that the foregoing limit on the total number of elements MUST NOT be exceeded. The number of elements in an Array is fixed as that given by the size(s) of its dimension(s), except when the array has a variable length dimension.

Semantic Limitations

1. Simple variables (see below) MAY be arrays.
2. Structures MAY be arrays.

### Simple Variables

A simple, dimensioned variable is declared using this XML form.

|  |
| --- |
| <Int32 name=”name”>  <Dim name=”FQN”/>  …  <Dim size=”integer”/>  …  <Dim size=”\*”/>  </Int32> |

A simple variable is one whose type is one of the Atomic Types (see Section ?). The name of the Atomic Type (Int32 in this example) is used as the XML element name. Within the body of that element, it is possible to specify zero or more dimension references. A dimension reference (<Dim…/>) MAY refer to a previously defined dimension declaration. It MAY also define an anonymous dimension with no name, but with a size. It MAY also define a variable length dimension using a size of “\*”.

Semantic Limitations

1. N.A.

### Dimension Ordering.

Consider this example.

|  |
| --- |
| <Int32 name=”i”> |
| <Dim name=”/d1”/> |
| <Dim name=”/d2”/> |
| … |
| <Dim name=”/dn”/> |
| </Int32> |

The dimensions are considered ordered from top to bottom. From this, a corresponding left-to-right order [d1][d2]…[dn] can be inferred where the top dimension is the left-most and the bottom dimension is the right-most. The assumption of row-major order means that in enumerating all possible combinations of these dimensions, the right-most is considered to vary the fastest. The terms “right(most)” or “left(most”) refer to this left-to-right ordering of dimensions.

### Structure Variables

As with simple variables, a structure variable specifies a type as well as any dimension for that variable. The type, however, is a Structure.

#### Structures

The XML scheme for a Structure typed variable is as follows.

|  |
| --- |
| <Structure name=”name”>  {variable definition}  {variable definition}  …  {variable definition}  <Dim name=”FQN”/>  …  <Dim name=”FQN”/>  </Structure> |

The Structure contains within it a list of variable definitions (Section ?). For discussion convenience, each such variable may be referred to as a “field” of the Structure. The list of fields may optionally be followed with a list of dimension references indicating the dimensions of the Structure typed variable.

Semantic Limitations

1. Structures MAY be dimensioned.

### Coverage Variables and Maps

A “Discrete Coverage” is a concept commonly found in many disciplines, where the term refers to a sampled function with both its domain and range explicitly enumerated by variables. DAP2 uses the name ‘Grid’ to denote what the OGC calls a ‘rectangular grid’ [cite: the abstract coverage spec.]. DAP4 expands on this so that other types of discrete coverages (hereafter ‘coverage()s’) can be explicitly represented.

In DAP4, the range for a coverage is the values of a (simple or container) variable that includes a specific set of ‘maps’ or ‘coordinate variables’ that define the domain for the sampled function. Taken as whole, this type of variable is called a “grid” for convenience sake.

Using OGC coverage terminology, we have this.

1. The maps specify the “Domain”
2. The array specifies the “Range”
3. The Grid itself is a “Coverage” per OGC.
4. The Domain and Range are sampled functions

A map is defined using the following XML scheme.

|  |
| --- |
| <Map name=”FQN for some variable defined in the DMR” /> |

An example might look like this.

|  |
| --- |
| <Float32 name=”A”>  <Dim name=”/lat”/>  <Dim name=”/lon”/>  <Map name=”/lat”/>  <Map name=”/lon”/>  </Float32> |

Where the map variables are defined elsewhere like this.

|  |
| --- |
| <Float32 name=”lat”>  <Dim name=”/lat”/>  </Float32  <Float32 name=”/lon”>  <Dim name=”/lon”/>  </Float32> |

The containing variable, A in the example, will be referred to as the “array variable”.

Semantic Limitations

1. Each map variable MUST have a rank no more than that of the array.
2. An array variable can have as many maps as desired.
3. The dimensions of the array variable may not contain duplicates so A[x,x] is disallowed.
4. Any map duplicates are ignored and the order of declaration of the maps is irrelevant.
5. A Map variable may not have a variable length dimension.
6. The fully qualified name of a map must either be in the same lexical scope as the array variable, or the map must be in some enclosing scope.
7. The set of named “associated dimensions for a map must be a subset of the set of named “associated dimensions” for the array variable.

The term “associated dimensions” is computed as follows.

1. The set of associated dimensions is initialized to empty.
2. For each element mentioned in the fully qualified name (FQN) of the map or the array variable, add any named dimensions associated with FQN element to the set of associated dimensions (removing duplicates, of course).

In practice, the means that an array variable or map variable must take into account any dimensions associated with any enclosing dimensioned Structure.

## Attributes and Arbitrary XML

### Attributes

Attributes are defined using the following XML scheme.

|  |
| --- |
| <Attribute name=”name” type=”atomic type name”>  <Namespace href=”http://netcdf.ucar.edu/cf”/>  <Value value=”value”/>  …  <Value value=”value”/>  </Attribute>  <Attribute name=”name” type=”container name”>  <Namespace href=”http://netcdf.ucar.edu/cf”/>    <Attribute name=”name” type=”…”>  …  </Attribute>  …  <Attribute name=”name” type=”…”>  …  </Attribute>  </Attribute> |

In DAP4, Attributes (not to be confused with XML attributes) are tuples with four components:

Name

Type (one of the defined atomic types such as Int16, String, etc.), or a child attribute container

Vector of values

One or more Namespaces (optional)

This differs slightly from DAP2 Attributes because the *namespace* feature has been added, although clients can choose to ignore it. For more about namespaces, refer to Section ?. The intent of including the namespace information is to simplify interactions with semantic web applications where certain schemas or standards have formal definitions of attributes.

Attributes are typically used to associate semantic metadata with the variables in a data source. Attributes are similar to variables in their range of types and values, except that they are somewhat limited when compared to those for variables: they cannot use structure types

Attributes defined at the top-level within a group are also referred to as “group attributes”. Attributes defined at the top-level of the default or root group are “global attributes,” which many file formats such as HDF4 or netCDF formally recognize.

While the DAP does not require any particular Attributes, some may be required by various metadata conventions. The semantic metadata for a data source comprises the Attributes associated with that data source and its variables[14]. Thus, Attributes provide a mechanism by which semantic metadata may be represented without prescribing that a data source use a particular semantic metadata convention or standard.

Semantic Limitations

1. DAP4 explicitly treats an attribute with one value as an attribute whose value is a one-element vector.
2. The following types are allowed for Attributes:
3. All of the Atomic types as well as containers are allowed as the type for an attribute
4. Attribute value constants MUST conform to the appropriate constant format for the given attribute type and as defined in Appendix ?.

### Arbitrary XML content

By supporting an explicit type to hold “arbitrary XML” markup, DAP4 provides a way for the protocol to transport information encoded in XML along with the attributes read from the dataset itself. This has proved very useful in work with semantic web software.

In an XML representation of DAP4, the name is optional, the XML element is *<OtherXML/>* and there are no *<value/>* elements because the “other xml” appears as the content of the *<OtherXML/>* element. The text of the otherXML element must be valid XML and must be distinct from the XML markup used to encode elements of the DAP4 data model (i.e., in a practical sense, the content of an <OtherXML> attribute will be in a namespace other than DAP4). XML content may appear anywhere that an attribute may appear.

### Attribute and OtherXML Specification and Placement

Attribute and OtherXML declarations MAY occur within the body of the following XML elements: Group, Dimension, Variable, and Structure.

## Namespaces

All elements of the DMR, Groups, Dimensions, Variables, and Attributes can contain an associated Namespace element. The namespace’s value is defined in the form of an XML style URI string defining the context for interpreting the element containing the namespace. Suppose, hypothetically, that we wanted to specify that an Attribute is to be interpreted as a CF convention []. One might specify this as follows.

|  |
| --- |
| <Attribute name=”latitude”>  <Namespace href=”http://cf.netcdf.unidata.ucar.edu”/>  …  </Attribute> |

Note that this is not to claim that this is how to specify a CF convention; this is purely illustrative.

# Data Representation

Data can be an elusive concept. Data may exist in some storage format on some disk somewhere, on paper somewhere else, in active memory on some server, or transmitted along some wire between two computers. All these can still represent the same data. That is, there is an important distinction to be made between the data and its representation. The data can consist of numbers: abstract entities that usually represent measurements of something, somewhere. Data also consist of the relationships between those numbers, as when one number defines a time at which some quantity was measured.

The abstract existence of data is in contrast to its concrete representation, which is how we manipulate and store it. Data can be stored as ASCII strings in a file on a disk, or as twos-complement integers in the memory of some computer, or as numbers printed on a page. It can be stored in netCDF[], HDF5[], GRIB[], a relational database, or any number of other digital storage forms.

The DAP specifies a particular representation of data, to be used in transmitting that data from one computer to another. This representation of some data is sometimes referred to as the serialized representation of that data, as distinguished from the representations used in some computer’s memory. The DAP standard outlined in this document has nothing at all to say about how data is stored or represented on either the sending or the receiving computer. The DAP transmission format is completely independent of these details.

## Response Structure

The DAP4 Data Response uses a format very similar to that used for DAP2; the data payload is broken into two logical parts. The first part holds metadata describing the names and types of the variables in the response while the second part holds the values of those variables. DAP4 provides several improvements over the DAP2 response, however.

The metadata information, sent as a preface to the Data Response, is the DMR limited to just those variables included in the response. DAP attributes may be included, but MAY be ignored by the receiving client.

Data values in the Data Response consist of a byte order indicator followed by the binary data for each variable in the order they are listed in the DMR given as the response preface. DAP4 uses a *receiver makes it right* encoding, so the servers simply write out binary data as they store it with the exceptions that floating-point data must be encoded according to IEEE 754 and Integer data must use twos-compliment notation for signed types. Clients are responsible for performing byte-swapping operations needed to compute using the values retrieved.

The Data Response is encoded using chunking scheme that breaks it into *N* parts where each part is prefixed with a *chunk type* and *chunk byte count* header. Chunk types include *data* and *error* types, making it simple for servers to indicate to clients that an error occurred during the transmission of the Data Response and (relatively) simple for clients to detect that error.

As with DAP2, the response describe here is a document that can be stored on disk or sent as the payload using a number of network transport protocols, HTTP being the primary transport in practice. However, any protocol that can transmit a document can be used to transmit these responses. As such, all critical information needed to decode the response is completely self-contained.

In the rest of this section we will describe the Data Response in the context of DAP4 using HTTP as its transport protocol.

### Structure of the DMR Preface

The first part of the Data Response always contains the DMR. The Data Response, when DAP is using HTTP as a transport protocol, is the payload for an HTTP response, is separated from the last of the HTTP response’s MIME headers by a single blank line, which MIME defines as a carriage return followed by a new line. The Data Response itself uses this as a separator between the DMR preface and the binary data. The logical organization of the Data Response is shown below

|  |
| --- |
| CRNL |
| <<DMR here>>  CRNL |
| <<Binary information>> |
|  |

### Structure of the Binary Data Part

The binary data part of the Data Response starts with a four-byte *byte-order header*. This encodes the byte order of the data as sent by the server. The client uses this information to transform the binary data according so it can use those values in computation (i.e., receiver-makes-it-right). Following the byte-order header, the values for each atomic or array variable appear according to their position in the DMR.

### How the Chunked Encoding Affects into the Data Response Format

In a sense, the chunked encoding does not affect the format of the Data Response at all. Conceptually, the entire Data Response is built and then passed through a ‘chunking encoder’ transforming the document into one that is broken up into a series of chunks. That ‘chunked document’ is the sent as the payload of some transport protocol, e.g., HTTP. In practice, that would be a wasteful implementation because a server would need to hold the entire response in memory. A better implementation would, for HTTP, write the initial parts of the HTTP response (its response code and headers) and then use a pipeline of filters to perform the encoding operations. The intent of the chunking scheme is to make it possible for servers to build responses in small chunks, and once they know those parts have been built without error, send them to the client. Thus a server should choose the chunk size to be small enough to fit comfortably in memory but large enough to limit the amount of overhead spent by the software that encodes and decodes those chunks. When an error is detected, the normal flow of building chunks and sending the data along is broken and an error chunk should be sent.



## The DAP4 Serialized Representation (DSR)

Given a DMR and the corresponding data, the serialized representation is formally described in this section.

### A note on dimension ordering

Consider this example.

|  |
| --- |
| <Int32 name=”i”> |
| <Dimension name=”d1”/> |
| <Dimension name=”d2”/> |
| … |
| <Dimension name=”dn”/> |
| </Int32> |

The dimensions are considered ordered the top-to-bottom lexically. This order is linearized into a corresponding left-to-right order [d1][d2]…[dn]. The assumption of row-major order means that in enumerating all possible combinations of these dimensions, the rightmost is considered to vary the fastest. The terms “right(most)” or “left(most”) refer to this ordering of dimensions.

### Order of Serialization

The data appearing in a serialized representation is the concatenation of the variables specified in the tree of Groups within a DMR, where the variables in a group are taken in depth-first, top-to-bottom order. The term “top-to-bottom” refers to the lexical ordering of the variables in an XML document specifying a given DMR.

If a variable is a Structure variable, then its data representation will be the concatenation of the variables it contains, which will appear in top-to-bottom order.

If a variable has dimensions, then the contents of each dimensioned data item will appear concatenated and taken in row-major order.

### Variable Representation in the Absence of Variable Length Dimensions

Given a dimensioned variable, with no dimension being variable length, it is represented as the N scalar values concatenated in row-major order.

If the variable is scalar, then it is represented as a single scalar value.

#### Numeric Scalar Atomic Types

For the numeric atomic types, scalar instances are represented as follows. In all cases a consistent byte ordering is assumed, but the choice of byte order is at the discretion of the program that generates the serial representation, typically a server program.

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Representation |
| Int8 | Signed 8-bit integer | 8 bits |
| UInt8 | Unsigned 8-bit integer | 8 bits |
| Byte | Unsigned 8-bit integer | Same as UInt8 |
| Char | Unsigned 8-bit integer | Same as UInt8 |
| Int16 | Signed 16-bit integer | 16 bits |
| UInt16 | Unsigned 16-bit integer | 16 bits |
| Int32 | Signed 32-bit integer | 32-bits |
| UInt32 | Unsigned 32-bit integer | 32-bits |
| Int64 | Signed 64-bit integer | 64-bits |
| UInt64 | Unsigned 64-bit integer | 64-bits |
| Float32 | 32-bit IEEE floating point | 32-bits |
| Float64 | 64-bit IEEE floating point | 64-bits |

In narrative form: all numeric quantities are used as a raw, unsigned vector of N bytes, where N is 1 for Char, Int8, and UInt8; it is 2 for Int16 and UInt16; it is 4 for Int32, UInt32, and Float32; and it is 8 for Int64, UInt64, and Float64. If the server chooses to byte swap the values, then the following swapping rules are used. Not that a size of 16 bytes is also included. This is used solely to represent checksums as 128-bit unsigned integers (see Section ?).

|  |  |  |
| --- | --- | --- |
| Size (in bytes) | Byte Swapping Rules | |
| 1 | Not Applicable. |  |
| 2 | Byte 0 -> Byte 1  Byte 1 ->Byte 0 |  |
| 4 | Byte 0 -> Byte 3  Byte 1 ->Byte 2  Byte 2 -> Byte 1  Byte 3 ->Byte 0 |  |
| 8 | Byte 0 -> Byte 7  Byte 1 ->Byte 6  Byte 2 -> Byte 5  Byte 3 ->Byte 4 | Byte 4 -> Byte 3  Byte 5 ->Byte 2  Byte 6 -> Byte 1  Byte 7 ->Byte 0 |
| 16 | Byte 0 -> Byte 15  Byte 1 ->Byte 14  Byte 2 -> Byte 13  Byte 3 ->Byte 12  Byte 4 -> Byte 11  Byte 5 ->Byte 10  Byte 6 -> Byte 9  Byte 7 ->Byte 8 | Byte 8 -> Byte 7  Byte 9 ->Byte 6  Byte 10 -> Byte 5  Byte 11 ->Byte 4  Byte 12 -> Byte 3  Byte 13 ->Byte 2  Byte 14 -> Byte 1  Byte 15 ->Byte 0 |

#### Variable-Length Scalar Atomic Types

|  |  |  |
| --- | --- | --- |
| Type Name | Description | Representation |
| String | Vector of 8-bit bytes representing a UTF-8 String | The number of bytes in the string (in UInt64 format) followed by the bytes. |
| URL | Vector of 8-bit bytes representing a URL | Same as String |
| Opaque | Vector of un-interpreted 8-bit bytes | The number of bytes in the vector (in UInt64 format) followed by the bytes. |

In narrative form, instances of String, Opaque, and URL types are represented as a 64 bit length (treated as UInt64) of the instance followed by the vector of bytes comprising the value.

#### Structure Variable Representation

A Structure typed variable is represented as the concatenation of the representations of the variables contained in the Structure taken in lexical top-to-bottom order. This representation may be recursive if one of the variables itself is a Structure variable. Dimensioned structures are represented in a form analogous to dimensioned variables of atomic type. The Structure array is represented by the concatenation of the instances of the dimensioned Structure, where the instances are listed in row-major order.

### Variable Representation in the Presence of Variable-Length Dimensions

Given a dimensioned variable, with at least one dimension being variable length, it is represented as follows.

The variable is represented as the concatenation of N “variable length vectors”. N is determined by taking the cross product of the dimensions sizes, left to right, up to, but not including, the first variable length dimension. For example, an array of the form *Int32 A[2][3][\*][7][8][\*]* has an element count (N) of 2x3 = 6 and all dimensions to the right and including the leftmost varying length dimension are ignored in computing N.

In our example, there will be 6 (2\*3) variable-length vectors concatenated together. Note that the length, L, of each of the variable length vectors may be different for each vector. Section ? provides some examples in detail.

This representation is recursive in that the values in a variable-length vector may themselves be variable-length vectors (as in our example above). Each such vector is encoding using the rules above. In our example, then, there will be 6 variable-length vectors, each of which in turn consists of 56 (7\*8) nested variable length vectors.

Each variable length vector consists of a length, L say, in UInt64 form and giving the number of elements for a specific occurrence of the variable-length dimension. The count, L, is then followed by L instances of the type of the variable (Int32 in this case because the type of the array A is Int32).

### Checksums

Checksums will be computed for the values of all the variables at the top-level of each Group in the response. The checksum value will follow the value of the variable. The checksum algorithm defaults to MD5 (chosen primarily for performance reasons).

Checksum values will be written as 128-bit values using the endian representation specified for the serialized form.

### Historical Note

The encoding described in Section ? is essentially the same as the serialization form of the DAP2 protocol [cite], but has been extended to support arrays with varying dimensions and stripped of redundant information added by various XDR implementations.

The DAP4 Serialization rules are derived from, but not the same as, XDR []. The differences are as follows.

1. Values are encoded using the byte order of the server. This is the so-called “receiver makes it right” rule.
2. No padding is used.
3. Floating point values always use the IEEE 754 standard.
4. One and two-byte values are not converted to four byte values.

## Example responses

In these examples, spaces and newlines have been added to make them easier to read. The real responses are as compact as they can be. Since this proposal is just about the form of the response - and it really focuses on the BLOB part - there is no mention of 'chunking.' For information on how this BLOB will/could be chunked, see Section ?. NB: Some poetic license used in the following and the checksums for single integer values seems silly, but these are really simple examples.

### A single scalar

|  |
| --- |
| ... |
| Content-Type: appliaction/vnd.opendap.org.dap4.data |
|  |
| <Group name=”foo”> |
| <Int32 name=”x”/> |
| </Group> |
|  |
| x |
| <<checksum>> |

### A single array

|  |
| --- |
| <Group name=”foo”> |
| <Int32 name=”x”? |
| <Dimension size=”2”/> |
| <Dimension size=”4”/> |
| </Int32> |
| </Group> |
|  |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| <<checksum>> |

### A single structure

|  |
| --- |
| <Group name=”foo”> |
| <Structure name=”S”> |
| <Int32 name=”x”> |
| <Dim size=”2”> |
| <Dim size=”4”> |
| </Int32> |
| <Float64 name=”y”/> |
| </Structure |
| </Group> |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| <<checksum>> |
|  |

Note that in this example, there is a single variable at the top-level of the root Group “/” and that is S, so it is S for which we compute the checksum.

### An array of structures

|  |
| --- |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dim size=”2”/> |
| <Dim size=”4”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dim size=”3”/> |
| </Structure> |
| </Group> |
|  |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| <<checksum>> |

### A single varying array (one varying dimension)

|  |
| --- |
| <Group name=”foo”> |
| <String name=”s”/> |
| <Int32 name=”a”> |
| <Dim size=”\*”/> |
| </Int32> |
| <Int32 name=”x”> |
| <Dim size=”2”/> |
| <Dim size=”\*”/> |
| </Int32> |
| </Group> |
|  |
| 16 This is a string |
| <<checksum>> |
|  |
| 5 a0 a1 a2 a3 a4 |
| <<checksum>> |
|  |
| 3 x00 x01 x02 6 x00 x01 x02 x03 x04 x05 |
| <<checksum>> |
|  |
|  |

Notes:

1. The checksum calculation includes only the values of the variable, not the prefix length bytes.
2. The varying dimensions are treated 'like strings' and prefixed with a length count. In the last of the three variables, the array x is a 2 by varying array with the example's first 'row' containing 3 elements and the second 6.

### A single varying array (two varying dimensions)

The array ‘x’ has two dimensions, both of which vary in size. In the example, at the time of serialization ‘x’ has three elements in its outer dimension and those have three, six and one element, respectively. Because these are ‘varying’ dimentions, the size of each much prefix the actual values.

|  |
| --- |
| <Group name=”foo”> |
| <Int32 name=”x”> |
| <Dim size=”\*”/> |
| <Dim size=”\*”/> |
| </Int32> |
| </Group> |
|  |
|  |
|  |
| 3 |
|  |
| 3 x00 x01 x02 |
|  |
| 6 x10 x11 x12 x3 x14 x15 |
|  |
| 1 x20 |
| <<checksum>> |
|  |
| A varying array of structures |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dim size=”4”/> |
| <Dim size=”4”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dim size=”\*”/> |
| </Structure> |
| </Group> |
|  |
| 2 |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
|  |
| x00 x01 x02 x03 x10 x11 x12 x13 |
| y |
| <<checksum>> |

Note that two rows are assumed.

### A varying array of structures with fields that have varying dimensions

|  |
| --- |
| <Group name=”foo”> |
| <Structure name=”s”> |
| <Int32 name=”x”> |
| <Dim size=”2”/> |
| <Dim size=”\*”/> |
| </Int32> |
| <Float64 name=”y”/> |
| <Dim size=”\*”/> |
| </Structure> |
| </Group> |
|  |
| 3 |
|  |
| 1 x00 4 x10 x11 x12 x13 |
| y |
|  |
| 3 x00 x01 x02 2 x10 x11 |
| y |
|  |
| 2 x00 x01 2 x10 x11 |
| y |
| <<checksum>> |

# DAP4 Chunked Data Representation

An important capability for DAP4 is supporting client in determining when a data transmission fails. This is especially difficult when sending binaray data (Section ?). In order to support such a capability, the DAP4 protocol uses a simplified variation on the HTTP/1.1 chunked transmission format [cite] to serialize the data part of the response document so that errors are simple to detect. Furthermore, this format is independent of the form or content of that part of the response, so the same format can be used with different response forms or dropped when/if DAP is used with protocols that support out-of-band error signaling, simplifying our ongoing refinement of the protocol.

The data part of a response document is “chunked” in a fashion similar to that outlined in HTTP/1.1. However, in addition to a prefix indicating the size of the chunk, DAP4 includes a chunk-type code. This provides a way for the receiver to know if the next chunk is part of the data response or if it contains an error response (Section ?). In the latter case, the client should assume that the data response has ended, even though the correct closing information was not provided.

Each chunk is prefixed by a chunk header consisting of a chunk type and byte count, all contained in a single four-byte word and encoded using network byte order. The chunk type will be encoded in the high-order byte of the four-byte word and chunk size will be given by the three remaining bytes of that word. The maximum chunk size possible is 2^24 (16777216) bytes. Immediately following the four-byte chunk header will be chunk-count bytes followed by another chunk header. More precisely the initial four bytes of the chunk are decoded using the following steps.

1. Treat the 32 bit header a single unsigned integer.
2. Convert the integer from network byte order to the local machine byte order by swapping bytes as necessary (see Section ?). Let the resulting integer be called H.
3. Compute the chunk type by the following expression: type = (H >> 24) & 0xff (Using C-language operators).
4. Compute the chunk length by the following expression: length = (H & 0x00ffffff) (Using C-language operators).

Three chunk-type types are defined in this proposal:

* Data –This chunk header prefixes the next chunk in the current data response
* Error – This chunk header prefixes an error message; the current data response has ended
* End – This chunk header is the last one for the current data response

## Chunked Format Grammar

|  |
| --- |
| chunked\_response: chunklist ; |
|  |
| chunklist: chunk | chunklist chunk; |
|  |
| /\* Semantic limitation: the number of bytes in the CHUNKDATA must be equal to SIZE\*/ |
| chunk: CHUNKTYPE SIZE CHUNKDATA; |

## Lexical Structure

|  |
| --- |
| /\* A single 8-bit byte, with the encoding 0 = data, 1 = error, 2 = end \*/ |
| CHUNKTYPE = '\x01'|'\x02'|'\0x03' |
|  |
| /\* A sequence of three 8-bit bytes, interpreted as an integer on network byte order \*/ |
| SIZE = [\0x00-\0xFF][\0x00-\0xFF][\0x00-\0xFF] |
|  |
| CHUNKDATA = [\0x00-\0xFF]\* |

## Error Chunk Schema

The error chunk is defined to be an XML document as defined by the following RELAX-NG schema.

|  |
| --- |
| <grammar xmlns="http://relaxng.org/ns/structure/1.0" |
| xmlns:doc="http://www.example.com/annotation" |
| datatypeLibrary="http://xml.opendap.org/datatypes/dap4" |
| ns="http://xml.opendap.org/ns/DAP/4.0#" |
| > |
| <start> |
| <ref name="error"/> |
| </start> |
| <define name="error"> |
| <element name="Error"> |
| <attribute name="errorcode"><data type="dap4\_integer"/></attribute> |
| <element name = "Message"><text/></Message> |
| <optional> |
| <interleave> |
| <element name = "Position"><text/></Message> |
| <element name = "Context"><text/></Message> |
| <element name = "OtherInformation"><text/></Message> |
| </interleave> |
| </optional> |
| </element> |
| </define> |

# Constraints

A request to a DAP4 server for either metadata (the DMR) or data may include a constraint expression. This constraint expression specifies which variables are to be returned and what subset of the data for each variable is to be returned.

It is important to define a minimal request language – a query language – to select information from a dataset on a server and obtaining in response a DMR and data corresponding to that request.

This section defines the syntax and semantics of the minimal request language that MUST be supported by all implementations. The method by which a server is provided with a query is specified in Volume 2. But as a typical example, if such a query were to be embedded in a URL, then it is presumed that it is prefixed with a “?” and is appended to the end of the URL.

## Syntax

The syntax of the minimal query language, also referred to as the “simple query” language, is as follows.

|  |
| --- |
| simplequery: /\*empty\*/ | constraintlist ; |
| constraintlist: constraint | constraintlist ',' constraint ; |
| constraint: variablesubset | namedslice ; |
| variablesubset: PATH structpath ; |
| structpath: ID dimset | structpath NAME dimset ; |
| dimset: /\*empty\*/ | slicelist ; |
| slicelist: slice | slicelist slice ; |
| slice: ‘[‘ INTEGER ‘]’ |
| | ‘[‘ INTEGER ‘:’ INTEGER ‘]’ |
| | ‘[‘ INTEGER ‘:’ INTEGER ‘:’ INTEGER ‘]’ |
| | ‘[‘ slicename ‘]’ ; |
| namedslice: slicename ‘=’ slice ; |
| slicename: ID ; |

The *variablesubset* rule specifies a subset of values for a variable as specified by the slices. The PATH lexical element is the same as the FQN path as defined in Section ?.

The *structpath* is almost the same as the FQN prefix as defined in that same Section. The difference is that each component (between '.' separators) of the *structpath* can have an optional *dimset* indicating the set of dimension slices to apply.

A *dimset* is either empty or is a *slicelist*.

A *slicelist* is a non-empty list of slices, where a slice indicates a subset of dimension indices. The first case of a slice (e.g. [5]) indicates a single dimension value, 5 in this case. The second case (e.g. [5:9] indicates the range of dimension values 5,6,7,8,9. The third case (e.g. [5:2:11]) indicates a range of dimension values separated by the stride (the middle values. Thus the example would be the dimension values 5,7,9,11. The fourth case (e.g. [time], shows the use of a named slice.

Note that unlike a suffix, intermediate structures in the *structlist* can have associated *dimset*s Thus we might have something like this. /g/S1[5][5:9].v[5:2:11].

A *namedslice* provides a way to define a *slice* and give it a *slicename*. The slice name has lexical type ID. The name, when enclosed in “[]” can be used anywhere a *slice* is legal. The goal of the *namedslice* is to ensure that the same *slice* is used consistently across multiple *variablesubsets* as a way to impose shared dimension semantics.

There are certain context sensitive constraints on *structpaths* and *slicelists*.

The terminal variable in the structpath must be an atomic-typed variable. [Do we want this restriction?]

The number of slices associated with a component in the structpath must correspond to the arity of that structure or the last, atomic-typed variable.

1. A slice name must be defined before it is used.

## Interpretation

Consider the following Array.

|  |
| --- |
| <Int32 name="A"> |
| <Dim size="d1"/> |
| <Dim size="d2"/> |
| ... |
| <Dim size="dn"/> |
| </Int32> |

where all of the dimension sizes, di, are integers.

Consider the following array subset query, where for the purposes of interpretation, all named slices are assumed to have been replaced with their defined slice.

|  |
| --- |
| A[start1:stride1:end1]...[startn:striden:endn] |

Where

for i=1 .. n, starti < di & endi < di & starti < endi & starti >= 0 & stridei >= 1 & endi >= 0.

The query selects the elements A[i1][i2]…[in] from A where ii is in the set {starti+stridei\*j} and where j=0..k such that starti+stridei\*k <= endi and starti+stridei\*(k+1) > endi.

Now consider the same array embedded in a dimensioned Structure.

|  |
| --- |
| <Structure name="S"> |
| <Int32 name="A"> |
| <Dim size="d3"/> |
| ... |
| <Dim size="dn"/> |
| </Int32> |
| <Dim size="d1"/> |
| <Dim size="d2"/> |
| </Structure> |

where all of the dimension sizes, d i , are again integers.

Consider the following subset query.

S[start1:stride1:end1][start2:stride2:end2].A[start3:stride3:end3]...[startn:striden:endn]

with conditions as before.

The query selects the Structure instances

S[i1][i2] where ii is in the set {starti+stridei\*j} and where j=0..k such that starti+stridei\*k <= endi and starti+stridei\*(k+1) > endi.

Then for each selected structure, the elements A[i3]…[in] are selected from that instance of A where ii is in the set {starti+stridei\*j} and where j=0..k such that starti+stridei\*k <= endi and starti+stridei\*(k+1) > endi.

The results of all of the selections of the instances of A are concatenated as the value of the whole query.

ReferencesIEEE Standard for Binary Floating-Point Arithmetic, ANSI/IEEE Std 754-1985, Digital Object Identifier: 10.1109/IEEESTD.1985.82928, Publication Year: 1985.

1. FQN Syntax

An FQN has two parts. First, there is the path, which refers to Group traversal, and second is the suffix, which refers to the traversal of Structures. An FQN is the concatenation of the path with the suffix and separated by the ‘/’ Character. The suffix may not exist if O is a group or is not a Structure typed variable.

Fully qualified names conform to the following syntax.

FQN: grouppath

| grouppath ‘/’ name

| grouppath ‘/’ structurepath

| grouppath ‘/’ structurepath ‘.’ name

grouppath: /\*empty\*/ | grouppath ‘/’ groupname

structurepath: /\*empty\*/ | structurepath ‘.’ structname

To write a path for an object O, follow these steps.

1. Locate the closest enclosing group G for O. If O is a group, then O and G will be the same.

2. Create the scope prefix for O by traversing a path through the Group tree to and including G. Concatenate the group names on that path and separating them with ‘/’. The name for root group is ignored, hence the FQN will begin with “/”.

If O is not a Structure typed variable, then we are done and the FQN for O is just the path. Otherwise, the suffix must be computed as follows.

1. Traverse the nested Structure declarations from G to O, including O, but not including G in the path. Traversal here means following the enclosing Structure typed variables until O is reached.

2. Concatenate the names on that suffix path and separating them with ‘.’ to create a suffix.

3. Create the final FQN as the concatenation of the path, the character ‘/’, and the suffix.

1. DAP4 Lexical Elements
   1. DMR Lexical Element Syntax

This section describes the lexical elements that occur in the DAP4 DMR.

Within the RELAXNG DAP4 grammar (Section ?) there are markers for occurrences of primitive type such as integers, floats, or strings (ignoring case). The markers typically look like this when defining an attribute that can occur in the DAP4 DMR.

|  |
| --- |
| <attribute name="Principal\_Investigator">  <datatype="dap4\_string"/>  </attribute> |

The "<data type="dap4\_string"/>" specifies the lexical class for the values that this attribute can have. In this case, the “Principal\_Investigator” attribute is defined to have a DAP4 string value. Similar notation is used for values occurring as text within an xml element.

The lexical specification later in this section defines the legal lexical structure for such items. Specifically, it defines the format of the following lexical items.

1. Constants, namely: string, float, integer, character, and opaque.
2. Identifiers
3. Fully qualified names (also referred to as FQNs) (see Section ?).

The specification is written using the extended POSIX regular expression notation [] with some additions.

1. Names are assigned to regular expressions using the notation “name = <regular expression>”
2. Named expressions can be used in subsequent regular expressions by using the notation “{name}”. Such occurrences are equivalent to textually substituting the expression associated with name for the “{name}” occurrence.

Notes:

1. The definition of {UTF8} is deferred to the next section.
2. Comments are indicated using the "//" notation. Standard xml escape formats (&x#DDD; or &<name>;) are assumed to be used as needed.
   * 1. Basic character set definitions

|  |
| --- |
| CONTROLS = [\x00-\x1F] // ASCII control characters  WHITESPACE = [ \r\n\t\f]+  HEXCHAR = [0-9a-zA-Z]  // ASCII printable characters  ASCII = [0-9a-zA-Z !"#$%&'()\*+,-./:;<=>?@[\\\]\\^\_`|{}~] |

* + 1. Ascii characters that may appear unescaped in Identifiers

This is assumed to be basically all ASCII printable characters except these characters: '.' '/' '"' ''' and '&'. Occurrences of these characters are assumed to be representable using the standard xml &<name>; notation (e.g. &amp;). In this expression, backslash is interpreted as an escape character.

|  |
| --- |
| IDASCII=[0-9a-zA-Z!#$%()\*+:;<=>?@\[\]\\^\_`|{}~] |

* + 1. The Numeric Constant Classes: integer and float

|  |
| --- |
| INTEGER = {INT}|{UINT}|{HEXINT}  INT = [+-][0-9]+{INTTYPE}?  UINT = [0-9]+{INTTYPE}?  HEXINT = {HEXSTRING}{INTTYPE}?  INTTYPE = ([BbSsLl]|"ll"|"LL")  HEXSTRING = (0[xX]{HEXCHAR}+)  FLOAT = ({MANTISSA}{EXPONENT}?)|{NANINF}  EXPONENT = ([eE][+-]?[0-9]+)  MANTISSA = [+-]?[0-9]\*\.[0-9]\*  NANINF = (-?inf|nan|NaN) |

* + 1. The String Constant Class

|  |
| --- |
| STRING = ([^"&<>]|{XMLESCAPE})\*  CHAR = ([^'&<>]|{XMLESCAPE})  URL = (http|https|[:][/][/][a-zA-Z0-9\-]+([.][a-zA-Z\-]+)+([:][0-9]+)?  ([/]([a-zA-Z0-9\-.\_?,'\\+&%#)\* |

* + 1. The Opaque Constant Class

|  |
| --- |
| OPAQUE = 0x([0-9A-Fa-f] [0-9A-Fa-f])+ |

* + 1. The Identifier Class

|  |
| --- |
| ID = {IDCHAR}+  IDCHAR = ({IDASCII}|{XMLESCAPE}|{UTF8})  XMLESCAPE = [&][#][0-9]+; |

* + 1. The Atomic Type Class

|  |
| --- |
| ATOMICTYPE = Char | Byte  | Int8 | UInt8 | Int16 | UInt16  | Int32 | UInt32 | Int64 | UInt64  | Float32 | Float64  | String | URL  | Enum | Opaque |

This list should be consistent with the atomic types in the grammar.

* + 1. The Fully Qualified Name Class

|  |
| --- |
| FQN = ([/]{EID})+([.]{EID})\* |
| EID = {EIDCHAR}+ |
| EIDCHAR = ({EIDASCII}|{XMLESCAPE}|{UTF8}) |
| EIDASCII = [0-9a-zA-Z!#$%()\*+:;<=>?@\[\]\\^\_`|{}~] |

This should be consistent with the definition in Section ?.

* + 1. DAP4 Type Definitions

The RELAXNG grammar references the following specific types. For each type, the following table give the lexical format as defined by the patterns previously given or by specific patterns as listed.

|  |  |
| --- | --- |
| RELAXNG Data Type Name | Lexical Pattern |
| dap4\_integer | {INTEGER} |
| dap4\_float | {FLOAT} |
| dap4\_char | {CHAR} |
| dap4\_string | {STRING} |
| dap4\_opaque | {OPAQUE} |
| dap4\_vdim | [\*] |
| dap4\_id | {ID} |
| dap4\_fqn | {FQN} |
| dap4\_uri | {URL} |
| dap4\_dim | [0-9]+ |

* + 1. Lexical Class Disambiguation

Note that the above lexical element classes are not disjoint. The type element “<datatype=…/>” should be sufficient to interpret the type within the DMR.

* + 1. UTF-8

The UTF-8 specification, <http://www.w3.org/2005/03/23-lex-U>, defines several ways to validate a UTF-8 string of characters.

The full (most correct) validating version of UTF8 character set is as follows.

|  |
| --- |
| UTF8 = ([\xC2-\xDF][\x80-\xBF])  | (\xE0[\xA0-\xBF][\x80-\xBF])  | ([\xE1-\xEC][\x80-\xBF][\x80-\xBF])  | (\xED[\x80-\x9F][\x80-\xBF])  | ([\xEE-\xEF][\x80-\xBF][\x80-\xBF])  | (\xF0[\x90-\xBF][\x80-\xBF][\x80-\xBF])  | ([\xF1-\xF3][\x80-\xBF][\x80-\xBF][\x80-\xBF])  | (\xF4[\x80-\x8F][\x80-\xBF][\x80-\xBF]) |

The lines of the above expression cover the UTF-8 characters as follows:

1. non-overlong 2-byte
2. excluding overlongs
3. straight 3-byte
4. excluding surrogates
5. straight 3-byte
6. planes 1-3
7. planes 4-15
8. plane 16

Note that ASCII and control characters are not included.

The above reference also defines some alternative regular expressions.

There is what is termed the partially relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6][\x80-\xBF])  | ([\xE0-\xEF][\x80-\xBF][\x80-\xBF])  | ([\xF0-\xF7][\x80-\xBF][\x80-\xBF][\x80-\xBF]) |

Second, there is what is termed the most-relaxed version of UTF8 defined by this regular expression.

|  |
| --- |
| UTF8 = ([\xC0-\xD6]...)|([\xE0-\xEF)...)|([\xF0 \xF7]...) |

Any conforming DAP4 implementation MUST use at least the most-relaxed expression for validating UTF-8 character strings, but MAY use either the partially-relaxed for full validatation expression.

1. DAP4 DMR Syntax as a RELAX NG Schema

Temporarily removed until dap4.rng settles down.