Chapter 12. Object Description Language Specification and Usage

The following provides a complete specification for Object Description Language (ODL), the language used to encode data labels for the Planetary Data System (PDS) and other NASA data systems. This standard contains a formal definition of the grammar semantics of the language. PDS specific implementation notes and standards are referenced in separate sections.

12.1 About the ODL Specification

This standard describes Version 2.1 of ODL. Version 2.1 of ODL supersedes Versions 0 and 1 of the language, which were used previously by the PDS and other groups. For the most part, ODL Version 2.1 is backwardly compatible with previous versions of ODL. There are, however, some features found in ODL Versions 0 and 1 that have been removed from or changed within Version 2. The differences between ODL versions are described in Section 12.7.

Following is a sample ODL data label describing a file and its contents:

```
/* File Format and Length */
       RECORD_TYPE         = FIXED_LENGTH
       RECORD_BYTES        = 800
       FILE_RECORDS        = 860
/* Pointer to First Record of Major Objects in File */
       ^IMAGE              = 40
       ^IMAGE_HISTOGRAM    = 840
       ^ANCILLARY_TABLE    = 842
/* Image Description */
       SPACECRAFT_NAME     = VOYAGER_2
       TARGET_NAME         = IO
       IMAGE_ID            = "0514J2-00"
       IMAGE_TIME          = 1979-07-08T05:19:11Z
       INSTRUMENT_NAME     = NARROW_ANGLE_CAMERA
       EXPOSURE_DURATION   = 1.9200 <SECONDS>
       NOTE                = "Routine multispectral longitude
                             coverage, 1 of 7 frames"
/* Description of the Objects Contained in the File */
       OBJECT              = IMAGE
        LINES              = 800
        LINE_SAMPLES       = 800
        SAMPLE_TYPE        = UNSIGNED_INTEGER
        SAMPLE_BITS        = 8
       END_OBJECT          = IMAGE

       OBJECT              = IMAGE_HISTOGRAM
        ITEMS              = 25
        ITEM_TYPE          = INTEGER
        ITEM_BITS          = 32
       END_OBJECT          = IMAGE_HISTOGRAM

       OBJECT              = ANCILLARY_TABLE
        ^STRUCTURE         = "TABLE.FMT"
        END_OBJECT         = ANCILLARY_TABLE
       END
```

12.1.1 Implementing ODL

Notes to implementers of software to read and write ODL-encoded data descriptions appear throughout the following sections. These notes deal with issues beyond language syntax and semantics, but are addressed to assure that software for reading and writing ODL will be uniform. The PDS, which is the major user of ODL-encoded data labels, has imposed additional implementation requirements for software used within the PDS. These PDS requirements are discussed below where appropriate.

12.1.1.1 Language Subsets

Implementers are allowed to develop software to read or write subsets of the ODL. Specifically, software developers may opt to:

- Eliminate support for the GROUP statement (see Section 12.4.5.2 for additional information)
- Not support pointer statements
- Not support certain types of data values

For every syntactic element supported by an implementation, the corresponding semantics, as spelled out in this chapter, must be fully supported. Software developers should be careful to assure that language features will not be needed for their particular applications before eliminating them. Documentation on label reading/writing software should clearly indicate whether or not the software supports the entire ODL specification and, if not, should clearly indicate the features not supported.

12.1.1.2 Language Supersets

Software for writing ODL must not provide or allow lexical or syntactic elements over and above those described below. With the exception of the PVL-specific extensions below, software for reading ODL must not provide or allow any extensions to the language.

12.1.1.3 PDS Implementation of PVL-Specific Extensions

PDS implementation of software for reading ODL may, in some cases, provide handling of lexical elements that are included in the CCSDS specification of the Parameter Value Language (PVL), which is a superset of ODL. Extensions handled by such software include:

- BEGIN_OBJECT as a synonym for the reserved word OBJECT
- BEGIN_GROUP as a synonym for the reserved word GROUP
- Use of the semicolon (;) as a statement terminator

These lexical elements are not supported by software that writes the ODL subset. They must either be removed (in the case of semicolons) or replaced (in the case of the BEGIN_OBJECT and BEGIN_GROUP synonyms) upon output.

12.1.2 Notation

The formal description of the ODL grammar is given below in Backus-Naur Form (BNF). Language elements are defined using rules of the following form:

defined_element ::= definition

where the definition is composed from the following components:

1. Lower case words, some containing underscores, are used to denote syntactic categories. For example:

   units_expression

   Whenever the name of a syntactic category is used outside of the formal BNF specification, spaces take the place of underscores (for example, units expression).

2. Boldface type is used to denote reserved identifiers. For example:

   object

   Special characters used as syntactic elements also appear in boldface type.

3. Square brackets enclose optional elements. Elements within brackets occur zero or one times.

4. Square brackets followed immediately by an asterisk or plus sign specify repeated elements. In the case of an asterisk, the elements in brackets may appear zero, one, or more times. In the case of a plus sign, the elements in brackets must appear at least once. The repetitions occur from left to right.

5. A vertical bar separates alternative elements.

6. If the name of any syntactic category starts with an italicized part, it is equivalent to the category name without the italicized part. The italicized part is intended to convey some semantic information. For example, both *object*_identifier and *units*_identifier are equivalent to identifier; *object*_identifier is used in places where the name of an object is required and *units*_identifier is used where the name of some unit of measurement is expected.

12.2 Character Set

The character set of ODL is the International Standards Organization's ISO 646 character set. The U.S. version of the ISO 646 character set is ASCII; the ASCII graphical symbols are used throughout this document. In other countries certain symbols have a different graphical representation.

The ODL character set is partitioned into letters, digits, special characters, spacing characters, format effectors and other characters:

character :: = letter | digit | special_character |
spacing_character | format_effector |
other_character

12.2.1 ODL Character Set - Letters

The letters are the uppercase letters A - Z and the lowercase letters a - z. ODL language elements are not case sensitive. Thus the following identifiers are equivalent:

- IMAGE_NUMBER
- Image_Number
- image_number

Case is significant inside of literal text strings, i.e., string “abc” is not the same as the string “ABC”.

12.2.2 ODL Character Set - Digits

The digits are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9.

12.2.3 ODL Character Set – Special Characters

The special characters used in ODL are:

Symbol	Name	Usage
=	Equals	The equals sign equates an attribute or pointer to a value.
{ }	Braces	Braces enclose an unordered set of values.
( )	Parentheses	Parentheses enclose an ordered sequence of values.
+	Plus	The plus sign indicates a positive numeric value.
-	Minus	The minus sign indicates a negative numeric value.
< >	Angle brackets	Angle brackets enclose a units expression associated with a numeric value.
.	Period	The period is the decimal place in real numbers.
" | Quotation Marks | Quotation marks denote the beginning and end of a text string value. Case is significant within the quotes of a text string. |
’ | Apostrophe | Apostrophes mark the beginning and end of a symbol value. Case is not significant within delimiting apostrophes (a.k.a. “single quotes”). |

_	Underscore	The underscore separates words within an identifier.
,	Comma	The comma separates individual values in a set or sequence.
/	Slant	The slant character indicates division in units expressions. The slant is also part of the comment delimiter.
*	Asterisk	The asterisk indicates multiplication in units expressions. Two asterisks in a row indicate exponentiation in units expressions. The asterisk is also part of the comment delimiter.
:	Colon	The colon is used in attribute assignment statements to separate a namespace_identifier from an attribute_identifier (see Section 12.4.2).
		The colon separates hours, minutes and seconds within a time value.
#	Crosshatch	Also known as “the pound sign”, this symbol delimits the digits in an integer number value expressed in notation other than base-10.
&	Ampersand	The ampersand denotes continuation of a statement onto another line.
^	Circumflex	The circumflex (or caret) indicates that a value is to be interpreted as a pointer.

12.2.4 ODL Character Set – Spacing Characters

Two characters, called the spacing characters, separate lexical elements of the language and can be used to format characters on a line:

Space
Horizontal Tabulation

12.2.5 ODL Character Set – Format Effectors

The following ISO characters are format effectors, used to separate ODL encoded statements into lines:

Carriage Return
Line Feed
Form Feed
Vertical Tabulation

The spacing characters and format effectors are discussed further in section 12.4.1 below. There are other characters in the ISO 646 character set that are not required to write ODL statements and labels. These characters may, however, appear within text strings and quoted symbolic literals:

! $ % ; ? @ [ ] ` | ~

12.2.6 ODL Character Set – Control Characters

The category of other characters also includes the ASCII control characters except for horizontal tabulation, carriage return, line feed, form feed and vertical tabulation (e.g., the control characters that serve as spacing characters or format effectors). As with the printing characters in this category, the control characters in this category can appear within a text string. The handling of control characters within text strings and symbolic literals is discussed in Section 12.3.3 below.

12.3 Lexical Elements

This section describes the lexical elements of ODL. Lexical elements are the basic building blocks of the ODL. Statements in the language are composed by stringing lexical elements together according to the grammatical rules presented in Section 12.4. The lexical elements of ODL are:

- Numbers
- Dates and Times
- Strings
- Identifiers
- Special symbols used for operators, etc.

There is no inherent limit on the length of any lexical element. However, software for reading and writing ODL may impose limitations on the length of text strings, symbol strings and identifiers. It is recommended that at least 32 characters be allowed for symbol strings and identifiers and at least 400 characters for text strings.

12.3.1 Numbers

ODL can represent both integer numbers and real numbers. Integer numbers are usually represented in decimal notation (“123”), but ODL also provides for integer values in other number bases (for example, “2#1111011#” is the binary representation of the decimal integer “123”). Real numbers can be represented in simple decimal notation (“123.4”) or in scientific notation (i.e., with a base 10 exponent: “1.234E2”).

12.3.1.1 Integer Numbers In Decimal Notation

An integer number in decimal notation consists of a string of digits optionally preceded by a number sign. A number without an explicit sign is always taken as positive.

integer :: = [sign] unsigned_integer
unsigned_integer :: = [digit] +

sign ::= **+** | **-**

Examples – *Decimal Integers*

0
123
+440
-150000

12.3.1.2 Integer Numbers In Based Notation

An integer number in based notation specifies the number base explicitly. The number base must be in the range 2 to 16, which allows for representations in the most popular number bases, including binary (base 2), octal (base 8) and hexadecimal (base 16). In general, for a number base X the digits 0 to X-1 are used. For example, in octal (base 8) the digits 0 to 7 are allowed. If X is greater than 10, then the letters A, B, C, D, E, F (or their lower case counterparts) are used as needed for the additional digits.

A based integer may optionally include a number sign. A number without an explicit sign is always taken as positive.

based_integer :: = radix **#** [sign] [extended_digit] + **#**
extended_digit :: = digit | letter
radix :: = unsigned_integer

Examples – *Based Integers*

2#1001011#
8#113#
10#75#
16#4B#
16#+4B#
16#-4B#

All but the last example above are equivalent to the decimal integer number 75. The final example is the hexadecimal representation of -75 decimal.

12.3.1.3 Real Numbers

Real numbers may be represented in floating-point notation (“123.4”) or in scientific notation with a base 10 exponent (“1.234E2”). A real number may optionally include a sign. Unsigned numbers are always taken as positive.

real :: = [sign] unscaled_real | [sign] scaled_real
unscaled_real :: = unsigned_integer**.** [unsigned_integer] | **.**unsigned_integer

scaled_real :: = unscaled_real exponent
exponent :: = **E** integer | **e** integer

Note that the letter 'E' in the exponent of a real number may appear in either upper or lower case.

Examples – *Real Numbers*

```
    0.0
  123.
+1234.56
   -.9981
  -1.E-3
31459e1
```

12.3.2 Dates and Times

ODL includes lexical elements for representing dates and times. The formats for dates and times are a subset of the formats defined by the International Standards Organization draft standard ISO/DIS 8601. (For information regarding PDS specific use of dates and times, see the *Date/Time* chapter in this document.)

12.3.2.1 Date and Time Values

Date and time scalar values represent a date, a time, or a combination of date and time:

date_time_value :: = date | time | date_time

The following rules apply to date values:

- The year must be Anno Domini. PDS requires a 4-digit year format be used (i.e., "2000", not "00").
- Month must be a number between 1 and 12.
- Day of month must be a number in the range 1 to 31, as appropriate for the particular month and year.
- Day of year must be in the range 1 to 365, or 366 in a leap year.

The following rules apply to time values:

- Hours must be in the range 0 to 23.
- Minutes must be in the range 0 to 59.
- Seconds, if specified, must be greater than or equal to 0 and less than 60.

The following rules apply to zone offsets within zoned time values:

- Hours must be in the range -12 to + 12 (the sign is mandatory).

- Minutes, if specified, must be in the range 0 to 59.

12.3.2.2 Implementation of Dates and Times

All ODL reading/writing software shall be able to handle any date within the 20th and 21st centuries. Software for writing ODL must always output full four-digit year numbers so that labels will be valid across century boundaries.

Times in ODL may be specified with unlimited precision, but the actual precision with which times will be handled by label reading/writing software is determined by the software implementers, based upon limitations of the hardware on which the software is implemented. Developers of label reading/writing software should document the precision to which times can be represented.

Software for writing ODL must not output local time values, since a label may be read in a time zone other than where it was written. Use either the UTC or zoned time format instead.

12.3.2.3 PDS Implementation of Dates and Times

PDS software for reading ODL labels interprets label times as UTC times. On output, a “Z” will be appended to label times.

12.3.2.4 Dates

Dates can be represented in two formats: as year and day of year; or as year, month and day of month.

date	:: = year_doy | year_month_day
year_doy	:: = year-doy
year_month_day	:: = year-month-day
year	:: = unsigned_integer
month	:: = unsigned_integer
day	:: = unsigned_integer
doy	:: = unsigned_integer

Examples – *Dates*

1990-07-04
1990-158
2001-001

12.3.2.5 Times

Times are represented as hours, minutes and (optionally) seconds using a 24-hour clock. Times may be specified in Universal Time Coordinated (UTC) by following the time with the letter Z (for Zulu, a common designator for Greenwich Mean Time). Alternately, the time may be

referenced to any time zone by following the time with a number that specifies the offset from UTC. Most time zones are an integral number of hours from Greenwich, but some are different by some non-integral time; both can be represented in the ODL. A time that is not followed by either the Zulu indicator or a time zone offset is assumed to be a local time.

```
time          :: = local_time | utc_time |  zoned_time
local time    :: = hour_min_sec
utc_time      :: = hour_min_sec Z
zoned_time    :: = hour_min_sec zone_offset
hour_min_sec :: = hour: minute [:second]
zone_offset   :: = sign hour [: minute]
hour          :: = unsigned_integer
minute        :: = unsigned_integer
second        :: = unsigned_integer | unscaled_real
```

Note that either an integral or a fractional number of seconds can be specified in a time value.

Examples – *Times*

```
12:00
15:24:12Z
01:10:39.4575+07  (time offset of 7 hours from UTC)
```

12.3.2.5.1 Combining Date and Time

A date and time can be specified together using the format below. Either of the two date formats can be combined with any time format - UTC, zoned or local.

```
date_time::=date T time
```

The letter T separating the date from the time may be specified in either upper or lower case. Note that, because this is a lexical element, spaces may not appear within a date, within a time or before or after the letter T.

Examples – *Date/Times*

```
1990-07-04T12:00
1990-158T15:24:12Z
2001-001T01:10:39.457591+7
```

12.3.3 Strings

There are two kinds of string elements in ODL: text strings and symbol strings.

12.3.3.1 Text Strings

Text strings are used to hold arbitrary strings of characters.

quoted_text ::= "[character]*"

The empty string — a quoted text string with no characters within the delimiters — is allowed.

A quoted text string may not contain the quotation mark, which is reserved to be the text string delimiter. A quoted text string may contain format effectors, hence it may span multiple lines in a label: the lexical element begins with the opening quotation mark and extends to the closing quotation mark, even if the closing mark is on a following line. The rules for interpreting the characters within a text string, including format effectors, are given in the subsection on string values in Section 12.5.3.

12.3.3.2 Symbol Strings

Symbol strings are sequences of characters used to represent symbolic values. For example, an image ID may be a symbol string like 'J123-U2A', or a camera filter might be a symbol string like 'UV1'.

quoted_symbol ::= '[character]+'

A symbol string may not contain any of the following characters:

- The apostrophe, which is reserved to be the symbol string delimiter
- Format effectors, which means that a symbol string must fit on a single line
- Control characters

12.3.4 Identifiers

Identifiers are used as the names of objects, attributes and units of measurement. They can also appear as the value of a symbolic literal.

Identifiers are composed of letters, digits, and underscores. Underscores are used to separate words in an identifier. The first character of an identifier must be a letter. The last character may not be an underscore.

identifier : : = letter [letter | digit | _letter | _digit]*

Because ODL is not case sensitive, lower case characters in an identifier can be converted to their upper case equivalent upon input to simplify comparisons and parsing.

Examples – *Identifiers*

VOYAGER
VOYAGER_2

BLUE_FILTER
USA_NASA_PDS_1_0007
SHOT_1_RANGE_TO_SURFACE

12.3.4.1 Reserved Identifiers

A few identifiers have special significance in ODL statements and are therefore reserved. They cannot be used for any other purpose (specifically, they may not be used to name objects or attributes):

end	end_group	end_object
group	object	begin_object

12.3.5 Special Characters

ODL is a simple language and it is usually clear where one lexical element ends and another begins. Spacing characters or format effectors may appear before a lexical element, between any pair of lexical elements, or after a lexical element without changing the meaning of a statement.

Some lexical elements incorporate special characters (e.g., the decimal point in real numbers or the quotation marks that delimit a text string). Some special characters are also lexical elements in their own right. These are:

=	The equals sign is the assignment operator.
,	The comma separates the elements of an array or a set.
*	The asterisk serves as the multiplication operator in units expressions.
/	The slant serves as the division operator within units expressions.
^	The circumflex denotes a pointer to an object.
<>	The angle brackets enclose units expressions.
()	The parentheses enclose the elements of a sequence.
{ }	The braces enclose the elements of a set.

The following two-character sequence is also a lexical element.

**	Two adjacent asterisks are the exponentiation sign within units expressions.

12.4 Statements

An ODL-encoded label is made up of a sequence of zero, one, or more statements followed by the reserve identifier **end**.

label ::= [statement]*
end

The body of a label is built from four types of statements:

statement :: = attribute_assignment_statement |
pointer_statement |
object_statement |
group_statement

Each of the four types of statements is discussed below.

12.4.1 Lines and Records

Labels are also typically composed of lines, where each line is a string of characters terminated by a format effector or a string of adjacent format effectors. The following recommendations are given for how software that writes ODL should format a label into lines:

- There should be at most one statement on a line, although a statement may be more than a single line in length. As noted in Section 12.3.5 above, format effectors may appear before, after or between the lexical elements of a statement without changing the meaning of the statement. For example, the following statements are identical in meaning:

```
FILTER_NAME          =   {RED, GREEN, BLUE}

FILTER_NAME          =   {RED,
                          GREEN,
                          BLUE}
```

- Each line should end with a carriage return character followed immediately by a line feed character. This sequence is an end-of-line signal for most computer operating systems and text editors.

- The character immediately following the **END** statement must be either an optional spacing character or format effector, such as a space, line feed, carriage return, etc.

A line may include a comment. A comment begins with the two characters "/*" and ends with the two characters "*/". A comment may contain any character in the ODL character set except format effectors, which are reserved to mark the end of line (i.e., comments may not be more than one line long). Comments are ignored when parsing an ODL label. When the comment delimiters ("/*" and "*/") appear within a text string, they are not interpreted as a comment - they are simply part of the text string. For example, in the following example the comment will be included as part of the text string:

```
NOTE = "All good men come to the    /* Example of incorrect comment*/
        aid of their party"
```

Any characters on a line following a comment are ignored.

In some computer systems files are divided into records. Software for writing and reading ODL-encoded labels in record-oriented files should adhere to the following rules:

- A line of an ODL-encoded label may not cross a record boundary, i.e., each line should be contained within a single record. Any space left over at the end of a record after the last line in that record should be set to all space characters.
- The remainder of the record that contains the END statement is ignored. The data portion of the file begins with the next record in sequence.

12.4.2 Attribute Assignment Statement

The attribute asignment statement is the most common type of statement in ODL and is used to specify the value for an attribute of an object. The value may be a singular scalar value, an ordered sequence of values, or an unordered set of values.

The attribute assignment statement may optionally contain a namespace_identifier. When a namespace_identifier is prepended to the element_identifier statement, it indicates that the element_identifier has a local definition within the context indicated by the namespace_identifier.

assignment_statement ::= *attribute*_identifier = value

where *attribute*_identifier::= *element*_identifier |
*namespace*_identifier:*element*_identifier

The syntax and semantics of values are given in Section 12.5.

Examples – *Assignment Statements*

```
RECORD_BYTES              = 800
TARGET_NAME               = JUPITER
SOLAR_LATITUDE            = (0.25 <DEG>, 3.00 <DEG>)
FILTER_NAME               = {RED,
                             GREEN,
                             BLUE}
```

Examples – *Assignment Statements that use namespace_identifier*

```
CASSINI:TARGET_NAME = JUPITER
MRO:SOLAR_LATITUDE  = (0.25 <DEG>, 3.00 <DEG>)
VOYAGER:FILTER_NAME = { RED, GREEN, BLUE }
```

12.4.3 Pointer Statement

The pointer statement indicates the location of an object.

pointer_statement :: = ^*object*_identifier = value

As with the attribute assignment statement, the value may be a scalar value, an ordered sequence of values, or an unordered set of values.

A common use of pointer statements is to reference a file containing an auxiliary label. For example:

```
^STRUCTURE = "TABLE.FMT"
```

This is a pointer statement pointing to a file named “TABLE.FMT” that contains a description of the structure of the ancillary table from our sample label. Another use of the pointer statement is to indicate the position of an object within another object. This is often used to indicate the position of major objects within a file. The following examples are from the sample label in Section 12.1:

```
^IMAGE                = 40
^IMAGE_HISTOGRAM      = 840
^ANCILLARY_TABLE      = 842
```

The first pointer statement above indicates that the image is located starting at the 40th record from the beginning of the present file. If an integer value is used to indicate the relative position of an object, the units of measurement of position are determined by the nature of the object. For files, the default unit of measurement is records. Alternatively, a units expression can be specified for the integer value to indicate explicitly the units of measurement for the position. For example, this pointer:

```
^IMAGE          = 10200 <BYTES>
```

indicates that the image starts 10,200 bytes from the beginning of the file.
The object pointers above reference locations in the same files as the label containing the pointer. Pointers may also reference either byte or record locations in data files that are detached, or separate, from the label file:

```
^IMAGE          = ("IMAGE.DAT", 10)
^HEADER         = ("IMAGE.DAT", 512 <BYTES>)
```

12.4.4 OBJECT Statement

The OBJECT statement begins the description of an object. The description typically consists of a set of attribute assignment statements defining the values of the object’s attributes. If an object is itself composed of other objects, then OBJECT statements for the component objects are nested within the object’s description. There is no limit to the depth to which OBJECT

statements may be nested.

The format of the OBJECT statement is:

object_statement ::= **object** = *object*_identifier
[statement]*
end_object [= *object*_identifier]

The object identifier gives a name to the particular object being described. For example, in a file containing images of several planets, the image object descriptions might be named VENUS_IMAGE, JUPITER_IMAGE, etc. The object identifier at the end of the OBJECT statement is optional, but if it appears it must match the name given at the beginning of the OBJECT statement.

12.4.4.1 Implementation of OBJECT Statements

It is recommended that all software for writing ODL include the object identifier at the end as well as the beginning of every OBJECT statement.

12.4.5 GROUP Statement

The GROUP statement is used to group together statements that are not components of a larger object. For example, in a file containing many images, the group BEST_IMAGES might contain the object descriptions of the three highest quality images. The three image objects in the BEST_IMAGES group don't form a larger object: all they have in common is their superior quality.

The GROUP statement is also used to group related attributes of an object. For example, if two attributes of an image object are the time at which the camera shutter opened and closed, then the two attributes might be grouped as follows:

```
GROUP       = SHUTTER_TIMES
  START     = 12:30:42.177
  STOP      = 14:01:29.265
END_GROUP   = SHUTTER_TIMES
```

The format of the group statement is as follows:

group_statement ::= **group** = *group*_identifier
[statement]*
end_group [= *group*_identifier]

The group identifier gives a name to the particular group, as shown in the example for shutter times above. The object identifier at the end of the GROUP statement is optional, but if it appears it must match the name given at the beginning of the GROUP statement. Groups may be

nested within other groups. There is no limit to the depth to which groups can be nested.

As opposed to the above ODL implementation, the PDS applies the following restrictions to the use of GROUPS:

1. The GROUP structure may only be used in a data product label which also contains one or more data OBJECT definitions.
2. The GROUP statement must contain only attribute assignment statements, include pointers, or related information pointers (i.e., no data location pointers).
3. GROUP statements may not be nested.
4. GROUP statements may not contain OBJECT definitions.
5. Only PSDD elements may appear within a GROUP statement.
6. The keyword contents associated with a specific GROUP identifier must be identical across all labels of a single data set (with the exception of the “PARAMETERS” GROUP, as explained .

Use of the GROUP structure must be coordinated with the responsible PDS discipline Node.

12.4.5.1 Implementation of GROUP Statements

It is recommended that all software for writing ODL include the group identifier at the end as well as the beginning of every GROUP statement.

12.4.5.2 PDS Usage of GROUP

Although ODL includes the GROUP statement, the PDS does not recommend its use because of confusion concerning the difference between OBJECT and GROUP.

12.5 Values

ODL provides scalar values, ordered sequences of values, and unordered sets of values.

value :: = scalar_value | sequence_value | set_value

A scalar value consists of a single lexical element:

scalar_value :: = numeric_value |
date_time_value |
text_string_value |
symbol_value

The format and use of each of these scalar values are discussed in the sections below.

12.5.1 Numeric Values

A numeric scalar value is either a decimal or based integer number, or a real number. A numeric scalar value may optionally include a units expression.

```
numeric_value :: =   integer [units_expression] |
                     based_integer [units_expression] |
                     real [units_expression]
```

12.5.2 Units Expressions

Many of the values encountered in scientific data are measurements of something. In most computer languages, only the magnitude of a measurement is represented, without the units of measurement. ODL, however, can represent both the magnitude and the units of a measurement. A units expression has the following format:

```
units_expression    :: = < units_factor [mult_op units_factor] * >
units_factor        :: = units_identifier [exp_op integer]
mult_op             :: = * | /
exp_op              :: = **
```

A units expression is always enclosed within angle brackets. The expression may consist of a single units identifier like “KM”,for kilometers, or “SEC”, for seconds (for example, “1.341E6 <KM>” or “1.024 <SEC>”). More complex units can also be represented; for example, the velocity “3.471 <KM/SEC>” or the acceleration “0.414 < KM/SEC/SEC>”. There is often more than one way to represent a unit of measure. For example:

```
0.414          <KM/SEC/SEC>

0.414          <KM/SEC**2>

0.414          <KM*SEC**-2>
```

are all valid representations of the same acceleration. The following rules apply to units expressions:

- The exponentiation operator can specify only a decimal integer exponent. The exponent value may be negative, which signifies the reciprocal of the units. For example, “60.15 <HZ>” and “60.15 <SEC**-1>” are both ways to specify a frequency.
- Individual units may appear in any order. For example, a force might be specified as either “1.55 <GM*CM/ SEC**2>” or “1.55 <CM*GM/SEC**2>”.

12.5.2.1 Implementation of Numeric Values

There is no defined maximum or minimum magnitude or precision for numeric values. In general, the actual range and precision of numbers that can be represented will be different for each kind of computer used to read or write an ODL-encoded label. Developers of software for

reading/writing ODL should document the following:

- The largest magnitude positive and negative integers that can be represented
- The largest magnitude positive and negative real numbers that can be represented
- The minimum number of significant digits that a real number can be guaranteed to have without loss of precision. This is to account for the loss of precision that can occur when representing real numbers in floating point format within a computer. For example, a 32-bit floating-point number with 24 bits for the mantissa can guarantee at most 6 significant digits will be exact (the seventh and subsequent digits may not be exact because of truncation and round-off errors).

If software for reading ODL encounters a numeric value too large to be represented, the software must report an error to the user.

12.5.3 Text String Values

A text string value consists of a text string lexical element:

text_string_value :: = quoted_text

12.5.3.1 Implementation of String Values

A text string read in from a label is reassembled into a string of characters. The way in which the string is broken into lines in a label does not affect the format of the string after it has been reassembled. The following rules are used when reading text strings:

- If a format effector or a sequence of format effectors is encountered within a text string, the effector (or sequence of effectors) is replaced by a single space character, unless the last character is a hyphen (dash) character. Any spacing characters at the end of the line are removed and any spacing characters at the beginning of the following line are removed. This allows a text string in a label to appear with the left and right margins set at arbitrary points without changing the string value. For example, the following two strings are the same:

  “To be or not to be”

  and

  “To be or
  not to be"

- If the last character on a line prior to a format effector is a hyphen (dash) character, the hyphen is removed with any spacing characters at the beginning of the following line. This follows the standard convention in English of using a hyphen to break a word across lines. For example, the following two strings are the same:

"The planet Jupiter is very big"

and

"The planet Jupi-
ter is very big"

- Control codes, other than the horizontal tabulation character and format effectors, appearing within a text string are removed.

12.5.3.1.1 PDS Text String Formatting Conventions

The PDS defines a set of format specifiers that can be used in text strings to indicate the formatting of the string on output. These specifiers can be used to indicate where explicit line breaks should be placed, and so on. The format specifiers are:

\n	Indicates that an end-of-line sequence should be inserted.
\t	Indicates that a horizontal tab character should be inserted.
\f	Indicates that a page break should be inserted.
\v	Must be used in pairs, begin and end. Interpreted as verbatim.
\\	Used to place a backslash in a text string.

For example, the string

"This is the first line \n and this is the second line."

will print as:

This is the first line
and this is the second line.

Note: These format specifiers have meaning only when a text string is printed - not when the string is read in or stored.

12.5.4 Symbolic Literal Values

A symbolic value may be specified as either an identifier or a symbol string:

symbolic-value :: = identifier | quoted_symbol

The following statements assign attributes to symbolic values specified by identifiers:

```
TARGET_NAME       =  IO
SPACECRAFT_NAME   =  VOYAGER_2
SPACECRAFT_NAME   =  'VOYAGER-2'
SPACECRAFT_NAME   =  'VOYAGER 2'
REFERENCE_KEY_ID  =  SMITH1997
REFERENCE_KEY_ID  =  'LAUREL&HARDY1997'
```

The quotes must be used if the symbolic value does not have the proper format for an identifier or if it contains characters not allowed in an identifier. For example, the value 'FILTER_+_7' must be enclosed within quotes, since this would not be a legal ODL identifier. Similarly, the symbolic value 'U13-A4B' must be in quotes because it contains a special character (the dash) not allowed in an identifier. There is no harm in putting a legal identifier within quotes. For example:

```
SPACECRAFT_NAME = 'VOYAGER_2'
```

is equivalent to the second example in the list above.

Symbolic values may not contain format effectors, i.e., they may not cross a line boundary.

12.5.4.1 Implementation of Symbolic Literal Values

Symbolic values are converted to upper case on input. This means that a lowercase string is converted to the equivalent uppercase string; as in the following example:

Original string: `SPACECRAFT_NAME = 'Voyager_2'`
Converted string: `SPACECRAFT_NAME = 'VOYAGER_2'`

12.5.4.2 PDS Convention for Symbolic Literal Values

Since the current use of the ODL within the PDS does not require syntactic differentiation between symbols and text strings, PDS prefers that double quotation marks (") be used instead of apostrophes around symbol strings.

12.5.5 Sequences

A sequence represents an ordered set of values. It can be used to represent arrays and other kinds of ordered data. Only one- and two-dimensional sequences are allowed.

sequence_value :: = sequence_1D | sequence_2D
sequence_1D :: = **(**scalar_value [**,** scalar_value]***)**
sequence_2D :: = **(**[sequence _1D] +**)**

A sequence may have any kind of scalar value for its members. It is not required that all the members of the sequence be of the same type. Thus a sequence may represent a heterogeneous record. Each member of a two-dimensional sequence is a one-dimensional sequence. This can be used, for example, to represent a table of values. The order in which members of a sequence appear must be preserved. There is no upper limit on the number of values in a sequence.

For example: `AVERAGE_ECCENTRICITY = (0,1,2,3,4,5,9)`

12.5.6 Sets

Sets are used to specify unordered values drawn from some finite set of values.

set_value :: = **{**scalar_value [**,** scalar_value]***}** | **{}**

Note that the empty set is allowed: The empty set is denoted by opening and closing brackets with nothing except optional spacing characters or format effectors between them.

The order in which the members appear in the set is not significant and the order need not be preserved when a set is read and manipulated. There is no upper limit on the number of values in a set.

Example

```
FILTER_NAME  = { RED, BLUE, GREEN, HAZEL }
```

12.5.6.1 PDS Restrictions on Sets

The PDS allows only symbol values and integer values within sets.

12.6 ODL Summary

Character Set (Section 12.2)

ODL uses the ISO 646 character set (the American version of the ISO 646 standard is ASCII). The ODL character set is partitioned as follows:

character	: : = letter | digit | special_character | spacing_character | format_effector | other_character
letter	: : = A-Z | a-z
digit	: : = 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8| 9
special_character	: : = { | } | (|) | + | - | . | " | ’ | = | _ | , | / | * | : | # | & | ^ | < | >

spacing_character : : = space | horizontal tabulation
format_effector : : = carriage return | line feed |
form feed | vertical tabulation
other_character : : = ! | $ | % | ; | ? | @ | [|] | ` | ~ |
vertical bar | other control characters

Lexical Elements (Section 12.3)

integer : : = [sign] unsigned_integer
unsigned_integer : : = [digit]+
sign : : = **+** | **-**
based_integer : : = radix **#** [sign] [extended_digit]+ **#**
extended_digit : : = digit | letter
radix : : = unsigned_integer
real : : = [sign] unscaled_real | [sign] scaled_real
unscaled_real : : = unsigned_integer **.** [unsigned_integer] |
. unsigned_integer
scaled_real : : = unscaled_real exponent
exponent : : = **E** integer | **e** integer
date : : = year_doy | year_month_day
year_doy : : = year **-** doy
year_month_day : : = year **-** month **-** day
year : : = unsigned_integer
month : : = unsigned_integer
day : : = unsigned_integer
doy : : = unsigned_integer
time : : = local_time | utc_time | zoned_time
local_time : : = hour_min_sec
utc_time : : = hour_min_sec **Z**
zoned_time : : = hour_min_sec zone_offset
hour_min_sec : : = hour **:** minute [**:** second]
zone_offset : : = sign hour [**:** minute]
hour : : = unsigned_integer
minute : : = unsigned_integer
second : : = unsigned_integer | unscaled_real
date_time : : = date **T** time
quoted_text : : = “[character]*”
quoted_symbol : : = ‘[character]+**’**
identifier : : = letter [letter | digit | _letter | **_**digit]*

Statements (Section 12.4)

label : : = [statement]*
end
statement : : = assignment_stmt | pointer_stmt |

```
                                    object_stmt | group_stmt
assignment_stmt          : : = element_identifier = value |
                                namespace_identifier:element_identifier = value
      pointer_stmt       : : = ^object_identifier = value
      object_stmt        : : = object = object_identifier
                               [statement]*
                               end_object [= object_identifier]
      group_stmt         : : = group = group_identifier
                               [statement]*
                               end_group [= group_identifier]
```

Values (Section 12.5)

```
      value              : : = scalar_value | sequence_value |   set_value
      scalar_value       : : = numeric_value | date_time_value |
                                text_string_value | symbolic_value
      numeric_value      : : = integer [units_expression] |
                               based_integer [units_expression] |
                               real [units_expression]
      units_expression   : : =<units_factor[mult_op units_factor]* >
      units_factor       : : = units_identifier [exp_op integer]
      mult_op            : : = * | /
      exp_op             : : = **
      date_time_value    : : = date | time  | date_time
      text_string_value  : : = quoted_text
      symbolic_value     : : = identifier | quoted_symbol
      sequence_value     : : =sequence_lD | sequence_2D
      sequence_1D        : : = (scalar_value [, scalar_value]*)
      sequence_2D        : : = ([sequence_lD]+)
      set_value          : : = { scalar_value [,scalar_value]* } | { }
```

12.7 Differences Between ODL Versions

This section summarizes the differences between the current Version 2 of ODL and the previous Versions 0 and 1. Software can be constructed to read all three versions of ODL, however it is important that software for writing labels only write labels that conform to ODL Version 2.

12.7.1 Differences from ODL Version 1

Version 1 labels were used on the *Voyager to the Outer Planets* CD-ROM disks and many other data sets. Version 1 did not include the GROUP statement and had more restrictive definitions for sets, which were limited to integer or symbolic literal values, and sequences, which were limited to arrays of homogeneous values. The following sections detail non-compatible differences and how they can be handled by software writers.

12.7.1.1 Ranges

Version 1 of ODL had a specific notation for integer ranges:

range_value :: = integer..integer

This notation is not allowed in ODL Version 2, though parsers may still recognize the 'double-dot' range notation. On output, a range is now encoded as a two value sequence, with the low-value of the range being the first element of the sequence and the high-value being the second element of the sequence.

12.7.1.1.1 Delimiters in Sequences and Sets

In Version 1 the individual values in sets and sequences could be separated by a comma or by a spacing character. As of Version 2, a comma is required. Parsers may allow spacing characters between values rather than commas. Software that writes ODL should place commas between all values in a sequence or set.

12.7.1.1.2 Exponentiation Operator in Units Expressions

In Version 1 of ODL the circumflex character (^) was used as the exponentiation operator in units expressions rather than the two-asterisk sequence (**). Parsers may still allow the circumflex to appear within units expressions as an exponentiation operator. Software for writing ODL should use only the ** notation.

12.7.2 Differences from ODL Version 0

Version 0 of ODL was developed for and used on the *PDS Space Science Sampler* CD-ROM disks. The major difference between this and subsequent versions is that Version 0 did not include the OBJECT statement. All of the attributes specified in a label described a single object: the file that contained the label (or that was referenced by a pointer).

12.7.2.1 Date–Time Format

ODL Version 0 was produced prior to the space community's acceptance of the ISO/DIS 8601 standard for dates and time and it uses a different date and date-time format. The format for Version 0 dates and date-times is as follows:

date	:: = year / month / day_of_month | year / day_of_year
date_time	:: = date - time zone
zone	:: = < identifier>

The options for time specification in ODL Version 0 are a subset of those in Version 2. Consequently, parsers that handle Version 2 time formats will also handle Version 0 times.

12.7.3 ODL/PVL Usage

The concept for a Parameter Value Language/Format (PVL) is being formalized by the Consultative Committee for Space Data Systems (CCSDS). It is intended to provide a human readable data element/value structure to encode data for interchange. The CCSDS version of the PVL specification is in preliminary form.

Some organizations that deal with the PDS have accepted PVL as their standard language for product labels. PVL is a superset of ODL, so some PVL constructs are not supported by the PDS. In addition, some ODL constructs may be interpreted differently by PVL software.

The ODL/PVL usage standard defines restrictions on the use of ODL/PVL in archive quality data sets. These restrictions are intended to ensure the compatibility of PVL with ODL and existing software.

1. A label constructed using PVL may be attached - embedded in the same file as the data object it describes, or detached - residing in a separate file and pointing to the data file the label describes.

2. All statements must be terminated by a <CR> <LF> pair. Semicolons may not be used to terminate statements.

3. Only alphanumeric characters and the underscore character may be used in data elements and undelimited text values (literals). In addition, data elements and undelimited text values must begin with a letter.

4. Keywords must be 30 characters or less in length.

5. Keywords and standard values must be in upper case. Literals and strings may be in upper case, lower case, or mixed case.

6. Comments must be contained on a single line, and a comment terminator (*/) must be used. Comments may not be embedded within statements. Comments may not be used on the same line as any statement if the comment precedes the statement. Comments may be on the same line as a statement if the comment follows the statement and is separated from the statement by at least one white space, but this is not recommended.

7. Text values that cross line boundaries must be enclosed in double quotation marks (“ ”).

8. Values that consist only of letters, numbers, and underscores and that begin with a letter may be used without quotation marks. All other text values must be enclosed in either single (‘ ’) or double (“ ”) quotation marks.

9. Sequences are limited to two dimensions. Null (empty) sequences are not allowed. Sets are limited to one dimension. In other words, sets and sequences may not be used inside a set.

10. Only the OBJECT, END_OBJECT, GROUP and END_GROUP aggregation markers may be used.

11. Unit expressions are only allowed following numeric values (i.e., "DATA_ELEMENT = 7 <BYTES>" is valid. but "DATA_ELEMENT = MANY <METERS>" is not).

12. Unit expressions may include only alphanumeric characters, the underscore, and the symbols "*", "/", "(", ")", and "**" (the last representing exponentiation).

13. Signs may not be used in non-decimal numbers (i.e., "2#10001#" is valid, but "-2#10001#" and "2#-10001#" are not). Only the bases 2, 8, and 16 may be used for non-decimal numbers.

14. Alternate time zones (e.g., YYYY-MM-DDTHH:MM:SS.SSS + HH:MM) may not be used in a PDS label. The only allowed time formats are

    (1) YYYY-MM-DDTHH:MM:SS.SSS.
    (2) YYYY-DDDTHH:MM:SS.SSS.

    See Section 7.3.2(6) for a more detailed description.

15. Values in integral parts of dates and times must be padded on the left with zeroes as necessary to fill the field. In other words, the first of April in the year 2001 must be written as "2001-04-01", *not* "2001-4-1"

16. An END statement must conclude each ODL/PVL statement list.

The following are guidelines for formatting ODL/PVL expressions.

1. The assignment symbol (=) must be surrounded by blanks.

2. Assignment symbols (=) should be aligned if possible.

3. Keywords placed inside an aggregator (OBJECT or GROUP) must be indented with respect to the OBJECT and END_OBJECT or GROUP and END_GROUP statements which enclose them.

4. PDS label lines must be 80 characters or less in length, including the end-of-statement (i.e., <CR> <LF>) delimiter. (Note that while 80 characters can be

displayed on most screens, some editors and databases will wrap or truncate lines that exceed 72 characters.)

5. Horizontal tab characters may not be used in PDS labels. Although both ODL and PVL allow the use of these characters some simple parsers cannot handle them. The equivalent number of space characters should be used instead.